# 19th International Symposium on Experimental Algorithms

**SEA 2021, June 7–9, 2021, Nice, France**

Edited by

David Coudert
Emanuele Natale

LIPICS

*Editors*

**David Coudert**
I3S (CNRS-UCA)/Inria, Sophia Antipolis, France
david.coudert@inria.fr

**Emanuele Natale**
Université Côte d'Azur, CNRS, France
emanuele.natale@inria.fr

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

**ISBN 978-3-95977-185-6**

*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/978-3-95977-185-6.

*Publication date*
June, 2021

*Bibliographic information published by the Deutsche Nationalbibliothek*
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.SEA.2021.0

## LIPIcs – Leibniz International Proceedings in Informatics

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

**ISSN 1868-8969**

**https://www.dagstuhl.de/lipics**

# Contents

# Preface

We are pleased to present the collection of papers accepted for presentation at the 19th edition of the International Symposium on Experimental Algorithms (SEA 2021), originally planned to be held in Nice (France) from June 6 to 9, 2021. Unfortunately, due to the health situation in France and in the rest of world, we were unable to organize a physical event, and the symposium will be again a virtual event, as it has been last year.

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. The main theme of the symposium is the role of experimentation and of algorithm engineering techniques in the design and evaluation of algorithms and data structures. Submissions to SEA are requested to present significant contributions supported by experimental evaluation, methodological issues in the design and interpretation of experiments, the use of heuristics and meta-heuritics, or application-driven case studies that deepen the understanding of the complexity of a problem. A main goal of SEA is also the creation of a friendly environment that can lead to and ease the establishment or strengthening of scientific collaborations and exchanges among attendees. For this reason, the symposium solicits high-quality original research papers (including significant work-in-progress) on any aspect of experimental algorithms.

Each submission to SEA 2021 was reviewed by at least three Program Committee members or external reviewers. After a careful peer review and evaluation process, 23 papers were accepted for presentation and for inclusion in the LIPIcs proceedings, according to the reviewers' recommendations. The acceptance rate was 56%. The scientific program of the symposium also includes presentations by three keynote speakers: Dominik Kempa (Johns Hopkins University, USA), Petra Mutzel (University of Bonn, Germany) and Blair D. Sullivan (University of Utah, USA).

The 19th edition of SEA was organized by the I3S laboratory (Université Côte d'Azur, CNRS). We thank Corinne Julien-Haddad for her help in the organization of this symposium. We also thanks Université Côte d'Azur, the research center Inria Sophia Antipolis - Méditerranée, and the city of Nice (Comité Doyen Lépine) for their financial support. We also thank the SEA steering committee for giving us the opportunity to host SEA 2021. We express our gratitude to the EasyChair platform. Thanks are also due to the editors of the ACM Journal of Experimental Algorithmics for their interest in hosting a special issue of the best papers presented at SEA 2021. Finally, we express our gratitude to the members of the Program Committee for their support, collaboration, and excellent work.

Nice, June 6 2021
David Coudert and Emanuele Natale

## Steering Committee

# Organization

## Program Chairs

- David Coudert (Université Côte d'Azur, INRIA, CNRS, I3S, France)
- Natale, Emanuele (Université Côte d'Azur, INRIA, CNRS, I3S, France)

## Program Committee

- Thomas Bläsius (KIT, Germany)
- Maike Buchin (Ruhr-Universität Bochum, Germany)
- Eranda Çela (Graz University of Technology, Austria)
- Gianlorenzo D'Angelo (GSSI, Italy)
- Simone Faro (University of Catania, Italy)
- Paola Festa (University of Naples Federico II, Italy)
- Cyril Gavoille (Université de Bordeaux, France)
- Loukas Georgiadis (University of Ioannina, Greece)
- Oguzhan Kulekci (Istanbul Technical University, Turkey)
- Erwan Le Merrer (INRIA Rennes, France)
- Stefano Leucci (Università dell'Aquila, Italy)
- Henning Meyerhenke (Humboldt-Universität zu Berlin, Germany)
- Shin-Ichi Minato (Hokkaido University, Japan)
- Gonzalo Navarro ( University of Chile, Chile)
- Marcin Pilipczuk (University of Warsaw, Poland)
- Ely Porat (Bar-Ilan University, Israel)
- Michael Poss (LIRMM, France)
- Mauricio G. C. Resende (Amazon.com, USA)
- Celso C. Ribeiro (U. Federal Fluminense, Brasil)
- Kunihiko Sadakane (Univesity of Tokyo, Japan)
- Stefan Schmid (University of Vienna, Austria)
- Celine Scornavacca (ISEM, France)
- Bertrand Simon (CC-IN2P3, France)
- Sabine Storandt (University of Konstanz, Germany)
- Hisao Tamaki (Meiji University, China)
- Annegret K. Wagler (Isima - Limos, France)

## External Reviewers

Ali Al Zoobi, Amihood Amir, Eugenio Angriman, Diego Arroyuelo, Berenger Bramas, Luciana Buriol, Arthur Carvalho Walraven da Cunha, Richard Chen, Dustin Cobas, Francesco d'Amore, Diego Delle Donne, Mattia D'Emidio, Yuanyuan Dong, Stefan Edelkamp, Konstantinos Giannis, Shay Golan, Adrián Gómez Brandón, Dionysios Kefallinos, Hervé Kervin, Matan Kraus, Alberto Kummer, Nelson Maculan, Spyridon Maniatis, Anna Mpanti, Arnur Nigmetov, Thiago Noronha, André Nusser, Noujan Pashanasangi, Maria Predari, Mirko Rossi, Stefano Scafiti.

# Engineering Nearly Linear-Time Algorithms for Small Vertex Connectivity

Max Franck
Department of Computer Science, Aalto University, Espoo, Finland

Sorrachai Yingchareonthawornchai
Department of Computer Science, Aalto University, Espoo, Finland

## Abstract

Vertex connectivity is a well-studied concept in graph theory with numerous applications. A graph is $k$-connected if it remains connected after removing any $k-1$ vertices. The vertex connectivity of a graph is the maximum $k$ such that the graph is $k$-connected. There is a long history of algorithmic development for efficiently computing vertex connectivity. Recently, two near linear-time algorithms for small $k$ were introduced by [Forster et al. SODA 2020]. Prior to that, the best known algorithm was one by [Henzinger et al. FOCS'96] with quadratic running time when $k$ is small.

In this paper, we study the practical performance of the algorithms by Forster et al. In addition, we introduce a new heuristic on a key subroutine called local cut detection, which we call degree counting. We prove that the new heuristic improves space-efficiency (which can be good for caching purposes) and allows the subroutine to terminate earlier. According to experimental results on random graphs with planted vertex cuts, random hyperbolic graphs, and real world graphs with vertex connectivity between 4 and 15, the degree counting heuristic offers a factor of 2-4 speedup over the original non-degree counting version for most of our data. It also outperforms the previous state-of-the-art algorithm by Henzinger et al. even on relatively small graphs.

**2012 ACM Subject Classification** Theory of computation → Graph algorithms analysis

**Keywords and phrases** Algorithm Engineering, Algorithmic Graph Theory, Sublinear Algorithms

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.1

**Related Version** *Full Version*: `https://arxiv.org/abs/2103.15703`

**Supplementary Material** *Software (Source Code)*: `https://github.com/untellect/local-vertex-connectivity`
archived at `swh:1:dir:5afaa6ba812ccf4708141e05db5cd5bee88281fb`

**Funding** This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 759557.

## 1 Introduction

Given an undirected graph, the *vertex connectivity problem* is to compute the minimum size of a vertex set $S$ such that after removing $S$, the remaining graph is disconnected or a singleton. Such a vertex-set is called a *minimum vertex cut*. Vertex connectivity is well-studied concept in graph theory with applications in many fields. For example, for network reliability [13, 20], a minimum vertex-cut has the highest chance to disconnect the network assuming each node fails independently with the same probability; in sociology, vertex connectivity of a social network measures social cohesion [29].

There is a long history of algorithmic development for efficiently computing vertex connectivity (see [23] for more elaborated discussion of algorithmic development). Let $n$ and $m$ be the number of vertices and edges respectively in the input graph. The time complexity for computing vertex connectivity has been $O(n^2)$ since 1970 [17] even for the special case where the connectivity is a constant until very recently, when [9] introduced randomized (Monte Carlo)[1] algorithms to compute vertex connectivity in time $O(m + n\kappa^3 \log^2 n)$ (for undirected graphs) where $\kappa$ is the vertex connectivity of the graph. The algorithm follows the framework by [23]. This makes progress toward the conjecture (when $\kappa$ is a constant) by Aho, Hopcroft and Ullman [1] (Problem 5.30) that there exists a linear time algorithm for computing vertex connectivity. Before that, the state-of-the-art algorithm was due to [15], which runs in time $O(n^2\kappa \log n)$.

In this paper, we study the practical performance of the near-linear time algorithms by [9] for small vertex connectivity. We briefly describe their framework and point out the potential improvement of the framework. [23] provide a fast reduction from vertex connectivity to a subroutine called *local vertex-cut detection*. Roughly speaking, the framework deals with two extreme cases: detecting balanced cuts and unbalanced cuts. The balanced cuts can be detected using (multiple calls to) a standard *st*-max flow algorithm; the unbalanced cuts can be detected using (multiple calls to) local vertex-cut detection. Reference [9] follow the same framework and observe that local vertex-cut detection can be further reduced to another subroutine called *local edge-cut detection* as well as provide fast edge cut detection algorithms that finally prove the near-linear time vertex connectivity algorithm for any constant $\kappa$. The full algorithm is discussed in Appendix B. From our internal testing, we observe that, overall the framework, the performance bottleneck is on the local edge detection algorithm.

Therefore, our focus is on speeding up the local edge-cut detection algorithm. To define the problem precisely, we first set up notations. Let $G = (V, E)$ be a directed graph. Let $E(S, T)$ be the set of edges from vertex-set $S$ to vertex-set $T$. For any vertex-set $S$, let $\text{vol}^{\text{out}}(S) := \sum_{v \in S} \deg^{\text{out}}(v)$ denote the volume of $S$ which is total number of edges originating in $S$. Undirected edges are treated as one directed edge in each direction. We now define the interface of the local edge-cut detection algorithm.

▶ **Definition 1.** *An algorithm $\mathcal{A}$ is LocalEC if it takes as input a vertex $x$ of a graph $G = (V, E)$, and two parameters $\nu, k$ such that $\nu k = O(|E|)$, and output in the following manner:*
- *either output a vertex-set $S$ such that $x \in S$ and $|E(S, V \setminus S)| < k$ or,*
- *the symbol $\perp$ certifying that there is no non-empty vertex-set $S$ such that*

$$x \in S, \text{vol}^{\text{out}}(S) \le \nu, \text{ and } |E(S, V \setminus S)| < k. \tag{1}$$

*The algorithm is allowed to have bounded one-sided error in the following sense. If there is a non-empty vertex-set $S$ satisfying Equation* (1) *then $\perp$ is returned with probability at most* 1/2.

Reference [9] introduced two LocalEC algorithms with the running time $O(\nu k^2)$. The algorithms are very simple: they use repeated DFS (depth-first search) with different conditions for early termination. We note that this running time is enough to get a near-linear time algorithm for small connectivity using the framework by [23].

**Our Results and Contribution.** We introduce a heuristic called *degree counting* that is applicable to both variants of LocalEC in [9], which we call Local1+ and Local2+. We prove that the degree counting heuristic version is more space-efficient in terms of *edge-query* complexity and *vertex-query* complexity. Edge-query complexity is defined as the number of

[1] With at most $\frac{1}{n^c}$ error rate for any constant $c$.

edges that the algorithm accesses, and vertex-query complexity is defined as the number of vertices that the algorithm accesses. The results are shown in Table 1. These complexity measures can be relevant in practice. For example, an algorithm with low query complexity may be able to store the accessed data in a smaller cache than an algorithm with high query complexity.

**Table 1** Comparisons among various implementation of LocalEC algorithms. Local1+ denotes Local1 with the degree counting heuristic. Similarly, Local2+ denotes Local2 with the degree counting heuristic.

| LocalEC Variants | Time | Edge-query | Vertex-query | Reference |
|---|---|---|---|---|
| Local1 | $O(\nu k^2)$ | $O(\nu k^2)$ | $O(\nu k^2)$ | [9] |
| Local1+ | $O(\nu k^2)$ | $O(\nu k^2)$ | $O(\nu k)$ | This paper |
| Local2 | $O(\nu k^2)$ | $O(\nu k)$ | $O(\nu k)$ | [9] |
| Local2+ | $O(\nu k^2)$ | $O(\nu k)$ | $O(\nu)$ | This paper |

We conducted experiments on three types of undirected graphs: (1) graphs with planted cuts where we have control over size and volume of the cuts, and (2) random hyperbolic graphs, and (3) real-world networks. We denote LOCAL1, LOCAL1+, and LOCAL2+ to be the same local-search based vertex connectivity algorithm [9] (see Appendix B for details) except that the unbalanced part is implemented with different LocalEC algorithms using Local1, Local1+, Local2+, respectively. We use Local1 as a baseline for LocalEC algorithms. We denote HRG to be the preflow-push-relabel-based algorithm by [15]. We implement HRG as a baseline because when $k$ is small (say $k = O(1)$) HRG is the fastest known alternative to [9, 23]. The implementation detail can be found in Appendix C. By sparsification algorithm [22], we can assume that the input graph size depends on $n$ and $k$. The following summarize the key finding of our empirical studies.

1. **Internal Comparisons (Section 5.5).** We compare three LocalEC algorithms (Local1, Local1+, Local2+). According to the experiments (Figure 4), for any $\nu$ parameter, Local1+ and Local2+ visit *significantly* fewer edges than Local1. Also, Local2+ visits slightly fewer edges than Local1+ overall. The degree counting is also very effective at low volume parameter. When plugging into full vertex connectivity algorithms, the degree counting heuristics (LOCAL1+ and LOCAL2+) improve the performance over non-degree counting counter part (LOCAL1) by a factor 2 to 4 for most data used in our experiments, although for some larger graphs the speedup was noticeably larger. The greatest observed speedup over LOCAL1 is 18.4x for LOCAL2+ at $n = 100000$, $\kappa_G = 16$. For graphs of this size, LOCAL2+ performs slightly better than LOCAL1+. Finally, according to CPU sampling, the local search is the main bottleneck for the performance of LOCAL1 at roughly at least 90% for large instances. On the other hand, for the degree counting versions (LOCAL1+ and LOCAL2+), the CPU usage of local search part is improved to be almost the same as the other main component (i.e., finding a balanced cut using the Ford-Fulkerson's max-flow algorithm).
2. **Comparisons to HRG.** We compare four vertex connectivity algorithms, namely HRG, LOCAL1, LOCAL1+, LOCAL2+. For planted cuts (Section 5.2), LOCAL1, LOCAL1+, and LOCAL2+ scale with $n$ much better than HRG when $\kappa_G$ is fixed. In particular, LOCAL1+ and LOCAL2+ start to outperform HRG on graphs as small as $n \leq 500$ (when $\kappa \leq 15$). For random hyperbolic graphs (Section 5.3), HRG performs much better than on the planted cut instances, but is still outperformed relatively early. In particular, LOCAL1+ and LOCAL2+ outperform HRG for $n \geq 5000$ when $\kappa \leq 12$. In

real-world graphs (Section 5.4), LOCAL1+ and LOCAL2+ are the fastest among the four algorithms with LOCAL2+ being slightly faster than LOCAL1+. We also observe that the performance of all four algorithms is very similar on part of the real world dataset and graphs with planted cuts with the same size and vertex connectivity.

**Organization.** We discuss related work in Section 2, and preliminaries in Section 3. Then, we review two variants of LocalEC algorithms (Local1,Local2) [9], and describe new degree counting heuristic versions (Local1+, Local2+) in Section 4. Then, all the experimental results are discussed in Section 5. We conclude and discuss future work in Section 6.

## 2 Related Work

**Fast Vertex Connectivity Algorithms.** We consider a decision version where the problem is to decide if $G$ has a vertex cut of size at most $k-1$ (the general vertex connectivity can be solved using a binary search on $k$). We highlight only recent state-of-the-art algorithms. For more elaborated discussion, see [23]. When $k = O(1)$, the fastest known algorithm is by [9] with running time $O(m + nk^3 \log^2 n)$. The algorithm is based on local search approach. For larger $k$, the fastest known algorithm are based on preflow-push-relabel by [15] with the running time $O(n^2 k \log n)$, and based on algebraic techniques by [19] with the running time $O(n^\omega \log^2 n + k^\omega n \log n)$ where $\omega$ denotes the matrix multiplication exponent, currently $\omega \leq 2.37286$ [2]. When $k$ is small (say $k = O(1)$), the preflow-push-relabel-based algorithm by [15] is the fastest alternative to [9, 23]. Therefore, we implement the preflow-push-relabel-based algorithm [15] as a baseline for performance comparisons. We note both all aforementioned algorithms are randomized. Deterministic algorithms are much slower than the randomized ones. The fastest known deterministic algorithms are by [10] for large $k$ and by [11] for $k = O(1)$.

**Deciding $(k, s, t)$-Vertex Connectivity.** We mention another related problem which is to decide if the there is a vertex cut separating $s$ and $t$ of size at most $k-1$. By a standard reduction [7], it can be solved by $st$-maximum flow. $st$-maximum flow can be solved in time $O(mk)$ by augmenting paths algorithm by Ford-Fulkerson algorithm [8]. For larger $k$, a simple blocking flow algorithm by [6] runs in time $O(m\sqrt{n})$. The current state-of-the art algorithms are $O(m^{4/3+o(1)})$-time algorithm by [21], and $\tilde{O}(m + n^{1.5})$-time[2] algorithm by [27]. Note that when $k$ is small (e.g., $k = O(1)$), then Ford-Fulkerson algorithm [8] is the fastest, and we thus implement Ford-Fulkerson algorithm as a subroutine to find vertex cut for the balanced case.

**Local Search.** There are quite a few local search algorithm with different running time. The first LocalEC algorithm by [4] has running time of $O(\nu k^k)$. [9] introduced a new local search algorithm with improved time $O(\nu k^2)$. [9] also provide a reduction to local vertex cut detection problem, which we called LocalVC (similar to Definition 1, but uses vertex cut instead of edge cut). Therefore, there is a LocalVC algorithm with running time $O(\nu k^2)$. This improved the previous bound for LocalVC with running time $O(\nu^{1.5} k)$ by [23] when $k$ is small. For our purpose, when $k$ is small (say $k = O(1)$), the algorithm by [9] is the fastest, and thus we consider the LocalEC algorithm by [9].

[2] $\tilde{O}(f(n)) = O(\text{poly}(\log n) f(n))$.

**Implementation and Experimental Studies.** To the best of our knowledge, this paper is the first experimental study on vertex connectivity algorithms; there were no prior experimental studies on vertex connectivity algorithms[3]. This is in stark contrast to the edge-connectivity problem (which is considered as a sibling problem) where we compute the minimum number of edges to be removed to disconnect the graph. For edge-connectivity, there are many experimental studies [16, 5, 24, 14]. More recently, the work by [12] implemented the local search framework in [9] to compute directed edge-connectivity.

## 3 Preliminaries

Let $G = (V, E)$ be an undirected graph. In general, we denote $m = |E|$ and $n = |V|$. We denote $E(S, T)$ be the set of edges from vertex-set $S$ to vertex-set $T$. We say that $S \subset V$ is a *vertex cut* if $G - S$ (the graph after removing $S$ from $G$) is disconnected. If no vertex cut of size $k$ exists, the graph is k-(vertex)-connected. We say that $S$ is an $xy$-vertex cut if $x$ cannot reach $y$ in $G - S$. Let $\kappa_G$ be vertex connectivity of $G$, i.e., the size of the minimum vertex-cut (or $n - 1$ if no cut exists). Let $\kappa_G(x, y)$ denote the size of the minimum $xy$-vertex cut in $G$ or $n - 1$ if the $xy$-vertex cut does not exist. We say that a triplet $(L, S, R)$ is a *separation triple* if $L, S$ and $R$ form a partition of $V$, $L$ and $R$ are not $\emptyset$ and $E(L, R) = \emptyset$. In this case, $S$ is a vertex-cut in $G$. The decision problem for vertex connectivity which we call $k$-connectivity problem is the following: Given $G = (V, E)$, and integer $k$, decide if $G$ is $k$-connected, and if not, output a vertex-cut of size $< k$.

**Sparsification.** For an undirected graph $G = (V, E)$, the algorithm by Nagamochi and Ibaraki [22] runs in $O(m)$ time and partitions $E$ into a sequence of forests $E_1, \ldots, E_n$ (possibly $E_i = E_{i+1} = \ldots = E_n = \emptyset$ for some $i$). For each $k \leq n$, the subgraph $FG_k := (V, \bigcup_{i \leq k} E_i)$ has the property that $FG_k$ is $k$-connected if and only if $G$ is $k$-connected. Moreover, any vertex cut of size $< k$ in $FG_k$ is also a vertex cut in $G$. Clearly, $|E(FG_k)| \leq nk$.

From now, with preprocessing in $O(m)$ time, we assume that the input graph to the $k$-connectivity problem is $FG_k$. In particular, we can assume that the number of edges is $O(nk)$. We can also assume that the minimum degree is at least $k$ (because otherwise we can output the neighbor of the vertex with minimum degree).

**Split Graph.** The split graph construct is a standard reduction from vertex connectivity based problems to edge connectivity based problems, used in the algorithms featured in this paper, among others [7, 9, 15]. Given graph $G$, we define the split graph $SG$ as follows. For each vertex $v$ in $G$, we replace $v$ with an "in-vertex" $v_{\text{in}}$ and an "out-vertex" $v_{\text{out}}$, and add an edge from $v_{\text{in}}$ and $v_{\text{out}}$. The reduction follows from the observation that edge-disjoint paths in $SG$ that start at an outvertex and end at an invertex correspond to (non-endpoint) vertex-disjoint paths in $G$. For each edge $(u, v)$ in $G$, we add an edge from $(v_{\text{in}}, u_{\text{out}})$ in $SG$.

## 4 LocalEC Algorithms and Degree Counting Heuristics

In this section, we review two variants of LocalEC algorithms by [9], and describe their corresponding new version using the degree counting heuristic. For completeness, we describe the complete vertex connectivity algorithm by [9] and some implementation details in Appendix B.

[3] The experimental work by [25] mentioned $k$-vertex connectivity problem. However, in the experiment, they studied only the algorithm for deciding $(k, s, t)$-vertex connectivity where the source $s$ and sink $t$ are given as inputs.

All the algorithms in this section follow a common framework called AbstractLocalEC as described in Algorithm 1. Let $G = (V, E)$ be the graph that we work on. The algorithm takes as inputs $x \in V$ and two integers $\nu, k$. The basic idea is to apply Depth-first Search (DFS) on the starting vertex $x$ but force early termination. We repeat for $k$ iterations. If DFS terminates normally at some iteration, i.e., without having to apply the early termination condition, then the set of reachable vertices satisfy Equation (1). Otherwise, we certify that no cut satisfying Equation (1) exists. The only main difference is at line 2 where we need to specify the condition for early termination and selection of the vertex $y \in V(T)$ in such a way that the entire algorithm outputs correctly with constant probability. If the minimum degree is less than $k$, we set $k$ to the minimum degree and return the trivial cut if no smaller cut is found.

**Algorithm 1** AbstractLocalEC$_G(x, \nu, k)$.

```
1 repeat $k$ times
2     Grow a DFS tree $T$ starting from $x$, stopping early at some point to get $y \in V(T)$.
3     If the DFS terminates normally, then return $V(T)$.
4     Reverse all edges along the unique path from $x$ to $y$ in the tree $T$, unless this is
        the last iteration.
5 return $\perp$.
```

Next, we define time and space complexity (in terms of edges and vertices required to run the algorithm) of a LocalEC algorithm.

▶ **Definition 2.** *Let $\mathcal{A}(x, \nu, k)$ be a LocalEC algorithm. $\mathcal{A}$ has $(t, s_e, s_v)$-complexity if $\mathcal{A}$ terminates in $O(t)$ time and accesses at most $O(s_e)$ distinct edges, and at most $O(s_v)$ distinct vertices.*

## 4.1 Local1 and Degree Counting Version

**Algorithm for Local1.** Replace line 2 in Algorithm 1 with the following process. Grow a DFS tree starting on vertex $x$ and stop when the number of accessed edges is exactly $8\nu k$. Let $E'$ be the set of accessed edges. We sample an edge $(u, v) \in E'$ uniformly at random. Finally, we set $y \leftarrow u$. If we sample $(u, v)$ to be the $\tau$-th edge visited, we can stop the DFS early after that edge (similarly to Local1+ below).

▶ **Theorem 3** (Theorem A.1 in [9]). *Local1$(x, \nu, k)$ is LocalEC with $(\nu k^2, \nu k^2, \nu k^2)$-complexity.*

Next, we present the degree counting version of Local1, which we call Local1+.

**Algorithm for Local1+.** Replace line 2 in Algorithm 1 with the following process. Let $\tau$ be a random integer in the range $[1, 8\nu k]$. If this is in the last iteration, we set $\tau \leftarrow 8\nu k$. Then, we grow a DFS tree $T$ starting on vertex $x$. At any time step, let $V(T)$ be the set of vertices visited by the DFS so far. We stop as soon as $\mathrm{vol}^{\mathrm{out}}(V(T)) \geq \tau$. Finally, we set $y$ to be the last vertex that the DFS visited.

▶ **Theorem 4.** *Local1+$(x, \nu, k)$ is LocalEC with $(\nu k^2, \nu k^2, \nu k)$-complexity.*

## 4.2 Local2 and Degree Counting Version

We say that an edge is *new* if it has not been accessed in earlier iterations. Otherwise, it is *old*. It follows that reversed edges are old.

**Algorithm for Local2.** [4] Replace line 2 in Algorithm 1 with the following process. We grow a DFS tree $T$ starting at vertex $x$. Let $E'(T)$ be the set of new edges visited. We stop as soon as $E'(T) \geq 8\nu$. Let $(u, v)$ be a random edge in $E'(T)$. Finally, we set $y \leftarrow u$. We do not need to store $E'(T)$ to sample from if we sample $\tau$ in the range $[1, 8\nu]$ and choose the $\tau$-th new edge.

▶ **Theorem 5** (Equivalent to Theorem 3.1 in [9]). *Local2$(x, \nu, k)$ is LocalEC with $(\nu k^2, \nu k, \nu k)$-complexity.*

Next, we present the degree counting version of Local2, which we call Local2+. The algorithm is slightly more complicated. We set up notations. For each $v \in V$, let $c(v)$ be the remaining capacity for $v$, representing uncounted edge volume. Initially, $c(v) = \deg^{\text{out}}(v)$.

**Algorithm for Local2+.** Replace line 2 in Algorithm 1 with the following process. Let $\tau$ be a random integer in the range $[1, 8\nu k]$. We grow a DFS tree starting on vertex $x$. At any time step, let $v_1, v_2, ..., v_i$ be the sequence of vertices visited by the DFS so far. For the first vertex where $\sum_{j \leq i} c(v_j) \geq \tau$, we set $y \leftarrow v_i$. As soon as $\sum_{j \leq i} c(v_j) \geq 8\nu$, we stop the DFS and update the remaining capacity $c(v)$ on each $v$ as follows. We set $c(v_j) \leftarrow 0$ for all $j < i$ and set $c(v_i) \leftarrow \sum_{j \leq i} c(v_j) - 8\nu$.

Intuitively, we collect previously uncounted outgoing edges and choose the origin vertex for one of them at random.

▶ **Theorem 6.** *Local2+$(x, \nu, k)$ is LocalEC with $(\nu k^2, \nu k, \nu)$-complexity.*

## 4.3 Proof of Theorems 3–6

In this section, we address proofs for Theorems 3–6.

**Correctness.** It can be shown that all four algorithms (Local1, Local1+, Local2, Local2+) are LocalEC through a similar argument as used in [9]. For completeness, we provide the proofs in Appendix A.

**Complexity.** Let $\mathcal{A}$ be an LocalEC algorithm (Definition 1), and let $\nu$, and $k$ be the parameters of the algorithm. We define three measure of complexity $T(\mathcal{A}, G), U_E(\mathcal{A}, G)$, and $U_V(\mathcal{A}, G)$ on input graph $G$ and LocalEC algorithm $\mathcal{A}$ as follows. Let $T(\mathcal{A}, G)$ be the number of times that the algorithm accesses edges on the input graph $G$. $T(\mathcal{A}, G)$ measures time complexity of the algorithm. Let $U_E(\mathcal{A}, G)$ be the number of unique edges accessed by the algorithm on graph $G$. This measures how much information (in terms of number of edges) that the algorithm needs to run. Let $U_V(\mathcal{A}, G)$ be the number of unique vertices accessed by the algorithm on graph $G$.

▶ **Observation 7.** *For any graph $G$ and LocalEC algorithm $\mathcal{A}$, $T(\mathcal{A}, G) \geq U_E(\mathcal{A}, G) \geq U_V(\mathcal{A}, G)$.*

[4] The algorithm Local2 described in this paper is similar to Algorithm 1 in [9]. Our description here is simpler, and achieves the same properties as Algorithm 1 in [9].

**Local1.** To see that Local1 has $(O(\nu k^2), O(\nu k^2), O(\nu k^2))$-complexity, it is enough to prove that $T(\text{Local1}, G) = O(\nu k^2)$. This follows easily because each iteration we stop the DFS after visiting exactly $8\nu k$ edges, and there are at most $k$ iterations.

**Local1+.** We first prove that $T(\text{Local1+}, G) = O(\nu k^2)$. Since there are $k$ iterations, it is enough to bound one iteration. Let $S$ be the set of vertices visited by the DFS before the step at which it stops early. Clearly, $\text{vol}^{\text{out}}(S) < 8\nu k$, or we would have stopped earlier. By design, new edges can be only visited within the set $E(S, S)$ or at the last step. Therefore, the number of edges visited is at most $|E(S, S)| + 1 \leq \text{vol}^{\text{out}}(S) + 1 = O(\nu k)$ per iteration and $O(\nu k^2)$ in total. We have $T(\text{Local1+}, G) = O(\nu k^2)$.

Remember that if the minimum degree is initially at least $k$ to avoid trivial cuts. When paths are reversed, no vertex other than $x$ will have reduced degree. Therefore we have $k(|S| - 1) \leq \text{vol}^{\text{out}}(S) < 8\nu k$. It follows that we visit at most $O(\nu)$ vertices in each iteration and $O(\nu k)$ in total.

**Local2.** We first prove that $U_E(\text{Local2}, G) = O(\nu k)$. By design, for each iteration, we collect at most $8\nu$ new edges. Since we repeat for $k$ iterations, we collect at most $8\nu k$ total new edges. Next, we prove $T(\text{Local2}, G) = O(\nu k^2)$. Since each edge can be revisited at most $k$ times, we have $T(\text{Local2}, G) \leq kU_E(\text{Local2}, G) = O(\nu k^2)$.

**Local2+.** We first prove that $U_E(\text{Local2+}, G) = O(\nu k)$. If true, then we also have $T_E(\text{Local2+}, G) \leq kU_E(\text{Local2+}, G) = O(\nu k^2)$. We will never visit an outgoing edge of vertex $v$ unless all its capacity has been exhausted. Therefore the total used capacity (at most $k$ times $8\nu$) is an upper bound for the number of distinct edges visited. For $U_V(\text{Local2+}, G)$, fix any iteration. Let $S$ be the set of vertices visited by the DFS one step before terminating and $S' \subseteq S$ the subset of $S$ that have not been visited before. Clearly, we have $k|S'| \leq \text{vol}^{\text{out}}(S') = \sum_{v \in S'} c(v) \leq \sum_{v \in S} c(v) < 8\nu$. The first inequality follows since the minimum degree is at least $k$. We visit at most $|S'| + 1 = O(\nu/k)$ distinct vertices per iteration for a total of $O(\nu)$ distinct vertices.

## 5 Experimental Results

### 5.1 Experimental Setup

The algorithms were implemented and compiled using C++17 with Microsoft Visual Studio 2019. All experiments were run on a Windows 10 computer with Intel i7-9750H CPU (2.60GHz) and 16 GB DDR4-2667 RAM.

Four algorithms are compared. **LOCAL1**, **LOCAL1+** and **LOCAL2+** are implementations based on the algorithm by Forster et al [9]. The full algorithm to compute vertex connectivity using LocalEC is described in Appendix B and originally by [23]. LOCAL1 and LOCAL1+ use Local1 and Local1+ as their LocalEC algorithm with $2\nu k$ substituted for $8\nu k$. LOCAL2+ uses the LocalEC algorithm Local2+ with $3\nu$ substituted for $8\nu$. **HRG** is an implementation of the randomised version of the algorithm by Henzinger, Rao and Gabow [15]. The implementation details are described in Appendix C. All algorithms were implemented using parameters that bound theoretical success probability from below by a roughly equal constant. Since the data consists of undirected graphs only, the sparsification algorithm by Nagamochi and Ibaraki [22] is used together with each algorithm. The $O(m)$ partitioning of the edges into disjoint forests is not included in the measured time. Construction of the sparse graphs in $O(nk)$ time is included. As a result, none of the algorithms have time complexity dependent on m. Graph size is reported only in terms of vertices.

### 5.1.1 Data

The data consists of random graphs with planted vertex cuts, random hyperbolic graphs and real world data.

The first artificial dataset consists of graphs with a planted unique minimum vertex cut, which can be generated with full control over vertex connectivity and balancedness. We partition a complete graph into three sets $L$, $S$ and $R$ and use a subset of the edges in $E \setminus E(L, R)$, chosen using a modified version of the sparsification algorithm by Nagamochi and Ibaraki [22]. Like Nagamochi and Ibaraki, we label the edges to partition them into disjoint forests $\{E_1, E_2, ...\}$ such that $(x, y) \in E_i$ implies that there is a path between $x$ and $y$ in $E_1, E_2, ..., E_{i-1}$. Nagamochi and Ibaraki show that if this property holds for all edges, then the union of the $k$ first forests is $k$-connected if the original graph is $k$-connected. Unlike Nagamochi and Ibaraki, we randomly partition the edges by placing them in the applicable forest with the lowest index in a random order. We choose $k = 60 > |S|$ to guarantee that $S$ is a unique vertex cut that separates $L$ from $R$. For each set of parameters we generate five graphs and run the algorithm five times each and report the average.

The second artificial dataset consists of random hyperbolic graphs, generated using NetworKIT [26], which provides an implementation of the generator by von Looz et al. [28]. The properties of random hyperbolic graphs include a degree distribution that follows a power law and small diameter, which are common in real world graphs [3]. The graphs are generated with average degree 32 and a power law exponent of 10. We generate 20 graphs each for sizes $2^{10}, 2^{11}, ..., 2^{18}$ vertices and group them according to vertex connectivity. We run the algorithm five times per graph and report the average for each group with the same size and vertex connectivity.

The real world data is based on three graphs from the SNAP dataset [18], soc-Epinions1, com-LiveJournal and web-BerkStan. The LiveJournal dataset is originally undirected. The other two are directed graphs read as undirected, which means that we compute weak vertex connecitivity for these graphs. We preprocess these graphs by taking the largest connected component for a $k$-core. A $k$-core is defined as the edge-maximal subgraph with minimum degree at least $k$. Only $k$-cores whose vertex connectivity is over 1 but less than the minimum degree are used. For each $k$-core we run the algorithms 25 times and report the average.

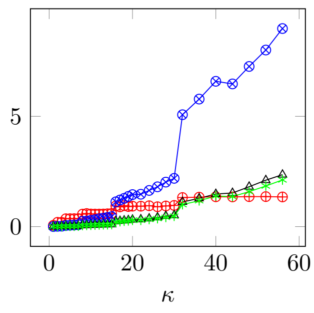
## 5.2 Planted Cuts

In theory the running time for HRG is linear in $\kappa$ and the algorithms based on Forster et al. [9] are cubic in $\kappa$. Figure 1a shows that the running time for HRG indeed grows much slower with $\kappa$. The running time for LOCAL1 exceeds that of HRG much earlier, at $\kappa \geq 17$, than LOCAL1+ ($\kappa \geq 40$) and LOCAL2+ ($\kappa \geq 48$).

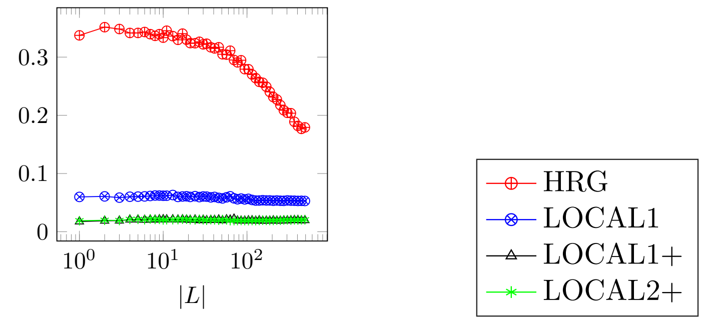
Figure 1b shows that all four algorithms perform reasonably well both for graphs with unbalanced cuts and balanced cuts, although HRG is faster for unbalanced graphs by a factor of 2. Internal testing suggests that the running time of HRG is roughly proportional to $|L|^2 + |R|^2$. The difference between the highest and lowest running time is a factor of 1.99 for HRG, 1.19 for LOCAL1, 1.27 for LOCAL1+ and 1.16 for LOCAL2+.

When $\kappa < 16$, LOCAL1, LOCAL1+ and LOCAL2+ outperform the quadratic-time HRG on very small graphs with planted cuts. At $\kappa = 4$ in figure 2a, HRG takes 23 ms for 100 vertices, which is already slower than both LOCAL1+ and LOCAL2+. LOCAL1 is faster than HRG at $n \geq 200$. When $\kappa = 15$ (figure 2d), HRG is slower than LOCAL1+ and LOCAL2+ at $n \geq 250$ and LOCAL1 at $n \geq 550$.

LOCAL1+ and LOCAL2+ perform very similarly for small graphs but on larger graphs, LOCAL2+ is faster, as shown by figure 2e.

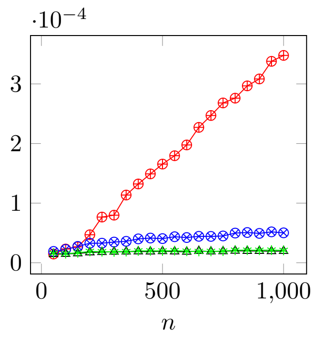


**(a)** $|L| = 5$, $n = 1000$

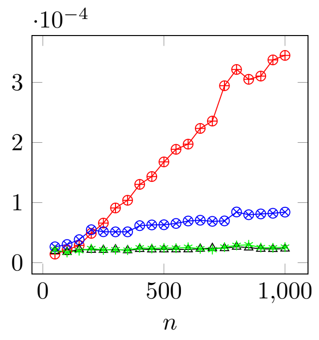


**(b)** $\kappa = 5$, $n = 1000$

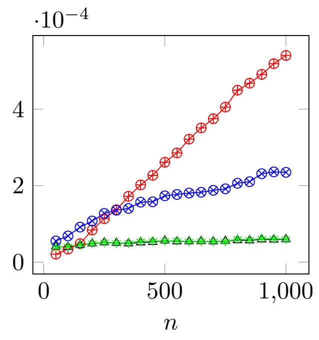
**Figure 1** Running time (seconds) for Planted Cuts with variable $|L|$ or $\kappa$.



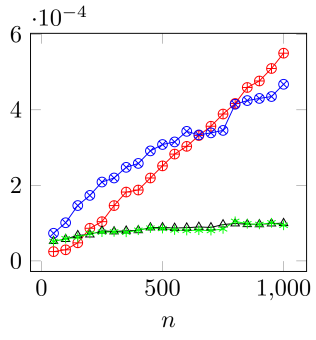
**(a)** $\kappa = 4$, $|L| = 5$



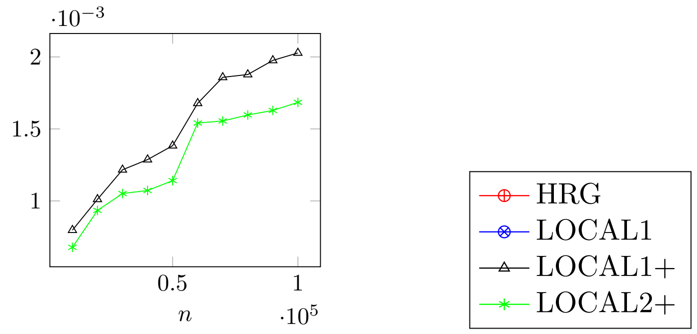
**(b)** $\kappa = 7$, $|L| = 5$



**(c)** $\kappa = 8$, $|L| = 5$



**(d)** $\kappa = 15$, $|L| = 5$



**(e)** $\kappa = 31$, $|L| = 5$

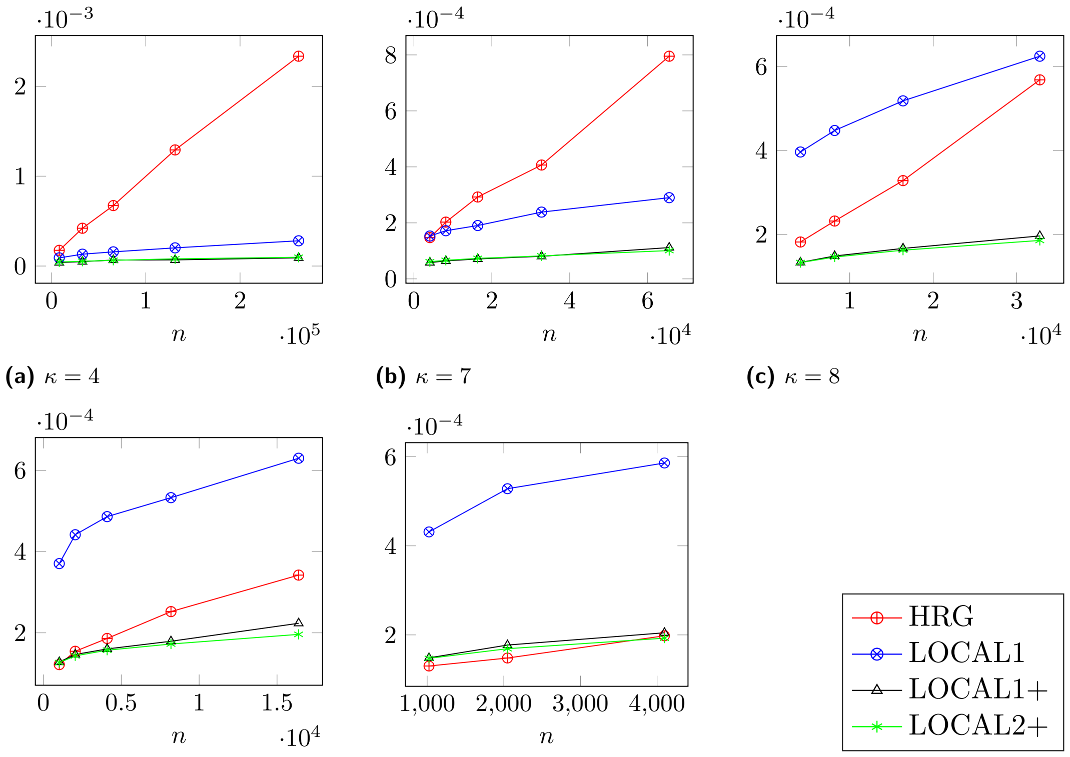
**Figure 2** Running time (seconds) per vertex for Planted cuts with fixed $|L|$ and $|S|$.

## 5.3 Random Hyperbolic Graphs

HRG is much faster on random hyperbolic graphs than on the planted cut dataset. Comparing figures 2c and 3c, the performance of HRG on 1000 vertex graphs with planted cuts of size 8 is similar to that on random hyperbolic graphs with the same vertex connectivity and over 30000 vertices. The performance differences are smaller for LOCAL1, LOCAL1+ and

LOCAL2+, which means that the point at which these algorithms outperform HRG occurs at somewhat higher $n$.

For random hyperbolic graphs with $\kappa = 7$ (figure 3b), HRG and LOCAL1 are equally fast at 4096 vertices (0.6 seconds). HRG is faster than LOCAL1 for all included random hyperbolic graphs where $\kappa > 7$, including graphs up to 32768 vertices. The running time for LOCAL1+ and LOCAL2+ is close to that of HRG for random hyperbolic graphs where $\kappa = 12$ and $n \in [1024, 4196]$ (figure 3e).



**(a)** $\kappa = 4$ **(b)** $\kappa = 7$ **(c)** $\kappa = 8$

**(d)** $\kappa = 10$ **(e)** $\kappa = 12$

**Figure 3** Running time (seconds) per vertex for Random Hyperbolic Graphs.

## 5.4 Real-World Networks

Table 2 presents real world network data. Each row represents a $\delta$-core, where $\delta$ is the minimum degree of the resulting graph. Note that in general, minimum degree for a $k$-core can exceed $k$. Table 3 shows data for graphs with planted cuts with similar parameters to the real world graphs, for comparison.

LOCAL1+ and LOCAL2+ clearly outperform LOCAL1 on real-world networks, as on artificial data. The $k$-cores of soc-Epinions1 have very similar performance in real world networks and graphs with planted cuts in table 3. Performance for other real network data is generally faster for all four algorithms than for planted cuts, especially for HRG, which is 5-8 times faster on real world data. Similarly, running times for LOCAL1, LOCAL1+ and LOCAL2+ are also higher on random hyperbolic graphs than on $k$-cores of com-lj.ungraph and web-BerkStan.

**Table 2** Running times (milliseconds) per vertex on $k$-cores for real world networks.

| n | $\kappa$ | $\delta$ | LOCAL1 | LOCAL1+ | LOCAL2+ | HRG | graph |
|---|---|---|---|---|---|---|---|
| 1493 | 10 | 160 | 0.192706 | 0.059531 | 0.055352 | 0.117033 | com-lj.ungraph |
| 1205 | 9 | 200 | 0.174805 | 0.056589 | 0.053237 | 0.097187 | com-lj.ungraph |
| 853 | 8 | 231 | 0.146436 | 0.054068 | 0.052052 | 0.069566 | com-lj.ungraph |
| 781 | 8 | 250 | 0.130679 | 0.054225 | 0.052612 | 0.06589 | com-lj.ungraph |
| 10147 | 5 | 9 | 0.143089 | 0.027456 | 0.027409 | 5.207559 | soc-Epinions1 |
| 8106 | 11 | 12 | 0.896141 | 0.095449 | 0.084268 | 6.20417 | soc-Epinions1 |
| 6246 | 16 | 17 | 3.063016 | 0.299094 | 0.236018 | 6.537896 | soc-Epinions1 |
| 5719 | 17 | 19 | 3.104109 | 0.325849 | 0.238042 | 5.831964 | soc-Epinions1 |
| 5480 | 2 | 80 | 0.011066 | 0.006058 | 0.006547 | 0.359113 | web-BerkStan |
| 5352 | 12 | 93 | 0.214441 | 0.055435 | 0.053617 | 0.800785 | web-BerkStan |

**Table 3** Running times (milliseconds) per vertex on Planted Cuts ($|L| = 5$).

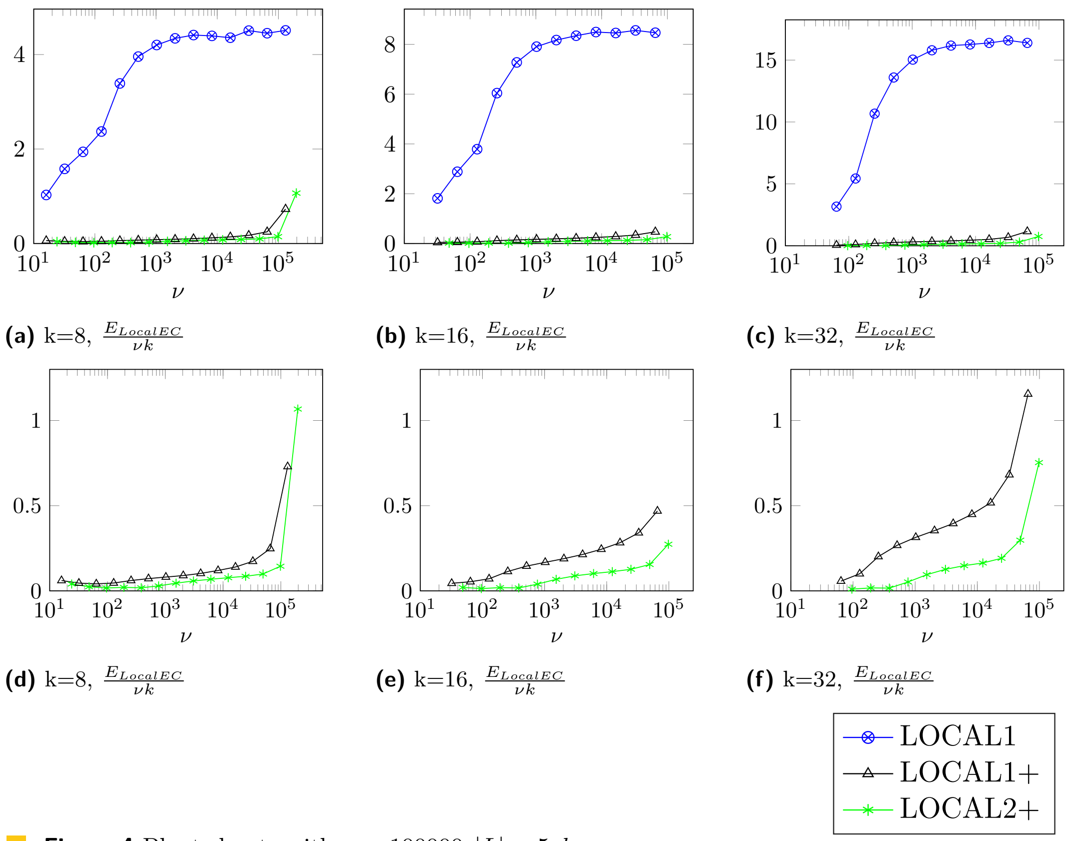
| n | $\kappa$ | LOCAL1 | LOCAL1+ | LOCAL2+ | HRG |
|---|---|---|---|---|---|
| 1493 | 10 | 0.34756 | 0.07509 | 0.07251 | 0.93666 |
| 1205 | 9 | 0.28973 | 0.06567 | 0.06596 | 0.71276 |
| 853 | 8 | 0.21861 | 0.05829 | 0.05881 | 0.46607 |
| 781 | 8 | 0.21249 | 0.05725 | 0.05921 | 0.44556 |
| 10147 | 5 | 0.14592 | 0.02637 | 0.02763 | 5.12975 |
| 8106 | 11 | 0.83576 | 0.11107 | 0.104 | 6.34581 |
| 6246 | 16 | 2.96568 | 0.33908 | 0.30892 | 6.61489 |
| 5719 | 17 | 2.99657 | 0.3346 | 0.29596 | 5.93766 |
| 5480 | 2 | 0.01858 | 0.00811 | 0.00853 | 1.69671 |
| 5352 | 12 | 0.77961 | 0.1108 | 0.10384 | 3.98331 |

## 5.5 Effectiveness of Degree Counting

In figure 4 we study internal measurements from LocalEC in the different algorithms. Note that Local1 and Local1+ apply a multiplicative factor of 2 to $\nu$ and Local2+ a factor of 3. The values used here include this increase. The number of edges visited by the average call to LocalEC at each value for $\nu$ is normalised by $\nu k$. This metric approximately doubles for Local1 and Local1+ when $k$ is doubled, as expected for algorithms quadratic in $k$. The metric grows for Local2+ too, but by a smaller factor around 1.5 for most values. The growth is faster for higher values for $\nu$ and for the highest values it is approximately by a factor 2, like the other two algorithms.

The number of edges explored relative to $\nu$ is higher for high $\nu$ for all algorithms and parameters in figure 4. For LOCAL1, it converges towards $\frac{\nu k}{2}$, which is the average of $[1, \nu k]$, the range of possible early stopping points $\tau$.

Local1 clearly visits more edges than in Local1+ and Local2+ by a large factor, according to figure 4. Table 4 shows that most of the running time of LOCAL1 is used searching for unbalanced cuts with LocalEC. However, LOCAL1+ and LOCAL2+ spend a similar amount of time on balanced and unbalanced cuts. The only difference between the versions is the choice of LocalEC. These results suggest that degree counting improves the practical performance of LocalEC significantly but there is not much more room for improvement through LocalEC without also further optimising x-y max flow to search for balanced cuts. When the number of vertices is increased by a factor of 10, the time spent searching for unbalanced cuts does not seem to grow faster than the time spent searching for balanced cuts. The category “other” is dominated by initial setup for the data structures.



**(a)** k=8, $\frac{E_{LocalEC}}{\nu k}$ **(b)** k=16, $\frac{E_{LocalEC}}{\nu k}$ **(c)** k=32, $\frac{E_{LocalEC}}{\nu k}$

**(d)** k=8, $\frac{E_{LocalEC}}{\nu k}$ **(e)** k=16, $\frac{E_{LocalEC}}{\nu k}$ **(f)** k=32, $\frac{E_{LocalEC}}{\nu k}$

**Figure 4** Planted cuts with $n = 100000, |L| = 5, k = \kappa$
(Non-unique) average edges per LocalEC call, normalised by $\nu k$.

**Table 4** CPU use: balanced cuts/Ford-Fulkerson(FF) vs unbalanced cuts/LocalEC(Local). Running time was measured separately.

**(a)** $n = 10000, \kappa = 8$

| algorithm | FF | Local | Other | Seconds |
|---|---|---|---|---|
| LOCAL | 6.2% | 92.5% | 1.3% | 5.96 |
| LOCAL+ | 45.7% | 44.2% | 10.2% | 0.79 |
| LOCAL2+ | 47.7% | 42.3% | 10% | 0.85 |

**(b)** $n = 10000, \kappa = 16$

| algorithm | FF | Local | Other | Seconds |
|---|---|---|---|---|
| LOCAL | 4.5% | 95.2% | 0.3% | 34.82 |
| LOCAL+ | 48.3% | 48.2% | 3.5% | 3.32 |
| LOCAL2+ | 42.7% | 53.5% | 3.7% | 3.07 |

**(c)** $n = 100000, \kappa = 8$

| algorithm | FF | Local | Other | Seconds |
|---|---|---|---|---|
| LOCAL1 | 4.3% | 95.2% | 0.5% | 205.2 |
| LOCAL1+ | 52.1% | 42.5% | 5.3% | 15.8 |
| LOCAL2+ | 51.9% | 43% | 5.1% | 14.1 |

**(d)** $n = 100000, \kappa = 16$

| algorithm | FF | Local | Other | Seconds |
|---|---|---|---|---|
| LOCAL1 | 2.6% | 97.3% | 0.1% | 1546.1 |
| LOCAL1+ | 49.5% | 48.9% | 1.6% | 96.8 |
| LOCAL2+ | 51.2% | 47.1% | 1.6% | 84.3 |

## 5.6 Success rate

We define the success rate of a vertex connectivity algorithm as the percentage of attempts that yields an optimal cut. The observed success rate for HRG is at or near 100% on all featured datasets. For random hyperbolic graphs, none of the algorithms returned nonoptimal cuts. For graphs with planted cuts and $k$-cores of real world networks, the success rates are 97%+ for LOCAL1, 96%+ for LOCAL1+ and 95%+ for LOCAL2+.

## 6 Conclusion and Future Work

We study the experimental performance of the near-linear time algorithm by [9] when the input graph connectivity is small. The algorithm is based on local search. We also introduce a new heuristic for the local search algorithm, which we call degree counting. Based on experimental results, the degree counting heuristic significantly improves the empirical running time of the algorithm over its non-degree counting counterpart. For future work, we plan to extend the experiments to directed graphs, and on larger instances of datasets (in the order of millions of edges).

## References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

2 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. *CoRR*, abs/2010.05846, 2020.

3 Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2, 2006.

4 Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *SODA*, pages 1900–1918. SIAM, 2017.

5 Chandra Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Clifford Stein. Experimental study of minimum cut algorithms. In *SODA*, pages 324–333. ACM/SIAM, 1997.

6 Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In *Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 218–240. Springer, 2006.

7 Shimon Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput.*, 4(3):393–396, 1975.

8 Lester Randolph Ford and Delbert Ray Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

9 Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In *SODA*, pages 2046–2065. SIAM, 2020.

10 Harold N. Gabow. Using expander graphs to find vertex connectivity. *J. ACM*, 53(5):800–844, 2006.

11 Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Deterministic graph cuts in subquadratic time: Sparse, balanced, and k-vertex. *CoRR*, abs/1910.07950, 2019.

12 Loukas Georgiadis, Dionysios Kefallinos, Luigi Laura, and Nikos Parotsidis. An experimental study of algorithms for computing the edge connectivity of a directed graph. *ALENEX*, pages 85–97, 2021.

13 Olivier Goldschmidt, Patrick Jaillet, and Richard Lasota. On reliability of graphs with node failures. *Networks*, 24(4):251–259, 1994.

14 Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM J. Exp. Algorithmics*, 23, 2018.

15 Monika Rauch Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. In *FOCS*, pages 462–471. IEEE Computer Society, 1996.

16 Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

17 D Kleitman. Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuit Theory*, 16(2):232–233, 1969.

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

19 Nathan Linial, László Lovász, and Avi Wigderson. Rubber bands, convex embeddings and graph connectivity. *Comb.*, 8(1):91–102, 1988.

20 Shaobin Liu, Kam-Hoi Cheng, and Xiaoping Liu. Network reliability with node failures. *Networks*, 35(2):109–117, 2000.

21 Yang P. Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. *CoRR*, abs/2003.08929, 2020.

22 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(5&6):583–596, 1992.

23 Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *STOC*, pages 241–252. ACM, 2019.

24 Manfred Padberg and Giovanni Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Math. Program.*, 47:19–36, 1990.

25 Azzeddine Rigat. An experimental study of k-vertex connectivity algorithms. *INFOCOMP*, 11, 2012.

26 Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Netw. Sci.*, 4(4):508–530, 2016.

27 Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *FOCS*, pages 919–930. IEEE, 2020.

28 Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. Generating random hyperbolic graphs in subquadratic time. In *ISAAC*, volume 9472 of *Lecture Notes in Computer Science*, pages 467–478. Springer, 2015.

29 Douglas R. White and Frank Harary. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociological Methodology*, 31(1):305–359, 2001.

## A Omitted Proofs

### A.1 Correctness

To show that any algorithm among Local1, Local1+, Local2, and Local2+ is LocalEC, it is enough to prove that it satisfies two properties:

▶ **Property 1.** *If $V(T)$ is returned, then $|E(V(T), V - V(T))| < k$ and $\emptyset \neq V(T) \subsetneq V$.*

▶ **Property 2.** *If there is a vertex-set $S$ satisfying Equation (1), then $\perp$ is returned with probability at most $1/2$.*

The following simple observation is due to [4].

▶ **Observation 8.** *Let $S$ be a vertex-set in graph $G$ and $x \in S$. Let $P$ be a path from $x$ to $y$. Let $G'$ be $G$ after reversing all edges along $P$. If $y \in S$, then $|E_{G'}(S, V - S)| = |E_G(S, V - S)|$. Otherwise, $|E_{G'}(S, V - S)| = |E_G(S, V - S)| - 1$.*

For the first property, the following argument works for all four algorithms.

▶ **Lemma 9.** *Local1, Local1+, Local2 and Local2+ satisfy Property 1.*

**Proof.** Let $S = V(T)$ be the cut the algorithm returned. Observe that $x \in S$ by design. By Observation 8, each iteration can only reduce the number of crossing edges by at most one. This can happen at most $k - 1$ times before the final iteration, which implies that initially' $|E(S, V - S)| \leq k - 1$. ◀

For the second property, the following argument works for Local1, and Local1+

▶ **Lemma 10.** *Local1 and Local1+ satisfy Property 2.*

**Proof.** We focus on proving that LOCAL1 satisfies Property 2 (the proof for Local1+ will be essentially identical). If the algorithm terminates before the $k$-th iteration, then it outputs $V(T)$, and thus $\perp$ is never returned. So now we assume that the algorithm terminates at the $k$-th iteration. Let $y_1, \dots y_{k-1}$ be the sequence of chosen path endpoints $y$ in DFS iterations. We first bound the probability that $y_i \in S$. Let $\mathrm{vol}_i^{\mathrm{out}}(S)$ be the volume of $S$ at iteration $i$. So,

$$\Pr(y_i \in S) \leq \frac{\mathrm{vol}_i^{\mathrm{out}}(S)}{8\nu k} \leq \frac{\mathrm{vol}^{\mathrm{out}}(S)}{8\nu k} \leq \frac{\nu}{8\nu k} = \frac{1}{8k}. \tag{2}$$

The first inequality follows by design. The second inequality follows by Observation 8.

By Observation 8, the algorithm can only return $\perp$ at the final iteration if at least one of the $y_i$'s is in $S$ (or if there is not viable cut). Let $\mathbb{1}[y_i \in S]$ be an indicator function. Let $Y = \sum_{i \leq k-1} \mathbb{1}[y_i \in S]$. Observe that $Y \geq 1$ if and only if the algorithm outputs $\perp$. We now bound the probability that $Y \geq 1$. By linearity of expectation, we have $\mathbb{E}[Y] = \sum_{i \leq k-1} \mathbb{E}[\mathbb{1}[y_i \in S]] = \sum_{i \leq k-1} \Pr(y_i \in S) \leq \frac{1}{8}$. Therefore, by Markov's inequality, we have

$$\Pr(Y \geq 1) = \Pr(Y \geq 8 \cdot \frac{1}{8}) \leq \Pr(Y \geq 8\mathbb{E}[Y]) \leq \frac{1}{8}. \tag{3}$$

This completes the proof for LOCAL1. To see that the same proof works for LOCAL1+, observe that the proof above (Equation (2) in particular) does not use the identity of the edges. Outgoing edges of a vertex are interchangible. The degree counting variant counts edges ensures that each outgoing edge for visited vertices is included in the collection of edges without collecting explicitly. The precomputed random number $\tau$ corresponds to a random edge from the collection. ◀

It remains to prove the second property for Local2 and Local2+. However, the arguments for Local2 and Local2+ are very similar to Local1 and Local1+:

▶ **Lemma 11.** *Local2 and Local2+ satisfy Property 2.*

**Proof.** For Local2, each edge in $E(S, V)$ has a $\frac{1}{8\nu}$ probability to be chosen if the edge is visited. The probabilities are not independent but can be used for Markov's inequality. If $Y$ is the number of edges in $E(S, V)$ that are chosen, or equivalently the number of times a vertex in $S$ is chosen, we have $\mathbb{E}[Y] \leq \frac{\nu}{8\nu} = \frac{1}{8}$, resulting in the same equation as Equation (3). If we consider the case where all edges in $E(S, V)$ are visited in a single iteration, we can see that the bound is tight. For Local2+, apply the same logic to $c(v)$ instead of edges. ◀

## B Full Near-Linear Vertex Connectivity Algorithm

### B.1 Vertex Connectivity via Local Edge Connectivity in Undirected Graphs

In this section, we describe the vertex connectivity algorithm that we implement in this paper. We will assume that we have a LocalEC algorithm (Definition 1) with time complexity $O(\nu k^2)$.

Let $G$ be a directed graph with n vertices and m edges, such that $(x, y) \in E(G) \iff (y, x) \in E(G)$. This is a directed representation of an undirected graph. Given a positive integer $k$, the following algorithm, which is very closely based on the framework by Nanongkai et al. [23], finds a minimum vertex cut of size less than $k$ or certifies that $\kappa \geq k$ with constant probability. Let $k'$ be the size of the minimum cut found so far in the algorithm, or $k$ if no cut has been found yet.

Suppose that there is a vertex cut in $G$, represented by a separation triple $(L, S, R)$. Assume without loss of generality that $\mathrm{vol}^{\mathrm{out}}(L) \leq \mathrm{vol}^{\mathrm{out}}(R)$. If $\mathrm{vol}^{\mathrm{out}}(L) < 2\delta$, where $\delta$ is the minimum degree, then $|L| = 1$. We find $\delta$ and such trivial cuts with a linear sweep.

Fix some value $a = \Theta(m/k)$, which must be a valid value for the parameter $\nu$ in LocalEC.

**Balanced Cut.** Suppose that $\mathrm{vol}^{\mathrm{out}}(L) \geq a$. If we sample pairs of edges $(x, x'), (y, y') \in E(G)$ we can show that $x \in L, y \in R$ with probability $\Theta(a/m)$ for each sample. We can find a x-y vertex cut of size less than $k'$ if one exists by using a max flow algorithm on the split graph through a well-known reduction (e.g. [7]). A sample size of $\Theta(m/a)$ is sufficient to find such a cut with high probability.

**Unbalanced Cut.** Now, for $\nu \in \{2^i\delta | i \in \mathbb{Z}^{\geq 0}, 2^i\delta < a\}$, i.e., power of two multiples of $\delta$ up to $a$. we sample $\Theta(m/\nu)$ edges $(x, x') \in E(G)$ and run LocalEC$(x_{\mathrm{out}}, \nu, k')$ on the split graph for each $x$. If $\mathrm{vol}^{\mathrm{out}}(L) = \Theta(\nu)$, the probability that any given edge yields $x \in L$ is $\Theta(\nu/m)$, which means that a sample size of $\Theta(m/\nu)$ is sufficient to find one with high probability. Let $L' = \{x_{\mathrm{in}}, x_{\mathrm{out}} | x \in L\} \cup \{x_{\mathrm{in}} | x \in S\}$. $L'$ is one side of an edge cut that corresponds to the vertex cut $S$, as in the reduction used for x-y connectivity for balanced cuts. We can show that $\mathrm{vol}^{\mathrm{out}}(L') = \frac{k+1}{k}\mathrm{vol}^{\mathrm{out}}(L) + k = \Theta(\mathrm{vol}^{\mathrm{out}}(L))$. Clearly, if $\mathrm{vol}^{\mathrm{out}}(L) = o(a)$, we will run LocalEC with some value $\nu$ for a sufficient sample size to find the cut with high probability.

In practice, if $\mathrm{vol}^{\mathrm{out}}(L) = \Theta(a)$, there is a fairly high probability to find the cut both with the max flow algorithm and LocalEC. At $\frac{a}{2}$, the max flow algorithm finds the cut at approximately half the probability at $a$. LocalEC, when configured to find cuts with reasonably high probability at $\nu$ will also often find cuts at higher volumes with diminishing probability as the actual volume goes up.

If we do not start with some $k > \kappa$, we can find one by doubling $k$ until we find a cut. When a cut can be found, a minimum cut will be find with high probability.

**Time Complexity.** Assuming Ford-Fulkerson max flow that runs in $\Theta(mk)$ time, the running time for finding balanced cuts is $\Theta(mk)\Theta(m/(m/k)) = \Theta(mk^2)$. Assuming LocalEC that runs in $O(\nu k^2)$ time, the running time for each of the $\Theta(\log(m/k))$ values for the parameter $\nu$ is $\Theta(\nu k^2)\Theta(m/\nu) = \Theta(mk^2)$. Due to preprocessing by Nagamochi and Ibaraki, which runs in $\Theta(m)$ time, we have $m = \Theta(nk)$, for a final time complexity of $\Theta(m + k^3 n \log n)$. If we repeat for high rather than constant probability we square the logfactor.

## B.2 Implementation Details

We use the following numbers for the unspecified values above: $a = \frac{m}{3k}$, $\frac{m}{a} = 3k$ samples for Ford-Fulkerson and $\lfloor \frac{m}{\nu} \rfloor$ for LocalEC. For Local1 and Local1+ we collect/count $2\nu k$ edges rather than $8\nu k$ and for Local2+ we count to $3\nu$ rather than $8\nu$. Local2+ seems to need a slightly higher factor for similar success rate.

The graph implementation used for this paper is based on adjacency lists with c++ vectors. When we reverse edges along a path we save the relevant vector indices to enable us to perform the opposite operations later, in order from the newest reversed path to the oldest. We store information such as DFS visited vertex flags and the number of uncounted edges/coins in LOCAL2+ per vertex. To avoid resetting this information for every vertex, we also maintain lists of vertices that have been visited within the most recent DFS or LocalEC call.

## C Preflow-push based Vertex Connectivity Algorithm

We use the algorithm by Henzinger, Rao and Gabow [15] with only minor optimisations. We omit most details here. The core algorithm uses a preflow based algorithm to calculate the minimum $S_i x_i$-cut, where $S_i = \{x\} \cup \{x_j : j < i\}$, for each vertex $x_i$ not adjacent to x. The algorithm maintains an "awake" set $W$ of vertices from where the current sink may be reachable. If there exists a minimum vertex cut $S \ni x$, which is very probable for small $\kappa$, then the minimum of these cuts will be a minimum vertex cut. The algorithm is repeated if needed to achieve a 50% or lower error rate, which should not be the case for any included test case. As with the algorithms by Forster et al. [9], we use the spit graph reduction and the sparsification algorithm by Nagamochi and Ibaraki [22] to reduce the average degree of the graph to at most $k$, doubling $k$ until we find a cut smaller than $k$. In case of weighted edges, dynamic trees would be used to improve time complexity, but this article only uses unweighted edges.

On page 10 of [15], Henzinger et al. describe a guaranteed method of doubling $k$ to find some $k \in (\kappa, 4\kappa)$. There, the algorithm is run on an arbitrary nonrandom vertex of degree $k$. To obtain an optimal cut with any probability guarantee, the algorithm needs to be repeated on a random seed vertex. We use random seed vertices during doubling to avoid having to repeat the algorithm after already finding a cut of size less than $k$. For small $k$, the "bad case" of not finding a cut despite $\kappa < k$ is highly unlikely.

On page 20 of [15], Henzinger et al. describe multiple auxiliary data structures used to achieve the desired time complexity. One of these is a partition of vertices in the awake set $W$ by their current distance values. We add another auxiliary data structure that stores the index of a vertex in this data structure to speed up finding and removing a vertex, which happened frequently enough to create a CPU hotspot.

# Parallel Five-Cycle Counting Algorithms

**Louisa Ruixue Huang** ✉
MIT, CSAIL, Cambridge, MA, USA

**Jessica Shi** ✉
MIT, CSAIL, Cambridge, MA, USA

**Julian Shun** ✉
MIT, CSAIL, Cambridge, MA, USA

## Abstract

Counting the frequency of subgraphs in large networks is a classic research question that reveals the underlying substructures of these networks for important applications. However, subgraph counting is a challenging problem, even for subgraph sizes as small as five, due to the combinatorial explosion in the number of possible occurrences. This paper focuses on the five-cycle, which is an important special case of five-vertex subgraph counting and one of the most difficult to count efficiently.

We design two new parallel five-cycle counting algorithms and prove that they are work-efficient and achieve polylogarithmic span. Both algorithms are based on computing low out-degree orientations, which enables the efficient computation of directed two-paths and three-paths, and the algorithms differ in the ways in which they use this orientation to eliminate double-counting. We develop fast multicore implementations of the algorithms and propose a work scheduling optimization to improve their performance. Our experiments on a variety of real-world graphs using a 36-core machine with two-way hyper-threading show that our algorithms achieves 10–46x self-relative speed-up, outperform our serial benchmarks by 10–32x, and outperform the previous state-of-the-art serial algorithm by up to 818x.

**2012 ACM Subject Classification** Theory of computation → Shared memory algorithms; Theory of computation → Graph algorithms analysis; Computing methodologies → Shared memory algorithms

**Keywords and phrases** Cycle counting, parallel algorithms, graph algorithms

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.2

**Supplementary Material** *Software (five-cycle counting code)*: `https://github.com/ParAlg/gbbs/tree/master/benchmarks/CycleCounting`
archived at `swh:1:dir:ce4ed35f4a35d29a32825f5e1b982e74c1497b87`

**Funding** This research was supported by NSF Graduate Research Fellowship #1122374, DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

## 1 Introduction

Subgraph or graphlet counting is a long standing research topic in graph processing with rich applications in bioinformatics, social network analysis, and network model evaluation [25, 14, 20, 21]. While there has been significant recent work on counting subgraphs of size three or four [18, 19, 2], counting subgraphs of size five or more is a difficult task even on the most modern hardware due to the massive number of such subgraphs in large graphs. As the subgraph sizes grow, the number of possible subgraphs grows exponentially.

We consider specifically the efficient counting of five-cycles. This pattern is particularly important for fraud detection [35]. Compared to other connected five-vertex patterns, five-cycles are much more difficult to count because they are the only such pattern that requires first counting all directed three-paths. Notably, the Efficient Subgraph Counting Algorithmic

PackagE (ESCAPE), a software package by Pinar et al. that serially counts all five-vertex subgraphs in large graphs [34], spends between 25–58% of the total runtime on counting five-cycles alone based on our measurement.

While there has been prior work on developing and implementing serial five-cycle counting algorithms [27, 34, 24], there has been no prior work on designing and implementing theoretically-efficient and scalable parallel five-cycle counting algorithms. We focus on designing multicore solutions, as all publicly-available graphs (which have up to hundreds of billions of edges [31]) can fit on a commodity multicore machine [16, 17].

We present two new parallel five-cycle counting algorithms that not only have strong theoretical guarantees, but are also demonstrably fast in practice. These algorithms are based on two different serial algorithms, namely by Kowalik [27] and from ESCAPE by Pinar et al. [34]. Kowalik studied $k$-cycle counting in graphs for $k \leq 6$ and proposed a five-cycle counting algorithm that runs in $O(md^2) = O(m\alpha^2)$ time for $d$-degenerate graphs [27], where $m$ is the number of edges in the graph and $\alpha$ is the arboricity of the graph.[1] The ESCAPE implementation contains a five-cycle counting algorithm that, with an important modification that we make, achieves the same asymptotic complexity of $O(m\alpha^2)$ [34]. The arboricity of a graph is a measure of its sparsity, and having running times parameterized by $\alpha$ is desirable since most real-world graphs have low arboricity [16].

The main procedure in both algorithms and the essential modification to the ESCAPE algorithm is to first compute an appropriate arboricity orientation of the graph in parallel, where the vertices' out-degrees are upper-bounded by $O(\alpha)$. This orientation then enables the efficient counting of directed two-paths and three-paths, which are then appropriately aggregated to form five-cycles. Notably, the counting and aggregation steps can each be efficiently parallelized. The two algorithms differ fundamentally in the ways in which they use the orientations of these path substructures to eliminate double-counting. We prove theoretical bounds that show that both of our algorithms match the work of the best sequential algorithms, taking $O(m\alpha^2)$ work and $O(\log^2 n)$ span with high probability (w.h.p.).[2]

We present optimized implementations of our algorithms, which use thread-local data structures, fast resetting of arrays, and a new work scheduling strategy to improve load balancing. We provide a comprehensive experimental evaluation of our five-cycle counting algorithms. On a 36-core machine with 2-way hyperthreading, our parallel algorithms achieve between 10–46x self-relative speed-up, and between 162–818x speed-ups over the fastest prior serial five-cycle counting implementation, which is from ESCAPE [34]. We also implement our own serial versions of the two algorithms, which are 7–38.91x faster than ESCAPE's algorithm due to improved theoretical work complexities. Our best parallel algorithms achieve between 10–32x speed-ups over our best serial algorithms. Our parallel five-cycle counting code is available at `https://github.com/ParAlg/gbbs/tree/master/benchmarks/CycleCounting`.

## 2 Background and Related Work

The difficulty of cycle counting has attracted considerable research effort over the years. Counting the number of $k$-cycles with $k$ as an input parameter is NP-complete since it includes the problem of finding a Hamiltonian cycle. However, efficient algorithms have been developed to count $k$-cycles for $k \leq 5$. Notably, Alon et al. [3] developed algorithms for

[1] A graph is $d$-degenerate if every subgraph has a vertex of degree at most $d$, and a graph has arboricity $\alpha$ if the minimum number of spanning forests needed to cover all of the edges of the graph is $\alpha$.

[2] With high probability (w.h.p.) means that the probability is at least $1 - 1/n^c$ for some constant $c > 0$ for an input of size $n$.

efficiently finding a $k$-cycle for general $k$, but these translate to efficient $k$-cycle counting algorithms only for planar graphs where $k \leq 5$. For $k = 3, 4$, Chiba and Nishizeki [15] proposed algorithms that take $O(m\alpha)$ time. More recently, Bera et al. [7] analyzed the subgraph counting problem for $k = 5$ and gave an algorithm that takes $O(m\alpha^3)$ time, and the five-cycle counting part of the algorithm takes $O(m\alpha^3)$ time. However, it is shown in the same study that this result is unlikely to be extended to $k > 5$, due to the Triangle Detection Conjecture, which puts a lower bound of $\Omega(m^{1+\gamma})$ time with $\gamma > 0$ on any triangle detection algorithm on an input graph with $m$ edges [1]. If the conjecture holds, a reduction of the triangle detection problem to the six-cycle counting problem implies that there cannot be a $o(f(\alpha)m^{1+\gamma})$ time algorithm for six-cycle counting.

Until recently, because of the high computational power required, exact five-vertex subgraph counting was often deemed impractical on graphs with more than a few million edges. Most effort has focused on obtaining approximate counts or approximate graphlet frequency distributions [46, 10, 36]. Hocevar and Demsar [24] developed Orca to count subgraphs of up to size five and tested them on graphs with tens of thousands of vertices. Pinar et al. [34] developed ESCAPE, which is the first package that aims to perform exact counting of all five-vertex subgraphs on moderately large graphs. However, ESCAPE does not exploit parallelism and is not optimized for cycle-counting. Kowalik [27] gave a serial algorithm for five-cycle counting that takes $O(m\alpha^2)$, the best known theoretical bound for five-cycle counting, but does not provide an implementation. In Section 4, we describe Kowalik's and Pinar et al.'s five-cycle counting algorithms in more detail.

While there has not been prior work on parallel five-cycle counting algorithms, parallel cycle counting algorithms for smaller cycles have been studied over the years. Specifically, for the case of three-cycles, or triangles, there has been a significant amount of attention over the past two decades (e.g., [40, 43, 33, 8], among many others).

Moreover, fast sequential algorithms for four-cycles have been studied extensively. For bipartite graphs, four-cycles, also known as butterflies, are the smallest non-trivial subgraphs. Chiba and Nishizeki's [15] described a four-cycle counting algorithm that takes $O(m\alpha)$ work by using a degree ordering of the graph. Subsequently, butterfly counting algorithms using degree ordering and other orderings have also been designed [47, 44, 38, 48, 39].

There have been fewer studies on parallel four-cycle counting algorithms. The Parametrized Graphlet Decomposition package by Ahmed et al. [2] provides efficient parallel implementations of exact counting of subgraphs of up to size four, including four-cycles. Wang et al. [44] implement a distributed algorithm using MPI that partitions the vertices across processors, where each processor sequentially counts the number of butterflies for vertices in its partition. Shi and Shun [42] presented a framework for parallel butterfly counting with several algorithms achieving $O(m\alpha)$ expected work and $O(\log m)$ span with high probability. Wang et al. [45] describe a similar parallel butterfly counting algorithm, with an additional cache optimization in their implementation.

## 3 Preliminaries

**Graph Notation.** The input to our algorithms is a simple, undirected, unweighted graph $G(V, E)$. The number of vertices is $|V| = n$ and the number of edges is $|E| = m$. Vertices are labeled $0, 1, \ldots, n-1$. In our analysis, we assume that $m = \Omega(n)$. For a vertex $v$, we use $N(v)$ to denote the neighbors of $v$ and $\deg(v)$ to denote the degree of v. When discussing directed graphs, $N^{\rightarrow}(v)$ denotes $v$'s out-neighbors and $N^{\leftarrow}(v)$ denote the in-neighbors.

Furthermore, we use $N_v(u)$ ($N_v^{\rightarrow}(u)$ for directed graphs) to represent the neighbors of vertex $u$ that are after $v$ given a non-increasing degree ordering. When vertices are relabeled by non-increasing degree order, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

The ***arboricity*** of a graph $G$, denoted $\alpha(G)$, is defined as the minimum number of spanning forests needed to cover the graph. $\alpha(G)$ is known to be upper-bounded by $O(\sqrt{m})$. Due to the fact that graphs modeling the real world tend to be sparse, $\alpha(G)$ tends to be small for graphs that we are interested in processing. It is also known that $\sum_{(u,v)\in E}\min(\deg(u),\deg(v)) = O(m\alpha(G))$ [15]. Closely related to arboricity is the ***degeneracy*** of a graph $G$, $d(G)$, or the smallest $k$ such that every subgraph of $G$ contains a vertex of degree at most $k$. It is known that $d(G) = \Theta(\alpha(G))$ [32]. As such, our asymptotic bounds can use $\alpha(G)$ or $d(G)$ interchangeably. When it is unambiguous, we write the arboricity and degeneracy of a graph as $\alpha$ and $d$, respectively.

**Graph Format.** For the theoretical analysis, we assume the graphs are stored in hash tables in the adjacency list format, to obtain constant time edge queries. In our implementations, graphs are stored in Compressed Sparse Row (CSR) format, which is more compact and has better cache locality.

**The Work-Span Model.** To analyze the complexity of our parallel algorithms, we use the work-span model [26] with arbitrary forking. In this model, a computation is seen as a series-parallel DAG. Each instruction is a vertex, and sequential executions in a thread are composed in series and different children threads forked together are composed in parallel. The ***work*** in a computation is the total number of vertices and the ***span*** is the length of the longest path in the computation graph. The work of a sequential algorithm is the same as its time. A ***work-efficient*** parallel algorithm has a work complexity matching the time of the best sequential algorithm for the problem. For an algorithm with work $W$ and span $S$, the running time on $P$ processors is upper bounded by $W/P + S$ [13]. Since the number of processors in practice is modest, it is important to be work-efficient in addition to minimizing the span of the computation.

**Parallel Primitives.** We use the following parallel primitives. A parallel for-loop (**parfor**) with $n$ iterations that can be executed in parallel launches all of its iterations in $O(n)$ work and $O(1)$ span. ***Parallel integer sort*** sorts $n$ integers in the range $[0, O(n)]$ in $O(n)$ work and $O(\log n)$ span w.h.p. [37]. We also use ***parallel hash tables***, which support a batch of $n$ instructions in $O(n)$ work and $O(\log^* n)$ span w.h.p. [22]. We assume atomic adds take $O(1)$ work and span.

## 4 Five-Cycle Counting Algorithms

In this section, we present two new parallel algorithms for counting five-cycles. The first algorithm is based on the serial algorithm by Kowalik [27]. Kowalik shows that the algorithm achieves a time complexity of $O(m)$ on planar graphs, in which $\alpha = O(1)$, or $O(md^2) = O(m\alpha^2)$ on $d$-degenerate graphs. The second algorithm is based on the serial algorithm by Pinar et al. in their ESCAPE framework for counting all 5-vertex subgraphs in a graph [34]. We show that both of the parallel algorithms that we design are provably work-efficient with polylogarithmic span.

### 4.1 Preprocessing: Graph Orientation

Similar to many previous subgraph counting algorithms [7, 41], a key step in our algorithms is a preprocessing step that orients the graph $G$, creating a directed acyclic graph $G^{\rightarrow}$ where the out-degrees of vertices are upper-bounded. We use ***$l$-orientation*** to refer to an orientation where each vertex's out-degree is bounded by $l$. Furthermore, orientations in our context are always induced from a total ordering of the vertices, where directed edges point from vertices lower in the ordering to vertices higher in the ordering. As such, the problem of orienting the graph is reduced to the problem of finding an appropriate ordering of the vertices.

**Degree Orientation.** The core idea of orienting an undirected input graph based on ordering the vertices by non-increasing degree to perform subgraph counting or listing is attributed to Chiba and Nishizeki [15]. Using degree ordering, they proposed efficient triangle and four-cycle counting algorithms based on this key result:

▶ **Lemma 1** ([15]). *For a graph $G = (V, E)$, $\sum_{(u,v)\in E} \min\{deg(u), deg(v)\} \leq 2\alpha m$.*

This result allows us to bound the number of wedges in graph $G$ by $2m\alpha$, where a wedge is defined as a triple $(v, w, u)$ where $(v, w), (w, u) \in E$, $\deg(v) \geq \deg(w)$ and $\deg(v) \geq \deg(u)$. In Kowalik's five-cycle algorithm, wedges are the building blocks of five-cycles, and we can show that $O(\alpha)$ work is done for each wedge. Combined with the $O(m\alpha)$ bound on the number of wedges, this gives us the $O(m\alpha^2)$ running time bound.

**Arboricity Orientation.** An ***arboricity orientation*** of a graph is one where the vertices' out-degrees are upper-bounded by $O(\alpha(G))$. An arboricity-oriented graph has slightly different theoretical properties compared to a degree-oriented graph, but literature has shown that in some algorithms arboricity orientation can achieve the same practical efficiency as degree orientation [41]. We note that in Kowalik's five-cycle counting algorithm as well as our parallelization of the algorithm, both a degree ordering and an arboricity ordering are required to achieve work-efficiency.

One way to obtain an arboricity orientation is by computing the degeneracy ordering using a standard $k$-core decomposition algorithm [30, 6]. The algorithm repeatedly removes the vertex with the lowest degree from the graph. When we direct edges using this orientation, we obtain a DAG where each vertex's out-degree is bounded by $d(G)$. While this algorithm can be parallelized to be work-efficient, it does not attain polylogarithmic span; notably, the problem is P-complete [4].

Since the parallel algorithm for exact degeneracy ordering has sub-optimal span, we use approximate algorithms with polylogarithmic span. We test two such algorithms: Goodrich-Pszona and Barenboim-Elkin. Both algorithms work by peeling low-degree vertices in batches. Goodrich and Pszona originally designed the algorithm in the external-memory model [23], while Barenboim and Elkin designed the algorithm for a distributed model [5]. Shi et al. adapted both algorithms for shared memory and showed that both compute an $O(\alpha)$-orientation in $O(m)$ work and $O(\log^2 n)$ span (one of which is deterministic and the other of which is randomized) [41]. A different algorithm with the same (deterministic) work and span bounds was described by Besta et al. [9].

## 4.2 Kowalik's Algorithm

We present in Algorithm 1 our parallelization of Kowalik's serial five-cycle counting algorithm [27]. In this algorithm, vertices are sorted and processed in non-increasing degree order. Each vertex is processed by counting all five-cycles with the vertex itself as the lowest-ranked (i.e., highest-degree) vertex. After processing all vertices, each five-cycle is counted exactly once and the counts are summed and outputted.

Recall that we use $N_v(u)$ ($N_v^{\rightarrow}(u)$ for directed graphs) to represent the neighbors of vertex $u$ that are after $v$ in the non-increasing degree ordering. Since the vertices are relabeled by non-increasing degree order on line 3, we can easily obtain $N_v(u) = N(u) \cap \{w \in V \mid w > v\}$.

**Algorithm 1** Kowalik's Five-Cycle Counting Algorithm Parallelized.

```
 1: procedure COUNT-FIVE-CYCLES($G = (V, E)$)
 2:     $\#_c \leftarrow 0$
 3:     Relabel vertices of $G$ such that $d(0) \geq d(1) \geq \cdots \geq d(n-1)$
 4:     Orient $G$ using arboricity orientation to produce $G^{\rightarrow}$
 5:     parfor $v \leftarrow 0$ to $n-1$ do
 6:         Initialize an empty parallel hash table $U_v$
 7:         parfor $u \in N_v(v)$ do
 8:             parfor $w \in N_v(u)$ do $U_v[w] \leftarrow U_v[w] + 1$
 9:         parfor $u \in N_v(v)$ do
10:             Initialize an empty parallel hash table $T_{v,u}$
11:             parfor $w \in N_v(u)$ do
12:                 $T_{v,u}[w] \leftarrow 1$
13:             parfor $w \in N_v(u)$ do
14:                 parfor $x \in N_v^{\rightarrow}(w)$ do
15:                     if $x \neq u$ then
16:                         if $w \in N^{\rightarrow}(v)$ or $v \in N^{\rightarrow}(w)$ then
17:                             $\#_c \leftarrow \#_c + U_v[x] - T_{v,u}[x] - 1$
18:                         else
19:                             $\#_c \leftarrow \#_c + U_v[x] - T_{v,u}[x]$
20:     return $\#_c$
```

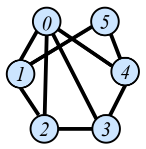
We now focus on the iteration $v$ of the outer for-loop. An example is shown in Figure 1. We note that for each $v$, we consider only vertices ranked higher than $v$ to complete five-cycles containing $v$. Lines 7–8 count in a parallel hash table $U_v$ all wedges, where $v$ is one of the endpoints and $v$ is the lowest-ranked vertex in the wedge. Then, lines 11–12 store in a parallel hash table $T_{v,u}$ all wedges where $v$ is one of the endpoints, $v$ is the lowest-ranked vertex in the wedge, and $u$ is the center. Both hash tables are indexed on $w$, the other endpoint of the wedge.

On each iteration of the loop in line 13, the algorithm counts all five-cycles that contain the wedge $v$ — $u$ — $w$. To accomplish this, the algorithm iterates through each neighbor $x$ of $w$ in $G^{\rightarrow}$, and considers the number of wedges that $x$ shares with $v$, which is stored in $U_v[x]$. Note that three vertices of the cycle are given ($v$, $u$, and $w$), so the algorithm must ensure that the two vertices used to complete the cycle do not include these existing vertices. Line 15 ensures that $x \neq u$ in the cycle; note that $x \neq w$ because the graph is assumed to not contain self-loops, and $x \neq v$ by definition of $N_v^{\rightarrow}$. Lines 16–19 check if $v$ and $w$ are neighbors; if so, then the number of wedges ending in $x$ includes the wedge $v$ — $w$ — $x$, which does not properly complete a five-cycle. In this case, there is one fewer five-cycle completed by the wedges ending in $x$, and so we subtract one on line 17. Finally, note that if there exists the wedge $v$ — $u$ — $x$, then this similarly does not properly complete a five-cycle, so we subtract $T_{v,u}[x]$, which stores precisely this wedge. We assume that indexing an entry that does not exist in a hash table returns a value of 0.

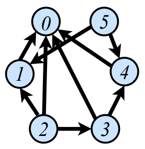
As every thread operates on the variable $\#_c$, we use atomic add for all of these operations, which takes $O(1)$ work. In practice, we use thread-local variables to keep the count and sum them in the end to avoid heavy contention. We now show that the parallel algorithm is work-efficient and has polylogarithmic span.

▶ **Theorem 2.** *Algorithm 1 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p., and $O(m\alpha)$ space on a graph with $m$ edges and arboricity $\alpha$.*

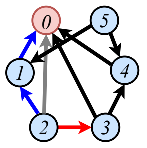
**Proof.** For line 3, we sort $n$ integers in the range $[0, n-1]$, which can be done in $O(n)$ work and $O(\log n)$ span w.h.p. using parallel integer sorting [37]. As discussed in Section 4.1, line 4 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [41]. As a result, the for-loops on



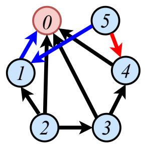
Example graph $G$.



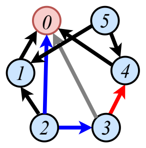
$G^{\rightarrow}$ under degree orientation.



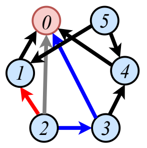
**(a)** $U_0 = [0, 1, 1, 2, 1, 1]$
$\#_c$ += 2 - 1.



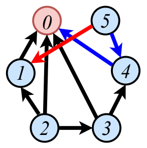
**(b)** $U_0 = [0, 1, 1, 2, 1, 1]$
$\#_c$ += 1.



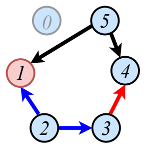
**(c)** $U_0 = [0, 0, 2, 1, 1, 2]$
$\#_c$ += 1 - 1.



**(d)** $U_0 = [0, 1, 1, 2, 0, 2]$
$\#_c$ += 1 - 1.



**(e)** $U_0 = [0, 1, 2, 1, 1, 1]$
$\#_c$ += 1.



**(f)** $U_1 = [0, 0, 0, 0, 1, 0]$
$\#_c$ += 1.

**Figure 1** This figure outlines steps in our parallelization of Kowalik's five-cycle counting algorithm where $\#_c$ is updated (Algorithm 1). Each subfigure considers a different $\{u, v, w, x\}$ from lines 13–14, and the corresponding $U_v$ is displayed for each subfigure. For simplicity, the $U_i$ hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint, and the updates to the parallel hash tables $T_{v,u}$ in lines 11–12 of Algorithm 1 are shown as subtracted directly from the corresponding $U_v$ for a fixed $u$ from line 9.
The vertices have already been relabeled by non-increasing degree and the entries in each $U_v$ have already been computed (lines 10–12). The vertex $v$ that we are considering on line 5 is colored in red. The edges colored in blue form wedges $v$ — $u$ — $w$, and the direction of those edges is irrelevant. The red edges represent the out-edge $w \rightarrow x$ on line 14. When $w$ and $v$ are neighbors (the edge is colored grey), the condition checked on line 16 returns true, and the subsequent line in each algorithm is executed (sub-figures (a), (c), and (d)). Otherwise, line 19 is executed (sub-figures (b), (e), and (f)). The final value of $\#_c$ is 4.

lines 7 and 9 iterating over $u \in N_v(v)$ take at most $\min(\deg(u), \deg(v))$ iterations, and by Lemma 1, the total number of times we iterate through $w \in N_v(u)$ on each of lines 8, 11, and 13 is at most $2m\alpha$.

Since parallel hash tables can perform a batch of $k$ operations in $O(k)$ work and $O(\log^* k)$ span w.h.p., the time complexities of lines 8 and 12 are given by $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p. Then, the for-loop of line 14 has at most $O(\alpha)$ iterations because of the $O(\alpha)$-orientation of the graph. In total, lines 15–19 are executed at most $O(\alpha) \cdot 2m\alpha = O(m\alpha^2)$ times, and again due to the parallel hash tables, the time complexity is given by $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p. In all, the total time complexity is given by $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

Finally, this algorithm uses $O(m\alpha)$ space. Based on Lemma 1, the total number of keys stored over all $U_v$'s is upper-bounded by $O(m\alpha)$, as is the number of keys stored over all $T_{v,u}$'s (over all pairs $(v, u)$). The parallel hash table's space usage is linear in the number of keys [22]. Hence, the total space usage is $O(m\alpha)$. ◀

**Algorithm 2** Five-Cycle Counting in ESCAPE Parallelized.

```
1: procedure Count-Five-Cycles($G = (V, E)$)
2:     $\#_c \leftarrow 0$
3:     Orient $G$ using arboricity orientation to produce $G^{\rightarrow}$
4:     parfor $v \leftarrow 0$ to $n - 1$ do
5:         Initialize an empty parallel hash table $U_v$
6:         parfor $w \in N^{\leftarrow}(v)$ do
7:             parfor $u \in N(w)$ do
8:                 $U_v[u] \leftarrow U_v[u] + 1$
9:         parfor $w \in N^{\rightarrow}(v)$ do
10:            parfor $u \in N^{\rightarrow}(w)$ do
11:                $U_v[u] \leftarrow U_v[u] + 1$
12:        parfor $u \in N^{\leftarrow}(v)$ do
13:            parfor $w \in N^{\leftarrow}(u)$ do
14:                parfor $x \in N^{\rightarrow}(w)$ do
15:                    if $x \neq v$ and $x \neq u$ then
16:                        $\#_c \leftarrow \#_c + U_v[x]$
17:                        if $w \in N(v)$ then
18:                            $\#_c \leftarrow \#_c - 1$
19:                        if $x \in N(u)$ then
20:                            $\#_c \leftarrow \#_c - 1$
21:    return $\#_c$
```

## 4.3 ESCAPE Algorithm

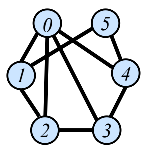
Another serial five-cycle counting algorithm is given by Pinar et al. as part of ESCAPE, which counts all 5-vertex subgraphs in a graph serially [34].

The first step of the ESCAPE five-cycle counting algorithm is to orient the graph. The ESCAPE framework uses degree orientation and achieves a time complexity of $O(m^2)$. We note that, if instead an arboricity orientation is used, the five-cycle counting algorithm achieves an improved time complexity of $O(m\alpha^2)$. We include this modification in our parallelization of the ESCAPE five-cycle counting algorithm to achieve work-efficient bounds. The proof of the serial time complexity with the arboricity orientation follows directly from the proof of our parallel algorithm.

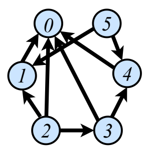
We present in Algorithm 2 our parallelization of the algorithm from ESCAPE, and an example is shown in Figure 2. We use $u \prec v$ to indicate that $u$ precedes $v$ in the ordering that produced the orientation, and so an edge from $u$ to $v$ exists in the directed graph $G^{\rightarrow}$ if and only if $u \prec v$.

After orienting the graph using an arboricity orientation (line 3), for each vertex $v$ (line 4), the algorithm counts all out-wedges and inout-wedges (see Figure 3). We denote the number of out-wedges with endpoints $v$ and $u$ by $W_{++}(v, u)$, and the number of inout-wedges with endpoints $v$ and $u$, starting with a directed edge out of $v$, by $W_{+-}(v, u)$. For each $v$, the algorithm computes $W_{++}(u, v) + W_{+-}(u, v)$ on lines 6–8 and $W_{+-}(v, u)$ on lines 9–11, and stores these counts in a parallel hash table $U_v$.

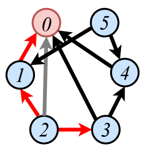
Figure 4 shows all possible orientations of acyclically directed five-cycles. We iterate over the 3-path shown in Figure 4 from vertex $v$ to vertex $x$ (lines 12–15), each of which can be completed by either an inout-wedge or an out-wedge with endpoints $v$ and $x$, assuming $x \neq v$ and $x \neq u$. Now, any orientation of a five-cycle has one of the three configurations shown in Figure 4, where exactly one of the vertices can be assigned to be $v$. Thus, every 3-path between a pair $(v, x)$ contributes $W_{+-}(v, x) + W_{++}(v, x) + W_{+-}(x, v)$ (which is stored in $U$ from lines 6–11) to the five-cycle count. However, this over-counts five-cycles since the wedge and the 3-path may overlap. Lines 16–20 deal with the over-counting when adding the number of wedges to the total count.



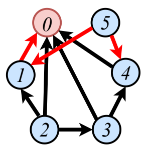
Example graph $G$.



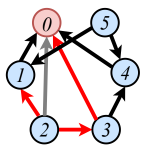
$G^{\rightarrow}$ under degree orientation.



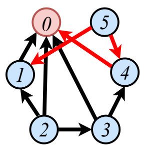
**(a)** $U_0 = [0, 1, 2, 2, 1, 2]$
$\#_c$ += 2 - 1.



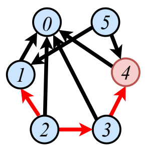
**(b)** $U_0 = [0, 1, 2, 2, 1, 2]$
$\#_c$ += 1.



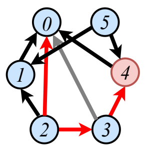
**(c)** $U_0 = [0, 1, 2, 2, 1, 2]$
$\#_c$ += 1 - 1.



**(d)** $U_0 = [0, 1, 2, 2, 1, 2]$
$\#_c$ += 1.



**(e)** $U_4 = [1, 1, 1, 0, 0, 0]$
$\#_c$ += 1.



**(f)** $U_4 = [1, 1, 1, 0, 0, 0]$
$\#_c$ += 1 - 1.

**Figure 2** This figure outlines steps in the ESCAPE five-cycle counting algorithm where $\#_c$ is updated (Algorithm 2). Each subfigure considers a different $\{u, v, w, x\}$ from lines 12–15, and the corresponding $U_v$ is displayed for each subfigure. For simplicity, the $U_i$ hash tables are depicted as arrays, with the appropriate wedge counts stored at the index on the corresponding endpoint. Note that the entries in each $U_v$ have already been computed (lines 6–11). The vertex $v$ that we are considering on line 4 is colored in red. The red edges represent the directed 3-paths $v \leftarrow u \leftarrow w \rightarrow x$ found on lines 12–15. Lines 17 and 19 check whether $v$ and $w$ or $u$ and $x$ are neighbors, respectively. When either of the conditions holds, the relevant edge is colored grey. Each grey edge subtracts one from the five-cycle count. Note that in sub-figures (a) and (c), the condition that $v$ is adjacent to $w$ from line 17 holds, and in sub-figure (f), the condition that $u$ is adjacent to $x$ from line 19 holds. In sub-figures (b), (d), and (e), neither conditions hold, and therefore 1 is not subtracted from the final count. The final value of $\#_c$ is 4.

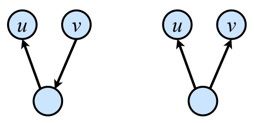


**Figure 3** An inout-wedge (left) and an out-wedge (right).

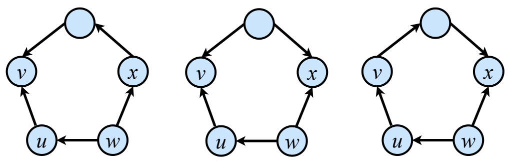


**Figure 4** All three possible orientations of directed five-cycles. All three forms have the component $v \leftarrow u \leftarrow w \rightarrow x$, which is a 3-path between $v$ and $x$. They are completed by an inout-wedge from $x$ to $v$, an out-wedge between $v$ and $x$, and an inout-wedge from $v$ to $x$, respectively.

In more detail, Line 16 first adds $U_v[x]$ to the count (again, assume that indexing an entry that does not exist in a hash table returns a value of 0). Line 17 checks if $w$ is adjacent to $v$; if so, depending on the direction of the edge between $w$ and $v$, there is either an out-wedge or an inout-wedge on $v$, $w$, and $x$, that does not complete a five-cycle with the 3-path. Line 18 subtracts the five-cycle counted for this case. Similarly, line 19 checks if $x$ is adjacent to $u$, and if so, there is either an out-wedge or an inout-wedge on $v$, $u$, and $x$, that does not complete a five-cycle; line 20 corrects this.

Similar to the parallelization of Kowalik's algorithm, in theory we use atomic adds for all of the increments on the $\#_c$ variable, and in practice we use thread-local variables.

▶ **Theorem 3.** *Algorithm 2 can be performed in $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p., and $O(m\alpha)$ space on a graph with $m$ edges and arboricity $\alpha$.*

**Proof.** As discussed in Section 4.1, line 3 can be implemented in $O(m)$ work and $O(\log^2 n)$ span [41]. Lines 6–11 go through all inout-wedges and out-wedges where $v$ is an endpoint. Because of the arboricity orientation, there are at most $m\alpha$ inout-wedges and out-wedges. Each wedge is counted at most twice, and so lines 6–11 incur $O(m\alpha)$ hash table operations, which takes $O(m\alpha)$ work and $O(\log^* n)$ span w.h.p.

There are $O(m\alpha^2)$ 3-paths (i.e., $v \leftarrow u \leftarrow w \rightarrow x$) and each is encountered exactly once in the triply-nested for-loop (lines 12–20). Again, by using an arboricity orientation, the algorithm executes lines 15–20 for at most $O(m\alpha^2)$ times, which due to the hash table operations, takes $O(m\alpha^2)$ work and $O(\log^* n)$ span w.h.p.

Overall, the algorithm takes $O(m\alpha^2)$ work and $O(\log^2 n)$ span w.h.p.

The parallel hash tables and the space to store the accumulated cycle counts account for all of the additional space usage. Since each wedge results in at most two additional keys in the hash tables, the number of keys in all of the hash tables $U_i$ is upper-bounded by twice the total number of out-wedges and inout-wedges. For the arboricity-oriented graph, there are $O(m\alpha)$ out-wedges and $O(m\alpha)$ inout-wedges, and so the number of keys across all hash tables is bounded by $O(m\alpha)$. Thus, the algorithm takes $O(m\alpha)$ space. ◀

## 4.4 Implementation

We implement the serial and parallel versions of Kowalik's algorithm and Pinar et al.'s algorithm using the Graph Based Benchmark Suite framework (GBBS) [16, 17]. GBBS provides many utilities for parallel algorithms, including sorting, parallel data structures, and implementations of the arboricity ordering algorithms mentioned above. In GBBS, graphs are represented in compressed sparse row (CSR) format. The compact representation improves memory locality, but this format does not allow us to check edge existence in $O(1)$ work, which is an operation required by both five-cycle counting algorithms. Using a separate data structure to store edges adversely affects locality, and so to improve performance, we sort the neighbor lists in the preprocessing step and use binary search to locate neighbors.

In our parallel implementation, we only parallelize the outer for-loop for each algorithm since there is sufficient parallelism provided by the outer for-loop alone. For Kowalik's algorithm, instead of using parallel integer sort, we use a cache-efficient implementation of parallel sample sort [11] provided by GBBS to sort the vertices by degree. We also use vertex-indexed size-$n$ arrays instead of parallel hash tables for $U_i$ and $T_{i,j}$. While hash tables have lower space usage for sparse graphs, they tend to have worse cache locality and are slower in practice. We introduce further practical optimizations below.

**Thread-local Data Structures.** As we parallelize the outer for-loop, the arrays $U_i$ in both algorithms must be allocated per iteration. We optimize this allocation by using the `parallel_for_alloc` construct in GBBS, which allocates one array per thread and reuses this space over iterations. Each iteration uses the array as a local array, and so this incurs no synchronization overhead. With this optimization, the algorithm only requires $O(Pn)$ space, where $P$ is the number of processors.

**Fast Reset.** Additionally, the thread-local arrays must be reset after each iteration of the outer for-loop. Depending on the structure of the graph, the array can be sparse, and naively resetting the entire array incurs $O(n^2)$ extra work, which is costly. We use a separate thread-local array to record the non-zero entries and reset only those entries after an iteration of the outer for-loop. The sparser the graph, the more effective this optimization is. This optimization at most doubles the space requirement for the algorithm, but drastically improves the running time by avoiding unnecessary writes.

**Work Scheduling.** The naive parallelization of the five-cycle counting algorithms blocks a fixed number of vertices together and processes them in series. For our experiments, we use a block size of 16, which we found to give the best performance in this setting. However, due to the nature of the algorithm, the amount of work per vertex is not uniform. This is particularly true for Kowalik's algorithm, which processes vertices in non-increasing degree order and deletes a vertex after processing it. The number of five-cycles that can be counted under a given vertex $v$ in the outermost loop falls off rapidly with the vertex's degree rank. In our work scheduling optimization, we block vertices together into groups that require similar amounts of work by estimating the work required for each vertex. We use the sum of the degrees of a vertex's neighbors as the estimator. That is, for each vertex $v$, we estimate the amount of work done on the vertex to be $\sum_{w \in N(v)} \deg(w)$.

## 5 Experiments

**Environment.** We run our experiments on a `c5.18xlarge` AWS EC2 instance, which is a dual-processor system with 18 cores per processor (2-way hyper-threading, 3.00GHz Intel Xeon(R) Platinum 8124M processors), and 144 GiB of main memory. We use Cilk Plus for parallelism [28, 12]. We use the `g++` compiler (version 8.2.1) with the `-O3` flag.

We test the performance of our two parallel five-cycle counting algorithms. Our parallel implementations use all of the optimizations described in Section 4.4, except that we test the performance with and without the work scheduling optimization. We compare the performance of the parallel implementations against our implementations of Kowalik's algorithm and the ESCAPE algorithm. We also tested the performance of the serial five-cycle counting algorithm in the ESCAPE package, the fastest known implementation of five-cycle counting. This algorithm is embedded inside the ESCAPE code for counting all five-vertex patterns, and so we obtained timings by running only the five-cycle counting portion of the code. We found our serial ESCAPE implementation to be 1.1–2.95x faster than the one provided in the ESCAPE package, and hence present only our running times in the tables.

We also test the effect of using different arboricity ordering algorithms. Besides Goodrich-Pszona, Barenboim-Elkin, and $k$-core, we also tested non-decreasing degree ordering as an approximation of degeneracy ordering. Intuitively, it limits the out-degree of the graph by directing edges from lower-degree vertices to higher-degree neighbors.

We perform these tests on a number of real-world graphs from the Stanford Network Analysis Platform [29]. Table 1 describes the properties of these graphs. All graphs are simple, unweighted, and undirected.

**Serial Five-cycle Counting.** Table 2 lists the running time of the two of serial five-cycle counting algorithms. Our serial Kowalik implementation always outperforms our serial ESCAPE implementation, and the difference in running times between the ESCAPE algorithm and Kowalik's algorithm grows as the graph size grows. The serial Kowalik algorithm achieves between 6.37–14.77x speed-up over our serial ESCAPE implementation, and between 7–38.91x speed-up over the original ESCAPE implementation.

**Table 1** Relevant statistics of our input graphs.

| Dataset | $\|V\|$ | $\|E\|$ | # 5-cycles |
|---|---|---|---|
| email-Eu-Core (email) | 1005 | 32128 | 245,585,096 |
| com-DBLP (dblp) | 425957 | $2.10 \times 10^6$ | 3,440,276,253 |
| com-YouTube (youtube) | $1.16 \times 10^6$ | $5.98 \times 10^6$ | 34,643,647,544 |
| com-LiveJournal (lj) | $4.03 \times 10^6$ | $6.94 \times 10^7$ | 6,668,633,603,006 |
| com-Orkut (orkut) | $3.27 \times 10^6$ | $2.34 \times 10^8$ | 42,499,585,526,270 |
| com-Friendster (friendster) | $1.25 \times 10^8$ | $3.61 \times 10^9$ | 96,281,214,210,322 |

**Table 2** Running times (seconds) of the two serial implementations and the two parallel five-cycle counting implementations without the work scheduling optimization. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. For the serial algorithms, $T_E$ is our implementation of the ESCAPE algorithm with arboricity orientation, and $T_K$ is our implementation of the serial Kowalik's algorithm. The serial runtimes are measured using the Goodrich-Pszona degeneracy ordering algorithm. For the parallel algorithms, we use superscripts to indicate the orientation that achieved the best running time. $^g$ refers to Goodrich-Pszona, $^b$ refers to Barenboim-Elkin, and $^k$ refers to $k$-core orientation. Note that degree orientation is never the fastest orientation. For the parallel algorithms, we list the runtimes obtained on a single thread ($T_1$), 36 cores without hyper-threading ($T_{36}$), and 36 cores with hyper-threading ($T_{36h}$). We also tested all implementations on friendster, but they all exceeded the time limit.

| | Serial Runtimes | | Parallel Kowalik Algorithm | | | | Parallel ESCAPE Algorithm | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Running times (s) | | | Speedup | Running times (s) | | | Speedup |
| | $T_E$ | $T_K$ | $T_1$ | $T_{36}$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36}$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| email | 0.36 | 0.026 | $0.027^b$ | $0.0027^g$ | $0.0029^b$ | 9.3 | $0.376^b$ | $0.017^g$ | $0.0177^g$ | 21.2 |
| dblp | 2.93 | 0.46 | $0.48^g$ | $0.046^b$ | $0.046^g$ | 10.4 | $3.24^g$ | $0.34^k$ | $0.277^k$ | 11.7 |
| youtube | 40.70 | 4.73 | $4.80^b$ | $1.73^g$ | $1.69^g$ | 2.8 | $43.94^g$ | $14.5^g$ | $9.96^g$ | 4.4 |
| lj | 2579.34 | 174.60 | $174.60^b$ | $29.0^g$ | $25.72^g$ | 6.8 | $2582.30^g$ | $426.38^g$ | $308.41^g$ | 8.4 |
| orkut | 38K | 2878.38 | $2867.07^b$ | $504.61^b$ | $487.4^b$ | 5.9 | TL | $8192.33^g$ | $6384.24^g$ | – |

**Parallel Five-cycle Counting.** Table 2 shows the best performance with the Kowalik and ESCAPE algorithms with 1 thread, 36 cores without hyper-threading, and 72 hyper-threads, without the work scheduling optimization. We see that the algorithms achieve decent speed-up without the work scheduling optimization. The parallel speed-up plateaus from 36 to 72 hyper-threads, especially for the parallelization of Kowalik's algorithm. The speed-up for the Kowalik algorithm is usually lower since, due to its degree ordering, it does not distribute work evenly across vertices, but rather concentrates the work on high-degree vertices. Our naive parallel algorithm groups a fixed number of vertices together regardless of whether they are high- or low-degree, resulting in unbalanced work distribution across workers.

From both the serial and parallel running times, we observe that the ESCAPE algorithm, with all of the same optimizations as the parallel Kowalik's algorithm, generally has about a 10x slowdown compared to Kowalik's algorithm. We attribute this difference to the discrepancy in the number of edge queries the two algorithms must perform. Since we store graphs in CSR format, each edge query requires a binary search. In Kowalik's algorithm, an edge query is performed for every $(v, w)$-pair, and it can be performed just before the for-loop with $x$, so there only needs to be $O(m\alpha)$ binary searches. In the ESCAPE algorithm, $(x, u)$ needs to be queried $O(m\alpha^2)$ times. Table 3 shows that the ESCAPE algorithm does significantly more binary searches than Kowalik's algorithm.

**Table 3** These are the number of binary searches each algorithm performed for each dataset, and the ratio of the number of binary searches in the ESCAPE algorithm to Kowalik's algorithm.

| | #binary searches | | |
|---|---|---|---|
| Dataset | Kowalik | ESCAPE | ESCAPE/Kowalik |
| email | $5.15 \times 10^5$ | $2.98 \times 10^7$ | 58 |
| dblp | $8.41 \times 10^6$ | $2.32 \times 10^8$ | 28 |
| youtube | $6.28 \times 10^7$ | $2.96 \times 10^9$ | 47 |
| lj | $1.39 \times 10^9$ | $1.72 \times 10^{11}$ | 124 |
| orkut | $1.25 \times 10^{10}$ | $3.44 \times 10^{12}$ | 273 |

**Table 4** Single-thread ($T_1$) and 36-core with hyper-threading ($T_{36h}$) running times (seconds) of the parallel Kowalik and ESCAPE algorithms with the work scheduling optimization, and their parallel speed-ups. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The superscripts indicate the orientation that achieved the best runtime. $^{\text{g}}$ refers to Goodrich-Pszona, $^{\text{b}}$ refers to Barenboim-Elkin, and $^{\circ}$ refers to degree orientation. In the appendix, we present the data for all orientations.

| | Parallel Kowalik Algorithm | | | Parallel ESCAPE Algorithm | | |
|---|---|---|---|---|---|---|
| | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| email | $0.0265^{\text{b}}$ | $0.00252^{\circ}$ | 10.5 | $0.357^{\text{b}}$ | $0.0165^{\text{b}}$ | 21.6 |
| dblp | $0.46^{\text{g}}$ | $0.0143^{\text{g}}$ | 32.2 | $3.07^{\text{b}}$ | $0.0866^{\text{g}}$ | 35.5 |
| youtube | $4.75^{\text{b}}$ | $0.338^{\text{b}}$ | 14.1 | $43.32^{\text{g}}$ | $1.42^{\text{g}}$ | 30.0 |
| lj | $171.92^{\text{b}}$ | $5.85^{\text{g}}$ | 29.4 | $2510.97^{\text{b}}$ | $58.75^{\text{g}}$ | 42.7 |
| orkut | $2858.18^{\text{b}}$ | $136.98^{\text{g}}$ | 20.9 | TL | $1269.1^{\text{b}}$ | – |
| friendster | TL | $8417.31^{\text{g}}$ | – | TL | TL | – |

Compared to the state-of-the-art serial five-cycle counting implementation provided in the ESCAPE package, without the work scheduling optimization, our parallel Kowalik implementation achieves a speed-up of 33.78–229.79x, and our parallel ESCAPE implementation achieves a speed-up of 5.73–23.16x.

**Work Scheduling Optimization.** We present the best running times of the parallel Kowalik and ESCAPE algorithms using work scheduling in Table 4. Compared to Table 2, we see that the work scheduling optimization is effective on both parallel algorithms. It allows five-cycle counting to be performed on the Friendster graph in under 2.5 hours using the parallel Kowalik algorithm. Figure 5 shows the relative running time of the parallel Kowalik algorithm with 72 hyper-threads with different arboricity orientation subroutines, including Goodrich-Pszona, Barenboim-Elkin, degree ordering, and $k$-core orientation, with and without the work scheduling optimization. The comparison shows that work scheduling significantly improves the running time and scaling of the parallel Kowalik algorithm.

Throughout our tests, we use the sum of neighbors' degrees as the estimator of the amount of work. Other work estimators were tested, including a simple out-degree count and the two-hop neighbor out-degree sum, but did not result in improved performance.

Compared to the state-of-the-art serial five-cycle counting implementation provided in the ESCAPE package, using the work scheduling optimization, our parallel Kowalik implementation achieves a speed-up of 162.70–818.12x, and our parallel ESCAPE implementation achieves a speed-up of 23.56–72.13x. Compared to our best serial baselines, our parallel Kowalik implementation achieves a speed-up of 10.5–32.2x.

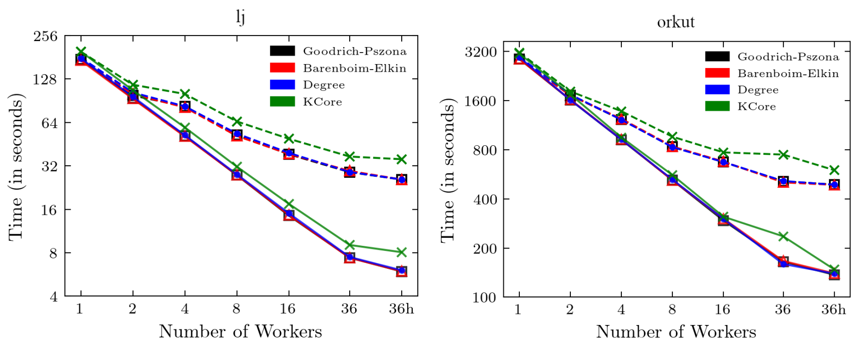


**Figure 5** Running time of the parallel Kowalik algorithm vs. number of threads. "36h" is 36 cores with hyper-threading. Dashed lines indicate that the work scheduling optimization is disabled and solid lines indicate that the work scheduling optimization is enabled. The lines for Goodrich-Pszona, Barenboim-Elkin, and degree ordering overlap each other for the most part.

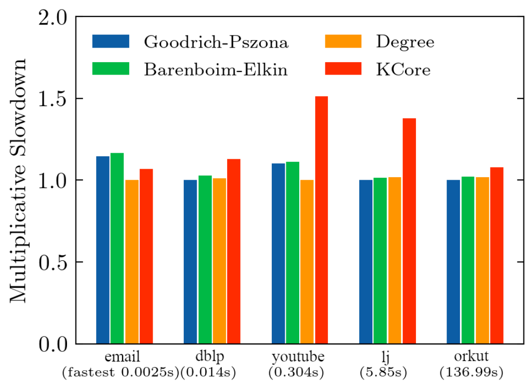


**Figure 6** Five-cycle counting times, excluding preprocessing steps like relabeling and orienting the graph, under different orientation schemes for each of the graphs, using 36 cores with hyper-threading.

**Graph Orientation.** Figure 6 compares the performance of our parallel Kowalik implementation using different orientation schemes. Goodrich-Pszona and Barenboim-Elkin have very similar performance. $k$-core performs slightly worse on all graphs except for the small email graph. From our experiments, degree ordering results in running times that are comparable to both Goodrich-Pszona and Barenboim-Elkin.

While Goodrich-Pszona, Barenboim-Elkin, and $k$-core produce arboricity orderings, we may want to use degree ordering as it is much more efficient to compute and can compensate for the potentially worse counting time. Figure 7 shows the proportion of time spent on preprocessing ($T_p$) versus counting ($T_c$) on different orientation methods on three of the graphs. As the graph size grows, the preprocessing time takes up a smaller fraction of the total running time and becomes negligible in the case of the orkut graph. However, for smaller graphs, degree orientation has a clear advantage, because it takes much less time to compute while allowing for similar performance in the counting step. $k$-core ordering does not perform well when considering the times for both preprocessing and counting.

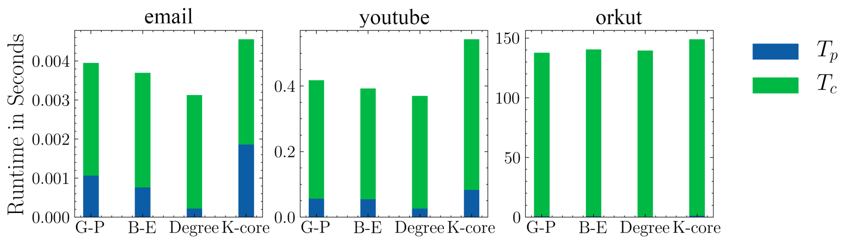


**Figure 7** Breakdown of time spent on preprocessing ($T_p$) vs. counting ($T_c$) for different orientation subroutines, using 36 cores with hyper-threading. G-P is Goodrich-Pszona; B-E is Barenboim-Elkin. For the orkut graph, the time spent on preprocessing is not visible.

## 6 Conclusion

We designed the first theoretically work-efficient parallel five-cycle counting algorithms with polylogarithmic span. On 36 cores, our implementations outperform the fastest existing serial implementation by up to 818x, and achieve self-relative speed-ups of 10–46x. Designing parallel algorithms for counting larger cycles is interesting for future work, although such algorithms are likely to require super-linear work, even for low-arboricity graphs [7].

## References

1 A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 434–443, 2014.

2 Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, Nick G. Duffield, and Theodore L. Willke. Graphlet decomposition: framework, algorithms, and applications. *Knowl. Inf. Syst.*, 50(3):689–722, 2017.

3 N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

4 Richard Anderson and Ernst W. Mayr. A P-complete problem and approximations to it. Technical report, Stanford University, 1984.

5 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using Nash-Williams decomposition. *Distributed Computing*, 22:363–379, 2010.

6 Vladimir Batagelj and Matjaž Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *Advances in Data Analysis and Classification*, 5(2):129–145, 2011.

7 Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Linear Time Subgraph Counting, Graph Degeneracy, and the Chasm at Size Six. In *Proceedings of the Innovations in Theoretical Computer Science Conference*, pages 38:1–38:20, 2020.

8 Jonathan W. Berry, Luke K. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. Why do simple algorithms for triangle enumeration work in the real world? In *Proceedings of the Conference on Innovations in Theoretical Computer Science*, page 225–234, 2014.

9 Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

10 M. A. Bhuiyan, M. Rahman, M. Rahman, and M. Al Hasan. GUISE: Uniform sampling of graphlets for large graph analysis. In *Proceedings of the IEEE International Conference on Data Mining*, pages 91–100, 2012.

11 Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, page 189–199, 2010.

12 Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

13 Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974.

14 Ronald S. Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–399, 2004.

15 Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, February 1985.

16 Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, page 393–404, 2018.

17 Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. The graph based benchmark suite (GBBS). In *Proceedings of the Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, 2020.

18 Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Beyond triangles: A distributed framework for estimating 3-profiles of large graphs. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 229–238, 2015.

19 Ethan R. Elenberg, Karthikeyan Shanmugam, Michael Borokhovich, and Alexandros G. Dimakis. Distributed estimation of graph 4-profiles. In *Proceedings of the International Conference on World Wide Web*, page 483–493, 2016.

20 Giorgio Fagiolo. Clustering in complex directed networks. *Physical Review E*, 76(2):026107, 2007.

21 Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 32(3):221–233, 2010.

22 J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the IEEE Symposium of Foundations of Computer Science*, pages 698–710, 1991.

23 Michael T. Goodrich and Paweł Pszona. External-memory network analysis algorithms for naturally sparse graphs. In *Proceedings of the European Symposium on Algorithms*, pages 664–676, 2011.

24 Tomaz Hocevar and Janez Demsar. A combinatorial approach to graphlet counting. *Bioinformatics*, pages 559–65, 2014.

25 Paul W. Holland and Samuel Leinhardt. A method for detecting structure in sociometric data. *American Journal of Sociology*, 76(3):492–513, 1970.

26 Joseph Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

27 Łukasz Kowalik. Short cycles in planar graphs. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 284–296, 2003.

28 Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010.

29 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2019.

30 David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3), July 1983.

31 Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the Web–analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015.

32 Crispin Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, s1-39(1):12–12, 1964.

**33** Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Inf. Process. Lett.*, 112(7), March 2012.

**34** Ali Pinar, C. Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the International Conference on World Wide Web*, page 1431–1440, 2017.

**35** Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

**36** M. Rahman, M. A. Bhuiyan, and M. Al Hasan. Graft: An efficient graphlet counting method for large graph analysis. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2466–2478, 2014.

**37** S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, June 1989.

**38** Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 2150–2159, 2018.

**39** Ahmet Erdem Sarıyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *Proceedings of the ACM International Conference on Web Search and Data Mining*, page 504–512, 2018.

**40** T. Schank. Algorithmic aspects of triangle-based network analysis. *PhD Thesis, Universitat Karlsruhe*, 2007.

**41** Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020. `arXiv:2002.10047`.

**42** Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In Bruce M. Maggs, editor, *Proceedings of the SIAM Symposium on Algorithmic Principles of Computer Systems*, pages 16–30, 2020.

**43** Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Network Analysis and Mining*, 1(2):75–81, April 2011.

**44** J. Wang, A. W. Fu, and J. Cheng. Rectangle counting in large bipartite graphs. In *Proceedings of the IEEE International Congress on Big Data*, pages 17–24, 2014.

**45** Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Vertex priority based butterfly counting for large-scale bipartite networks. *Proc. VLDB Endow.*, 12(10):1139–1152, 2019.

**46** Sebastian Wernicke and Florian Rasche. FANMOD: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

**47** Xiangzhou Xia. Efficient and scalable listing of four-vertex subgraphs. Master's thesis, Texas A&M University, 2016.

**48** R. Zhu, Z. Zou, and J. Li. Fast rectangle counting on massive networks. In *Proceedings of the IEEE International Conference on Data Mining*, pages 847–856, 2018.

## A Appendix

We present in Table 5 the running times of the parallelized Kowalik's algorithm and the ESCAPE algorithm using the work scheduling optimization for the different orientations described.

**Table 5** Single-thread ($T_1$) and 36-core with hyper-threading ($T_{36h}$) running times (seconds) of the parallel Kowalik and ESCAPE algorithms with the work scheduling optimization using all four orientations. All running times include both preprocessing (graph orienting) and five-cycle counting time. We stop each experiment after 5.5 hours, and "TL" indicates that the time limit was exceeded. The bold values mark the best serial and parallel runtimes for each of Kowalik and ESCAPE, out of the four orientations, which are used in Table 4.

**(a)** Goodrich-Pszona.

| | Kowalik | | | ESCAPE | | |
|---|---|---|---|---|---|---|
| | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| email | 0.0267 | 0.00289 | 9.3 | 0.396 | 0.0174 | 22.7 |
| dblp | **0.459** | **0.0143** | 32.2 | 3.17 | **0.0866** | 36.6 |
| youtube | 4.81 | 0.361 | 13.3 | **43.3** | **1.42** | 30.6 |
| lj | 174.40 | **5.85** | 29.8 | 2546.95 | **58.75** | 43.4 |
| orkut | 2867.78 | **136.98** | 20.9 | TL | 1552.70 | - |

**(b)** Barenboim-Elkin.

| | Kowalik | | | ESCAPE | | |
|---|---|---|---|---|---|---|
| | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| email | **0.0265** | 0.00294 | 9.0 | **0.357** | **0.0165** | 21.6 |
| dblp | 0.465 | 0.0147 | 31.6 | **3.07** | 0.0992 | 31.0 |
| youtube | **4.75** | **0.338** | 14.0 | 48.05 | 1.43 | 33.6 |
| lj | **171.92** | 5.95 | 28.9 | **2510.97** | 59.03 | 42.5 |
| orkut | **2858.18** | 139.87 | 20.4 | TL | **1269.07** | - |

**(c)** Degree.

| | Kowalik | | | ESCAPE | | |
|---|---|---|---|---|---|---|
| | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| email | 0.0276 | **0.00252** | 10.9 | 1.49 | 0.0403 | 37.0 |
| dblp | 0.472 | 0.0144 | 32.7 | 10.71 | 0.773 | 13.9 |
| youtube | 4.79 | 0.344 | 13.9 | 2177.52 | 59.61 | 36.5 |
| lj | 178.02 | 5.96 | 29.9 | 16651.40 | 417.00 | 39.9 |
| orkut | 2949.47 | 139.37 | 21.2 | TL | 16129.4 | - |

**(d)** K-Core.

| | Kowalik | | | ESCAPE | | |
|---|---|---|---|---|---|---|
| | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ | $T_1$ | $T_{36h}$ | $\frac{T_1}{T_{36h}}$ |
| | 0.0278 | 0.00304 | 9.2 | 0.675 | 0.0245 | 27.6 |
| | 0.473 | 0.0161 | 29.3 | 7.80 | 0.240 | 32.4 |
| | 5.84 | 0.531 | 11.0 | 922.19 | 23.62 | 39.0 |
| | 198.16 | 8.06 | 24.6 | 11151.50 | 309.71 | 36.0 |
| | 3100.79 | 147.83 | 21.0 | TL | 7113.44 | - |

# Fast and Robust Vectorized In-Place Sorting of Primitive Types

Mark Blacher ✉
Institute for Theoretical Computer Science, Friedrich Schiller Universität Jena, Germany

Joachim Giesen ✉
Institute for Theoretical Computer Science, Friedrich Schiller Universität Jena, Germany

Lars Kühne ✉
German Aerospace Center (DLR), Jena, Germany

**Abstract**

Modern CPUs provide *single instruction-multiple data* (SIMD) instructions. SIMD instructions process several elements of a primitive data type simultaneously in fixed-size vectors. Classical sorting algorithms are not directly expressible in SIMD instructions. Accelerating sorting algorithms with SIMD instruction is therefore a creative endeavor. A promising approach for sorting with SIMD instructions is to use sorting networks for small arrays and Quicksort for large arrays. In this paper we improve vectorization techniques for sorting networks and Quicksort. In particular, we show how to use the full capacity of vector registers in sorting networks and how to make vectorized Quicksort robust with respect to different key distributions. To demonstrate the performance of our techniques we implement an in-place hybrid sorting algorithm for the data type `int` with AVX2 intrinsics. Our implementation is at least 30% faster than state-of-the-art high-performance sorting alternatives.

**2012 ACM Subject Classification** Theory of computation → Sorting and searching

**Keywords and phrases** Quicksort, Sorting Networks, Vectorization, SIMD, AVX2

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.3

**Supplementary Material** *Software (Source Code)*: `https://github.com/simd-sorting/fast-and-robust` archived at `swh:1:dir:945e0841401092b83647202f46e8a60b084619f9`

**Funding** This work has been supported by the German Research Foundation (DFG) grant GI-711/5-1 within the Priority Program 1736 Algorithms for Big Data.

## 1 Introduction

Sorting is a fundamental problem in computer science, since sorting algorithms are part of numerous applications in both commercial and scientific environments. Among others, sorting plays an important role in combinatorial optimization [28], astrophysics [29], molecular dynamics [15], linguistics [39], genomics [33] and, weather forecasting [36]. Sorting is used in databases for building indices [30], in statistics software for estimating distributions [13], and in object recognition for computing convex hulls [2]. Sorting enables binary search, simplifies problems such as checking the uniqueness of elements in an array, or finding the closest pair of points in the plane [10]. Faster sorting implementations can thus reduce the computing time in a wide area of applications.

Some of the performance-critical applications mentioned above extensively sort arrays of numbers [36], that is, primitive types like `int` or `float`. High-performance libraries like Intel Performance Primitives (IPP) [21] or Boost.Sort [35] therefore offer fast sorting procedures for primitive types. But, these implementations do not take advantage of the available vector instruction sets, which are an integral part of modern CPUs. In computational domains such as linear algebra [4, 14], image processing [25], or cryptography [23] vector instructions are ubiquitous, however, vector instructions are rarely used in practice for sorting. One reason is that the vectorization of sorting algorithms is cumbersome, since existing sorting algorithms cannot be directly expressed in SIMD instructions. Another reason is that current vectorized sorting algorithms often cannot outperform an optimized radix sort. Furthermore, current vectorized Quicksort implementations [5, 16] are not robust to different key distributions.

The idea of using vector instructions for sorting is not new. The field of vectorized sorting has been explored since the 1970s, first on vector supercomputers with their specialized instruction sets and later on modern architectures (see our discussion of related work below). A large part of the work on vectorized sorting algorithms has been done on old hardware from today's point of view. Since then, SIMD instruction sets continued to evolve and vector registers became wider. Also, new vectorization techniques such as vectorized in-place partitioning in Quicksort [5] have been discovered. Due to these changes, a new look at vectorized sorting algorithms seems worthwhile.

**Related Work.** Vectorized sorting is a topic that has been extensively researched on vector supercomputers [31, 37, 38, 41]. However, the knowledge gained there cannot be transferred uncritically to modern hardware. For example, Zagha and Blelloch [41] vectorize Radix Sort on the vector supercomputer CRAY Y-MP. Compared to a vectorized hybrid algorithm of Quicksort and Odd-Even Transposition Sort [31], they achieve speedup factors of three to five with their Radix Sort. The speedup is partly due to the fact that the CRAY Y-MP has no hardware prefetcher, which means accessing data elements in RAM in random order takes the same time as accessing them sequentially.

Studies on vectorized sorting, which take into account modern hardware, that is, vector instructions, instruction-level parallelism, CPU caches, and multicore parallelism, focus primarily on the vectorization of Mergesort [8, 19, 20]. Essentially, a vectorized Mergesort is a hybrid of merging networks and the usual Mergesort or some multiway variant of it. Despite efficient utilization of the underlying hardware, single-threaded vectorized Mergesort variants on mainstream CPUs do not achieve the speed of a hardware optimized Radix Sort such as Intel's platform-aware Radix Sort (IPP Radix Sort) [21]. For special hardware like the Xeon Phi 7210 processor, which features AVX-512 instructions and high bandwidth memory, a fast vectorized Mergesort variant exists, but unfortunately without publicly available source code for testing [40].

Gueron and Krasnov [16] show that Quicksort can be vectorized on modern hardware using the AVX and AVX2 instruction sets. Their partitioning function requires $\mathcal{O}(n)$ additional memory. The pivot value is always the last element of the partition. After the vectorized compare operation with the pivot value, the elements in the vector register are shuffled using shuffle masks that are stored in a lookup table. Gueron and Krasnov miss the opportunity to sort even small arrays with vector instructions. For sub-partitions with fewer than 32 elements, they use insertion sort, similarly to the C++ Standard Template Library (STL). The running times of their vectorized Quicksort are higher than those of IPP radix sort when sorting randomly distributed 32-bit integers.

Using AVX-512 instructions, Bramas [5] designs a vectorized hybrid algorithm based on Quicksort and Bitonic Sort that outperforms IPP radix sort on random input data by a factor of 1.4. Using the new compress instructions in AVX-512, Bramas implements Quicksort's partitioning function without the use of pre-computed permutation masks. Compress instructions arrange elements contiguously in a vector according to a bitmask. Compress instructions are not new per se. Both Stone [38] and Levin [31] have already used compress instructions on vector supercomputers to implement Quicksort's partitioning function. The true contribution of Bramas is that he implements the partitioning function without additional memory. He does this by buffering the two SIMD vectors at the outer ends of the array. The pivot value for partitioning the array is determined with the median-of-three strategy. For sorting small sub-partitions with less than or equal to 256 integers Bramas uses a branch-free vectorized Bitonic Sort.

**Practically Important Features for Sorting.** Among the sorting implementations mentioned above, the fastest are sort AVX-512 [5] and Intel's IPP radix sort [21]. In addition to speed, the following features are often equally important in practice: low memory usage, robustness against different key distributions, portability, and efficient parallelization. Table 1 summarizes the features of our and other sorting implementations for primitive types. In this paper we consider only the single-threaded case, since our goal is to show that *pure* vectorized sorting is indeed the fastest available option for sorting primitive types. Vectorization and parallelization are orthogonal concepts. Parallelization would make the results highly dependent on the parallelization technique. The efficient combination of our vectorization techniques with parallelization is a topic that we plan to explore in future work.

**Table 1** Features of various high performance sorting implementations for primitive types. The state-of-the-art general purpose sorting algorithm IPS$^4$o [1] is included here for comparison, although it is not optimized for sorting primitive types. We use the term in-place if the sorting algorithm requires no more than $\mathcal{O}(\log n)$ additional non-constant memory on average.

| | **this paper** | sort AVX-512 [5] | IPP radix sort [21] | IPS$^4$o [1] |
|---|---|---|---|---|
| in-place | ✓ | ✓ | ✗ | ✓ |
| robust (distributions) | ✓ | ✗ | ✓ | ✓ |
| robust (duplicate keys) | ✓ | ✗ | ✓ | ✓ |
| portable ($\leq$ AVX2) | ✓ | ✗ | ✓ | ✓[a] |
| fast ($n \leq 1000$) | ✓ | ✓ | ✗ | ✗ |
| fast ($n > 1000$) | ✓ | ✓ | ✓ | ✗ |
| parallelized efficiently | ✗ | ✗ | ✗ | ✓ |

[a] Code does not compile on Windows.

**Our Contributions.** The main contributions of this paper are vectorization techniques for sorting networks and Quicksort that allow to design faster, more robust and memory efficient sorting algorithms for primitive types. In particular, we introduce the following techniques:

- For sorting small to medium-sized arrays we show how to use the full capacity of vector registers in sorting networks.
- To implement the vectorized partitioning function of Quicksort in-place with AVX2 instructions, we combine the lookup table strategy of Gueron and Krasnov [16] with the in-place partitioning of Bramas [5].
- To make vectorized Quicksort robust to different key distributions, we design an alternative pivot selection strategy that is used for unbalanced sub-arrays during partitioning. With this strategy, which is related to Radix Exchange Sort [7, 11], the worst case running time of our Quicksort is $\mathcal{O}(kn)$, where k is the number of bits in the primitive type.

Based on these techniques we implement a sorting algorithm with AVX2 vector instructions that outperforms general purpose and competing high-performance sorting implementations for primitive types. The source code of our implementation is available at `https://github.com/simd-sorting/fast-and-robust`. Additionaly, we provide an efficiently vectorized implementation of Quickselect [36], which uses the same vectorization techniques to find the $k$th smallest element in an unordered array.

## 2 Preliminaries

**Vector Instructions.** Vector instructions (SIMD instructions) in the x86 instruction set exist since 1997 [6]. The register width was steadily increased since then. Larger registers allow more elements of a primitive type to be processed simultaneously. Starting with 64 bits in MMX (Multi Media Extension), the register width increased to 128 bits for SSE (Streaming SIMD Extensions) and again doubled to 256 bits in AVX (Advanced Vector Extensions). Current CPUs for high-performance computing have 512-bit registers and support the AVX-512 instruction set [22]. Modern mainstream CPUs, however, support at most AVX2. Like its predecessor AVX, AVX2 uses vectors of up to 256 bits, but has a larger instruction set. Since we want our implementation to run on a large variety of available CPUs we use AVX2 and not AVX-512 to implement our ideas. Instead of writing the vectorized part of the algorithms in assembly language, we use intrinsic functions. Intrinsic functions have the advantage that explicit assignments of registers are omitted. Intrinsic functions are also more portable between different compilers and operating systems.

**Latency and Throughput.** Besides the vector width, the two metrics latency and throughput provide further information about the performance of an instruction. Latency is the number of cycles that an instruction takes until its result is available for another instruction. Throughput, on the other hand, indicates how many cycles it takes on average until an identical independent instruction can begin its execution [12]. If the throughput of an instruction is smaller than the latency, executing several identical independent instructions in sequence can lower the accumulated latency of the computation, because instructions can begin before the results of previous instructions are available. We exploit this heavily by simulating larger vectors to increase instruction-level parallelism of our implementation.

## 3 The Algorithm

In this section we describe the design and implementation of our algorithm for sorting arrays of data type `int` (32-bit signed integer). Most state-of-the-art sorting implementations combine several sorting algorithms for achieving good performance on different array sizes and key distributions. Here, we also take this approach and combine vectorized Quicksort with vectorized sorting networks into a fast and robust in-place sorting algorithm. The overall design of our algorithm is shown in the simplified C++ Listing 1.

The remainder of this section is organized as follows: In Subsection 3.1 we discuss our techniques to vectorize sorting networks for sorting small and medium-sized arrays, respectively. In Subsections 3.2 and 3.3 we describe our combined pivot strategy that makes Quicksort robust with respect to different key distributions and also makes it robust with respect to duplicate keys. As will be discussed in Subsection 3.4, our algorithm is in-place in the sense that the only non-constant memory overhead comes from the recursive function calls. This Section finishes with a brief worst and average running time analysis in Subsection 3.5.

### 3.1 Sorting Networks

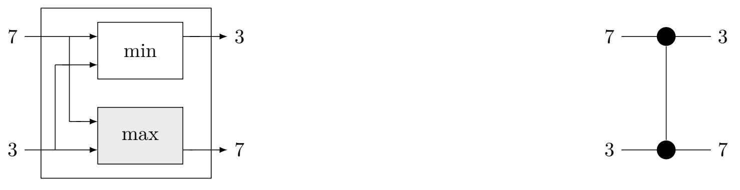
The building blocks of sorting networks are **Compare-and-exchange (COEX) modules**. A COEX module consists of two inputs and two outputs. Each input receives a value. The two values are sorted in the module and transferred to the two outputs. In our representation of a COEX module, the upper output track contains the smaller and the lower output track the larger value. See Figure 1 for different options of visualizing a COEX module.

**Listing 1** Our vectorized Quicksort algorithm partitions arrays with vector instructions until the sub-arrays become small enough ($\leq$ 512 elements) and then switches to vectorized sorting networks. For achieving higher performance, two different techniques are used to vectorize sorting networks (not shown in Listing 1). One for small arrays ($n < 128$) and one for medium-sized arrays ($128 \leq n \leq 512$). To detect sub-arrays in which all elements are equal, the *smallest* and *largest* values in the array are computed during the partitioning phase. We combine two pivot strategies to make our Quicksort robust with respect to various distributions. After partitioning, we do not set the pivot element to its correct position in the sorted array. This allows us to use values as pivots that are not contained in the array.

```
void qs_core(int *arr, int left, int right, bool choose_avg = false, int avg = 0) {
  if (right - left <= 512) { // sorting networks for array sizes <= 512
    sort_with_sorting_networks(arr + left, right - left + 1);
    return;
  }
  int pivot = choose_avg ? avg : get_pivot_median_medians(arr, left, right);
  int smallest = INT32_MAX; // smallest value in the array after partitioning
  int largest = INT32_MIN;  // largest value in the array after partitioning
  int bound = partition(arr, left, right + 1, pivot, &smallest, &largest);
  // the ratio of the size of the smaller partition to the size of the array
  double ratio = min(right - (bound - 1), bound - left) / double(right - left + 1);
  if (ratio < 0.2) // if unbalanced sub-arrays, change pivot strategy
    choose_avg = !choose_avg;
  if (pivot != smallest) // if values not identical in left sub-array, recurse
    qs_core(arr, left, bound - 1, choose_avg, average(smallest, pivot));
  if (pivot + 1 != largest) // if values not identical in right sub-array, recurse
    qs_core(arr, bound, right, choose_avg, average(largest, pivot));
}
```

In a sorting network, the COEX modules are combined in such a way that they always sort a fixed-length sequence of values. The amount of modules and their execution order are fixed for a network and do not depend on the input values. Sorting networks are therefore data-oblivious algorithms. A sorting network for eight elements is shown in Figure 2. The number of parallel steps and the number of COEX modules are the two key parameters that are used to optimize sorting networks. A sorting network with the minimum number of COEX modules does not necessarily have the fewest parallel steps and vice versa [9].

There are **regular** [3, 34] and **irregular sorting networks** [9, 27]. For a regular sorting network, an algorithmic rule exists that generates the network and can also generate larger networks of the same type. For irregular sorting networks it is not known how the construction of these networks can be generalized to create larger networks [26].



**(a)** Detailed. **(b)** Simplified.

**Figure 1** Compare-and-exchange module visualizations. To represent modules in sorting networks we use the simplified visualization depicted in (b).

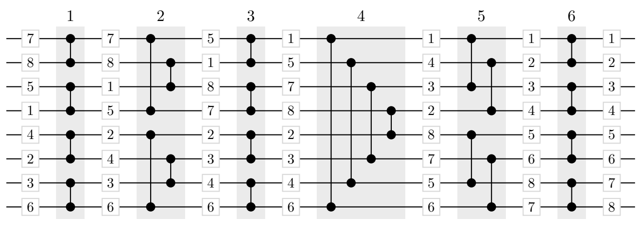


**Figure 2** Bitonic sorting network for eight elements (24 modules, six parallel steps). The input comes from the left and runs to the right through the network. The numbers above the gray boxes index the **parallel steps**. Each parallel step is characterized by the fact that there are no dependencies between individual modules. All COEX modules within a gray box can thus be executed simultaneously.

In the following two paragraphs we show how to efficiently vectorize sorting networks for sorting small ($n < 128$) and medium-sized ($128 \leq n \leq 512$) arrays. We define $n = 128$ as the boundary between small and medium-sized arrays, since we use different types of sorting networks. The technique we use for sorting small arrays is especially efficient for Bitonic Sort. Therefore, we apply it to small Bitonic sorting networks. In contrast, the technique we use for sorting medium-sized arrays can efficiently vectorize irregular sorting networks that have a minimum amount of COEX modules.

**Sorting Small Arrays With Sorting Networks.** To efficiently vectorize small Bitonic sorting networks, it is important to distinguish between modules and nodes. A COEX module schematically consists of two nodes and a vertical connecting line between them. (see Figure 1b). The elements passing through the two nodes are compared within a module. Here, we represent a COEX module as a tuple of two comma separated numbers. For example, $(1, 2)$ stands for a module in which the element at the first node is compared to the second. The numbers in brackets represent the nodes to be compared and not the values to be sorted. Nodes correspond only to the positions of elements in a sorting network.

The aim of vectorizing sorting networks is to execute as many modules as possible at once. To perform a vectorized COEX operation, the two nodes of a module must be at the same index in two different vectors. By computing the pairwise minimum and maximum between these two vectors, the vectorized COEX operation is executed. Before we perform a vectorized COEX operation, we use shuffle instructions to place the two nodes of a module under the same index in the two vectors. We use two different shuffle instructions. The first shuffles the elements only within one vector. The second called shuffle*, receives two vectors as input and mixes them according to a mask to create a new vector.

In our approach nodes are considered only virtually, while the values to be sorted are actually stored in the vector registers. When a COEX operation is executed, two types of exchanges are possible. On the one hand, values can be exchanged between the vector registers, on the other hand, nodes within a module can also be swapped. Since the nodes are virtual, the swaps of the nodes are also only virtual, and thus do not consume CPU cycles. Here, in order to simplify the presentation of our approach, the capacity of vectors is limited to four elements. Figure 3 shows our technique for sorting eight elements with the bitonic sorting network from Figure 2.

Our technique distributes nodes within modules to different vectors and thus uses the full capacity of vector registers. Bramas [5] uses, unlike our approach, only half the capacity of a vector register because the two nodes of a module are held within one vector. Furthermore, Bramas uses only permutation instructions for implementing sorting networks. We use shuffle instructions with lower latency wherever possible.

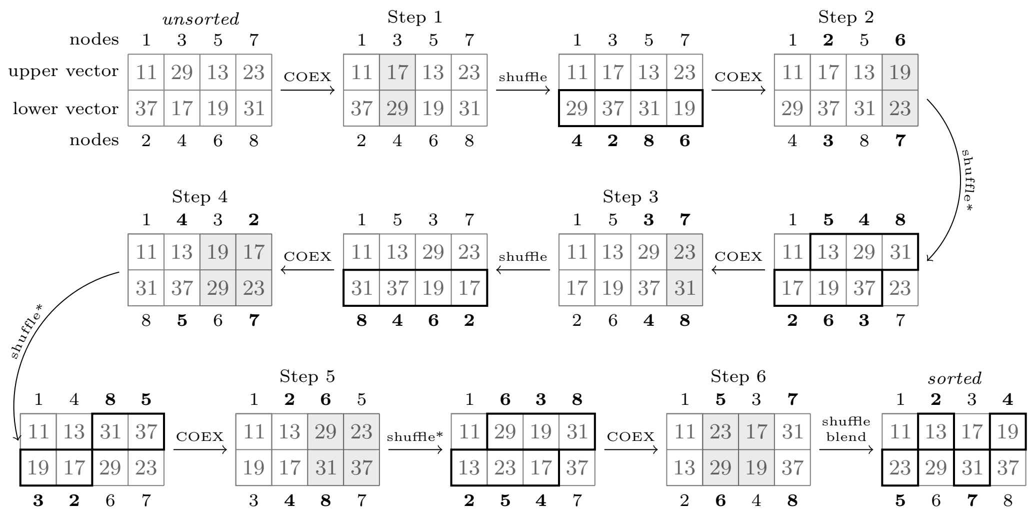


**Figure 3** Bitonic Sort of 8 elements with two vectors. Each vector has a capacity of 4 elements. We call the six parallel steps in which the COEX operations are executed Step 1, . . . , Step 6. In the first parallel step, Bitonic Sort has the four COEX modules $\{(1, 2), (3, 4), (5, 6), (7, 8)\}$. The association of the nodes and indices in the vectors at the beginning of the sorting operation is freely selectable. We start under the assumption that the upper vector represents the Nodes $1, 3, 5, 7$ and the lower vector represents the Nodes $2, 4, 6, 8$. This initial distribution of nodes prevents unnecessary shuffles of elements before executing the first parallel step because the nodes within modules are at the same index in the two vectors. In the vectorized COEX operation of the first parallel step only the values 29 and 17 are exchanged, which is indicated by the gray box in Step 1. The second parallel step requires the modules $\{(1, 4), (2, 3), (5, 8), (6, 7)\}$. Before executing Step 2, the lower vector is shuffled so that the nodes to be compared face each other. The COEX operation in Step 2 exchanges the values 23 and 19. The nodes within two modules must also be swapped (3 with 2, and 7 with 6). These swaps are necessary, because the minimum of two nodes after executing the vectorized COEX operation is always in the upper vector and the minimum is assigned to the node with the smaller index. The next four parallel steps are performed in a similar way. After executing Step 6, the upper vector always contains the elements at the Nodes 1, 5, 3 and 7, and, the lower vector contains the elements at the Nodes 2, 6, 4 and 8, respectively. The elements are sorted, but must be rearranged according to the indices of the nodes. After rearranging the elements with two shuffle and two blend instructions, the eight elements are properly sorted.

**Sorting Medium-Sized Arrays With Sorting Networks.** For sorting medium-sized arrays ($128 \leq n \leq 512$) we interpret the array as a matrix in row-major order, where the number of columns corresponds to the number of elements that can be placed in a vector. In an AVX2 vector there is space for eight values of data type `int`. An array with 128 values of datatype `int` thus corresponds to a matrix with eight columns and 16 rows.

To sort column-wise, the vectorized COEX operation is sufficient. Only pairwise minima and maxima between the vectors are computed, and no permutations or shuffles of the elements within the vectors are required for sorting the columns of a matrix. During a vectorized COEX operation, the same COEX module is executed in all columns. The number of vectorized COEX operations therefore depends on the number of modules in a sorting network. The advantage of sorting the elements inside columns is that even sorting networks with an irregular distribution of modules can be used, since each vectorized COEX operation uses only one module of the sorting network.

Sorting networks with a minimum number of COEX modules are particularly suitable for sorting the values in columns. To sort eight columns each containing 16 values of data type `int`, we use Green's irregular sorting network [27], which consists of only 60 COEX

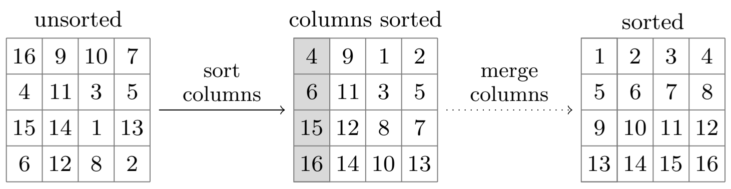


**Figure 4** Sorting medium-sized arrays. We interpret the array as a matrix in row-major order. For sorting columns we use sorting networks with a minimum number of COEX modules. To merge the sorted columns, we apply Bitonic Merge directly, without transposing the columns first.

modules. To fully sort the 128 values, the eight sorted columns must be merged. Instead of transposing the $16 \times 8$ matrix and merging the sorted rows with a Bitonic Merge network, we apply Bitonic Merge directly to the sorted columns of the matrix. The technique for merging sorted columns is similar to the technique we use for sorting small arrays. Before we execute a COEX operation, we shuffle or permute elements of vectors such that the nodes of each module are placed in different vectors at the same index. Figure 4 summarizes our approach for sorting medium-sized arrays with vector instructions.

## 3.2 Pivot Selection

To make our vectorized Quicksort robust with respect to various distributions, we combine two different pivot selection strategies and switch between them when one strategy leads to unbalanced sub-arrays (see Listing 1).

**First Strategy.** The first strategy determines the pivot with a heuristic similar to the median of medians. Without the necessity to place the pivot to its final position in the array, determining the median of medians becomes faster since the index of the pivot is not needed. With AVX2 vector instructions we can compute eight medians simultaneously. To compute eight times the median-of-nine, we gather 72 random elements from the array to be partitioned and store them in nine vectors. The 72 random indices for the nine gather instructions are computed with the random number generator xoroshiro128+ [26]. We implement xoroshiro128+ with vector instructions to speed up the computation of random numbers. The nine vectors with its 72 elements are interpreted as a $9 \times 8$ matrix in row-major order. We compute the medians column-wise with a median network [32], which we implement with min and max vector instructions. After applying a median network to the columns of the matrix, the fifth vector contains all eight medians. We sort the eight medians with vector instructions and choose as pivot the average of the fourth and fifth largest medians. If the resulting average is not an integer, we round down to the next smaller integer.

**Second Strategy.** If the vectorized median of medians heuristic leads to unbalanced sub-arrays, we switch to the second pivot strategy, where we choose the average of the *smallest* value and the *current pivot* as the pivot in the left sub-array, and, in the right sub-array, the average of the *largest* value and the *current pivot* as the pivot. The quadratic worst case of Quicksort is avoided, since choosing the average of the lower and upper bounds of the sub-array as pivot guarantees that the range of possible values is halved with each recursive call. The difference to Radix Exchange Sort [7, 11] in our second pivot strategy is that instead of bits, actual values are used to halve the range of possible values at each recursive call. If the second pivot strategy leads to unbalanced sub-arrays, the first pivot selection strategy is used in the next recursive call.

To realize the second pivot strategy, we always compute the *smallest* and the *largest* values in the array with min and max vector instructions when partitioning the array. We take advantage of instruction-level parallelism of modern processors to hide the execution time of the minimum and maximum computations. The latency of min and max vector instructions is only one cycle and the throughput is half a cycle [24], which means that their execution time can almost be hidden if they are called while waiting for the results of high latency instructions.

## 3.3 Many Duplicate Keys

The knowledge about the *smallest* and the *largest* values in an array also allows us to detect sub-arrays where all values are equal. If the *smallest* value in an array and the *current pivot* are identical, no further partitioning of the left sub-array is required, or, if the *largest* value and the *current pivot* + 1 in an array are identical, no further partitioning of the right sub-array is necessary. Applying these checks before partitioning sub-arrays makes our vectorized Quicksort robust against arrays with many duplicate keys.

## 3.4 In-place Partitioning

We start partitioning the array without vector instructions until the number of unpartitioned elements corresponds to a multiple of the number of elements in a vector. Next, we cache two vectors with unpartitioned elements from the outer ends of the array. This creates space to perform the partitioning in-place with Bramas' [5] technique. To partition a single vector we use, similar to Gueron and Krasnov [16], permutation masks that are stored in a lookup table. Only one permutation instruction is sufficient for partitioning a vector, since we fill, like Bramas, the array from left to right with values smaller than or equal to the pivot, and, from right to left with values greater than the pivot. After storing a partitioned vector on the left and right side of the array we choose the next vector to partition from the side of the array with fewer already partitioned elements. Overwriting of not yet partitioned values of the array is impossible, because there is space for storing a partitioned vector on each side of the array due to the two initially cached vectors. This space is only used up as soon as the vectors cached at the beginning are partitioned and stored in the array. See Appendix A for a worked example of our vectorized in-place partitioning algorithm.

To reduce the total latency of partitioning big arrays and thus increase instruction-level parallelism, we load 64 elements into eight vector registers, partition the eight vectors individually and save them back to the array. In order to realize the partitioning of big arrays in-place we cache 16 vectors instead of two, at the beginning of the partitioning.

## 3.5 Running Time Analysis

The running time of our vectorized sorting algorithm is determined by the two employed pivot strategies, namely, the (vectorized) median of the medians strategy and the strategy that is inspired by Radix Exchange Sort. While we can control the worst case behaviour of our vectorized sorting algorithm with the latter strategy, the median of medians strategy allows to control the average case behaviour. In particular, we can give the following guarantees.

▶ **Lemma 1.** *The worst case running time of our vectorized sorting algorithm is $\mathcal{O}(kn)$ where $n$ is the number of keys and $k$ is the number of bits per key.*

**Proof.** (Sketch) The claim follows since we always switch from the median of the medians pivot strategy to the second pivot strategy whenever the first strategy leads to unbalanced sub-arrays. The second pivot strategy uses the average of the smallest and the largest values in the array as pivot element. This is similar to Radix Exchange Sort that uses the most significant bit to subdivide the keys in an array. It follows that the worst case complexity of our algorithm when using the second pivot strategy is the same as for Radix Exchange Sort, namely $\mathcal{O}(kn)$. Hence, our algorithm runs in worst case time $\mathcal{O}(kn)$ if $k \geq \log n$. If $k < \log n$, then the array needs to contain duplicate keys that our algorithm detects such that the worst case running time still is in $\mathcal{O}(kn)$. ◀

▶ **Lemma 2.** *The average case running time of our vectorized sorting algorithm is $\mathcal{O}(n \log n)$, where the average is with respect to the uniform distribution over the set of permutations of $n$ different keys.*

**Proof.** (Sketch) This follows immediately from the analogous results for Quicksort with median of the medians pivot strategy. Remember that we only switch to the second pivot strategy when the median of medians strategy leads to unbalanced sub-arrays. Hence, switching the pivot strategy cannot degrade the asymptotic average running time. ◀

For putting our running time analysis into perspective, note that our vectorized sorting algorithm is not a comparison based algorithm, since we use an arithmetic operation in the pivot strategy that is inspired by Radix Exchange Sort.

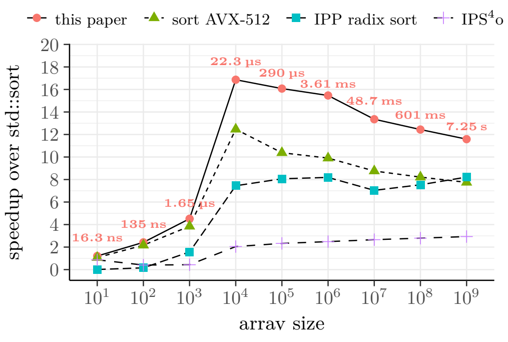
## 4 Experiments

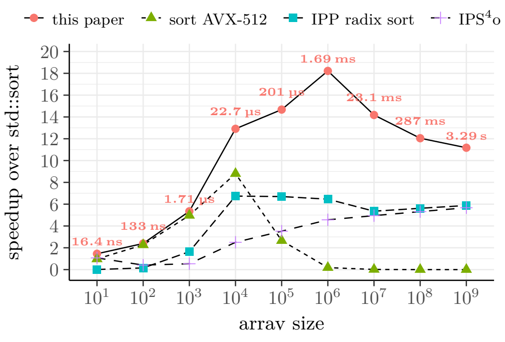
**Single-Threaded Performance.** We compare the single-threaded performance of our implementation with its high-speed competitors, sort AVX-512 [5], Intel's platform-aware radix sort (IPP radix sort) [21], and IPS$^4$o [1]. IPP radix sort is generally accepted as a benchmark for high-performance sorting of primitive data types [5, 16, 40]. IPS$^4$o is one of the fastest general purpose sorting algorithms. We report the performance of each algorithm in terms of speedup factors over std::sort from the C++ STL for different array sizes and distributions of integers. For computing a single speedup factor we divide the running time of std::sort by the running time of the respective algorithm. In our measurements we consider array sizes in the range of $10^1$ to $10^9$, whereby the next larger array size contains ten times more elements than the previous one. To evaluate the speed and robustness of our sorting algorithm, we fill the arrays with integers according to the following four distributions:

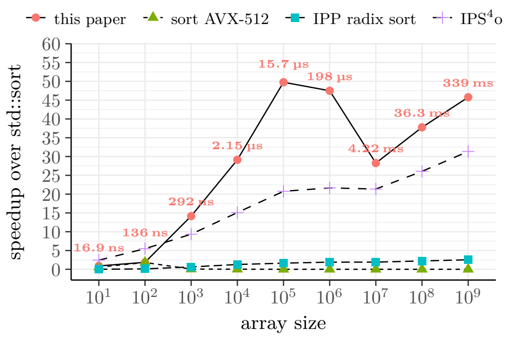
**(a)** *Uniform.* Random integers in the range $[-2^{31}, 2^{31} - 1]$.
**(b)** *Gaussian.* Normally distributed random numbers with expected value $\mu = 0$ and standard deviation $\sigma = 100$. We round the numbers to the nearest integer, resulting in many duplicate values in large arrays.
**(c)** *Zero.* Every value is set to a constant: an input distribution with zero entropy [17].
**(d)** *Almost Sorted.* Array of size $n$ with $\lfloor \frac{1}{2} \cdot 2^{\log_{10} n} \rfloor$ misplaced numbers. For example, an array of the size $10^6$ that contains 32 integers in wrong positions.

For our experiments, we use a machine with an Intel i9-10980XE 18-core processor running Ubuntu 20.04.1 LTS with 128 GB of RAM. Each core has a base frequency of 3.0 GHz and a max turbo frequency of 4.6 GHz and supports the AVX-512 vector instruction set. We compile our experiments with the Intel C++ compiler version 19.1.2.254 for 64 bit using the optimization flags `-Ofast` and `-march=native`.

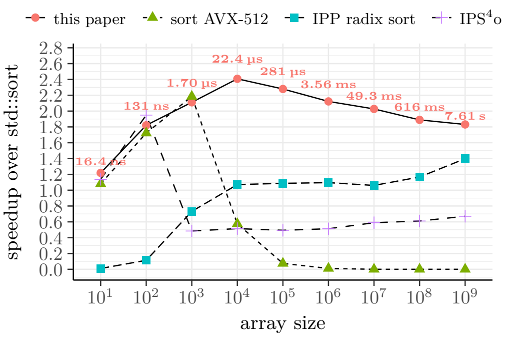
Figure 5 shows how the speedup factors of our implementation over std::sort compared to speedup factors of state-of-the-art high-speed sorting algorithms. Starting from an array length of $10^4$ our implementation is always significantly faster for the considered distributions of integers. We are on par with sort AVX-512, when sorting small arrays. Both our implementation and sort AVX-512 use vectorized sorting networks for small arrays, but our implementation only uses the more portable AVX2 vector instructions, but not AVX-512. In the AVX-512 instruction set the vectors are twice as wide than in AVX2, and in AVX-512 there are more suitable instructions for sorting available. Furthermore, Figure 5b and Figure 5c show the in general poor performance of Sort AVX-512, when sorting arrays with many duplicate keys, while our vectorized Quicksort implementation can handle this case efficiently. It also becomes apparent that a general purpose sorting algorithm like IPS$^4$o cannot keep up with the performance of an efficient vectorized sorting algorithm when sorting primitive data types, at least for the single-threaded case. For the multi-threaded case, the results may not be as clear-cut, since memory bandwidth becomes the ultimate performance limit for sorting algorithms.







**(c)** *Zero* (all values are equal).



**(d)** *Almost Sorted* ($\lfloor \frac{1}{2} \cdot 2^{\log_{10} n} \rfloor$ misplaced numbers).

**Figure 5** Speedup factors of our implementation, sort AVX-512 [5], IPP radix sort [21], and, IPS$^4$o [1] over std::sort for four different distributions of 32-bit signed integers. In addition, the graphs also contain the absolute running times of our implementation.

**Performance Indicators.** For finding out the reasons why our implementation performs better than state-of-the-art high-speed sorting algorithms, we look at performance indicators such as cycles required per sorted integer, cache misses, and instructions per cycle of the various implementations. Table 2 contains performance indicators for the evaluated sorting algorithms. Additionally, we also show performance indicators for std::sort. Each algorithm sorts an array of $10^9$ random integers. Our implementation needs on average 73 instructions per integer, while IPP radix sort requires only about 44 instructions. Also sort AVX-512 requires on average fewer instructions per integer ($\approx 46$) than our implementation. But both IPP radix sort and sort AVX-512 need more cycles to execute these fewer instructions than our implementation. This means the CPU utilization (instruction-level parallelism) of our implementation is higher than that of IPP radix sort or sort AVX-512. At the same wall-clock time, our implementation sorts more than 30% more integers than IPP radix sort or sort AVX-512. In addition, it should be noted that our implementation is in-place, while IPP radix sort uses $\mathcal{O}(n)$ extra memory.

**Table 2** Performance indicators per sorted integer. Each algorithm sorts an array of length $10^9$ populated with random integers from the range $[-2^{31}, 2^{31}-1]$. For example, sorting an integer with std::sort, on average, takes 377.14 CPU cycles and requires 226.25 instructions. The number of instructions per cycle (IPC) is defined as the quotient of the number of instructions and the number of cycles. A high IPC indicates efficient CPU utilization in terms of instruction-level parallelism. GHz denotes the average clock speed while sorting. When the integer is not in the cache, a cache miss occurs. The average number of cache misses per sorted integer is shown for the first level cache (L1) and the last level cache (LLC). The performance indicator branch-misses denotes the average number of mispredictions of the CPU branch-predictor per sorted integer.

| | cycles | instructions | L1-misses | LLC-misses | branch-misses | IPC | GHz |
|---|---|---|---|---|---|---|---|
| std::sort | 377.14 | 226.25 | 1.41 | 0.75 | 13.04 | 0.60 | 4.76 |
| **this paper** | 34.44 | 72.68 | 1.25 | 0.59 | 0.06 | **2.11** | 4.76 |
| sort AVX-512 | 43.32 | 46.44 | 1.42 | 0.71 | 0.34 | 1.07 | 3.97 |
| IPP radix sort | 48.83 | 43.68 | 5.53 | 0.46 | 0.00 | 0.89 | 4.71 |
| IPS$^4$o | 137.96 | 273.83 | 1.78 | 0.28 | 1.46 | 1.98 | 4.76 |

## 5 Conclusions

In this paper we presented a highly efficient single-threaded in-place sorting algorithm that we implemented for the data type `int` with AVX2 intrinsics in C++. To make our implementation fast and robust, we improved vectorization techniques for sorting networks and Quicksort. In particular, we showed how to utilize the full capacity of vector registers for executing the modules of sorting networks and developed a pivot selection technique to efficiently sort arrays of different key distributions. Furthermore, we presented a vectorized in-place partitioning technique for vectorized Quicksort that has a high degree of instruction-level parallelism. With our implementation we achieve a speedup of at least 30% over state-of-the-art high-performance sorting algorithms. The worst case running time of our sorting algorithm is $\mathcal{O}(kn)$, where k is the number of bits in the primitive type. Our implementation is easily extensible to other primitive types like `unsigned int` or `float`, and can also be adapted to sort primitive 64-bit types. Actually, there is no need to consider the sorting of floating-point numbers separately, since the IEEE format was designed in such a way that with some additional bit-twiddling floating-point numbers can be sorted with integer sorting routines [6, 18]. On future processors with larger vector registers and more diverse vector instructions, the speed difference in favor of vectorized sorting algorithms for primitive types is likely to increase further. Hence, we see a vectorized in-place hybrid algorithm based on sorting networks and Quicksort as a possible replacement for current sorting implementations in high-performance libraries.

## References

1 Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017)*, volume 87 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ESA.2017.9.

2 Jakob Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Guide to Computational Geometry Processing*. Springer, 2012. doi:10.1007/978-1-4471-4075-7.

3 Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.

4 BLAS. Basic linear algebra subprograms. URL: http://www.netlib.org/blas/.

5 Berenger Bramas. A novel hybrid quicksort algorithm vectorized using avx-512 on intel skylake. *International Journal of Advanced Computer Science and Applications*, 8(10), 2017. doi:10.14569/IJACSA.2017.081044.

6 Randal Bryant. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2016.

7 Shi-Kuo Chang. *Data Structures and Algorithms*. World Scientific, 2003.

8 Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008. doi:10.14778/1454159.1454171.

9 Michael Codish, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. Sorting networks: To the end and back again. *Journal of Computer and System Sciences*, 2016. doi:10.1016/j.jcss.2016.04.004.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3rd edition, 2009.

11 Amjad M. Daoud, Hussein Abdel-jaber, and Jafar Ababneh. Efficient non-quadratic quick sort (nqquicksort). In Ezendu Ariwa and Eyas El-Qawasmeh, editors, *Digital Enterprise and Information Systems*, pages 667–675. Springer, 2011.

12 Developer Zone. Intel® Software Developer Zone, 2008. URL: https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput.

13 Jay Devore and Kenneth Berk. *Modern Mathematical Statistics With Applications*. Springer, 2012.

14 Jack J. Dongarra, Fred G. Gustavson, and Alan H. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984. doi:10.1137/1026003.

15 Michael Griebel. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer, 2007.

16 Shay Gueron and Vlad Krasnov. Fast quicksort implementation using avx instructions. *The Computer Journal*, 59(1):83–90, 2016. doi:10.1093/comjnl/bxv063.

17 David R. Helman, David A. Bader, and Joseph JáJá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998. doi:10.1006/jpdc.1998.1462.

18 Michael Herf. Radix tricks, 2001. URL: http://stereopsis.com/radix.html.

19 Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. Aa-sort: A new parallel sorting algorithm for multi-core simd processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 189–198, 2007. doi:10.1109/PACT.2007.4336211.

20 Hiroshi Inoue and Kenjiro Taura. Simd- and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015. doi:10.14778/2809974.2809988.

21 Intel Corporation. Developer Reference for Intel® Integrated Performance Primitives. URL: https://software.intel.com/en-us/ipp-dev-reference.

22 Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2016.
23 Intel Corporation. Intel® Integrated Performance Primitives Cryptography: Developer Guide, 2020. URL: https://software.intel.com/sites/default/files/ippcp-devguide.pdf.
24 Intrinsics Guide. Intel intrinsics guide. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.
25 Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ With the OpenCV Library*. O'Reilly Media, 2016.
26 Ronald Kneusel. *Random Numbers and Computers*. Springer, 1st edition, 2018.
27 Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.
28 Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 2006.
29 Ibrahim Küçük. *Astrophysics*. IntechOpen, 2012.
30 Tapio Lahdenmäki and Michael Leach. *Relational Database Index Design and the Optimizers*. John Wiley & Sons, 2005.
31 Stewart A. Levin. A fully vectorized quicksort. *Parallel Computing*, 16:369–373, 1990. doi:10.1016/0167-8191(90)90074-J.
32 Peng Li, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc D. Riedel. Computation on stochastic bit streams digital image processing case studies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(3):449–462, 2014. doi:10.1109/TVLSI.2013.2247429.
33 Michal Ozery-Flato and Ron Shamir. Sorting by translocations via reversals theory. In *Comparative Genomics*, pages 87–98. Springer, 2006. doi:10.1007/11864127_8.
34 Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992. doi:10.1142/S0129626492000337.
35 Steven Ross. Boost.sort. URL: https://www.boost.org/doc/libs/1_67_0/libs/sort/doc/html/index.html.
36 Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.
37 Howard Jay Siegel. The universality of various types of simd machine interconnection networks. *SIGARCH Comput. Archit. News*, 5(7):70–79, 1977. doi:10.1145/633615.810655.
38 Harold S. Stone. Sorting on star. *IEEE Transactions on Software Engineering*, SE-4(2):138–146, 1978. doi:10.1109/TSE.1978.231484.
39 Martin Weisser. *Essential Programming for Linguistics*. Edinburgh University Press, 2009.
40 Zekun Yin, Tianyu Zhang, André Müller, Hui Liu, Yanjie Wei, Bertil Schmidt, and Weiguo Liu. Efficient parallel sort on avx-512-based multi-core and many-core architectures. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 168–176. IEEE, 2019. doi:10.1109/HPCC/SmartCity/DSS.2019.00038.
41 Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721. ACM, 1991. doi:10.1145/125826.126164.

## A Worked Example of Vectorized In-Place Partitioning

Our vectorized partitioning technique is a hybrid of the techniques of Gueron and Krasnov [16], and, Bramas [5] with additional optimizations. Our technique for partitioning an array is demonstrated in Figure 6 using vectors that can hold four elements.

In the example in Figure 6, the *array* to be partitioned consists of 20 elements. The pivot is 49. An arrow, with the text *partition vector* above, indicates the partitioning of a single vector using the pivot vector *v_pivot*. While partitioning a vector, the elements smaller than or equal to the pivot are arranged contiguously at the beginning of the vector, and the elements greater than the pivot are arranged thereafter. The function *partition vector* returns the number of elements greater than the pivot. Based on this information the indices

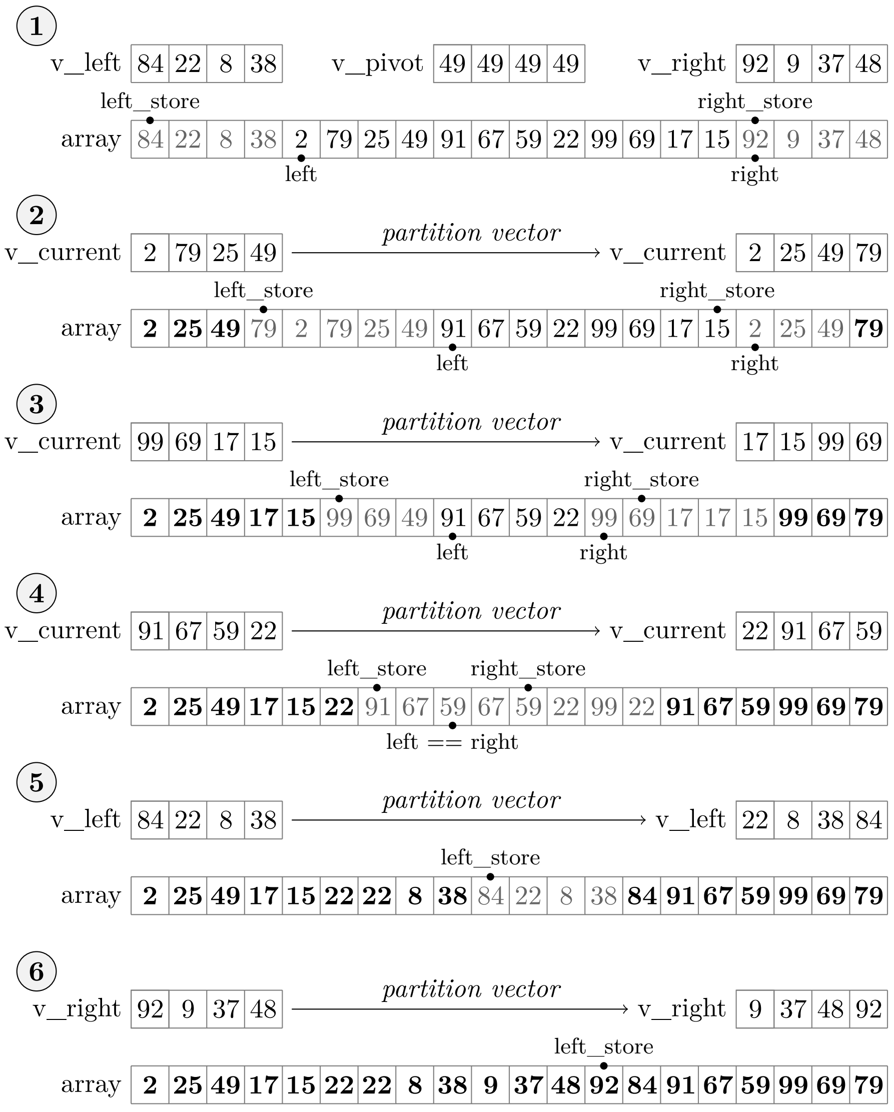


**Figure 6** Vectorized partitioning example.

*left_store* and *right_store* are updated. The indices *left_store* and *right_store* indicate the store points in the *array* for a partitioned vector. The *left* and *right* indices, on the other hand, are used to determine the next elements to load for partitioning. Already partitioned elements are printed in bold. Elements that can be overwritten are grayed out. The details of the procedure in Figure 6 are as follows:

1 At the beginning of partitioning, the first four elements of the *array* are stored in the vector *v_left* and the last four elements in the vector *v_right*. The vectors *v_left* and *v_right* are partitioned only after all the other elements of the *array* have been processed.

2 The four elements at the index *left* are loaded to *v_current*. The vector *v_current* is partitioned and stored at the outer ends of the *array*. The store point *left_store* moves three index positions to the right, because in *v_current* three elements are smaller than or equal to the pivot. Equivalently, *right_store* moves one index position to the left, since in *v_current* one element is greater than the pivot.

③ Since the *array* contains fewer partitioned elements on the right side than on the left (one element on the right side and three on the left), the four elements are loaded into *v_current* from the right side of the *array*. After the partitioning of *v_current* and its storage on both sides of the *array* according to the store points *left_store* and *right_store* a total of five elements are partitioned to the left and three to the right side of the *array*. The store points *left_store* and *right_store* are updated and *right* is moved four index positions to the left.

④ The vector *v_current* is again loaded from the index *right*, since the right side of the *array* contains fewer partitioned elements than the left side. The vector *v_current* is partitioned and stored according to the store points *left_store* and *right_store*. The index *right* and the store points are updated. The indices *left* and *right* are the same, which means that the vectors *v_left* and *v_right* must be partitioned before the complete *array* is fully partitioned.

⑤ The initially created vector *v_left* is partitioned and stored on both sides of the *array* according to *left_store* and *right_store*. Three elements are smaller than the pivot, so *left_store* is moved three index positions to the right. The store point *right_store* can be ignored because it is no longer needed.

⑥ The initially created vector *v_right* is partitioned and stored according to *left_store*. Three elements are smaller than the pivot, so *left_store* is moved three index positions to the right.

# Minimum Scan Cover and Variants - Theory and Experiments

**Kevin Buchin**
Department of Mathematics & Computer Science, TU Eindhoven, The Netherlands

**Sándor P. Fekete**
Department of Computer Science, TU Braunschweig, Germany

**Alexander Hill**
Department of Computer Science, TU Braunschweig, Germany

**Linda Kleist**
Department of Computer Science, TU Braunschweig, Germany

**Irina Kostitsyna**
Department of Mathematics & Computer Science, TU Eindhoven, The Netherlands

**Dominik Krupke**
Department of Computer Science, TU Braunschweig, Germany

**Roel Lambers**
Department of Mathematics & Computer Science, TU Eindhoven, The Netherlands

**Martijn Struijs**
Department of Mathematics & Computer Science, TU Eindhoven, The Netherlands

## Abstract

We consider a spectrum of geometric optimization problems motivated by contexts such as satellite communication and astrophysics. In the problem MINIMUM SCAN COVER WITH ANGULAR COSTS, we are given a graph $G$ that is embedded in Euclidean space. The edges of $G$ need to be scanned, i.e., probed from both of their vertices. In order to scan their edge, two vertices need to face each other; changing the heading of a vertex incurs some cost in terms of energy or rotation time that is proportional to the corresponding rotation angle. Our goal is to compute schedules that minimize the following objective functions: (i) in MINIMUM MAKESPAN SCAN COVER (MSC-MS), this is the time until all edges are scanned; (ii) in MINIMUM TOTAL ENERGY SCAN COVER (MSC-TE), the sum of all rotation angles; (iii) in MINIMUM BOTTLENECK ENERGY SCAN COVER (MSC-BE), the maximum total rotation angle at one vertex.

Previous theoretical work on MSC-MS revealed a close connection to graph coloring and the cut cover problem, leading to hardness and approximability results. In this paper, we present polynomial-time algorithms for 1D instances of MSC-TE and MSC-BE, but NP-hardness proofs for bipartite 2D instances. For bipartite graphs in 2D, we also give 2-approximation algorithms for both MSC-TE and MSC-BE. Most importantly, we provide a comprehensive study of practical methods for all three problems. We compare three different mixed-integer programming and two constraint programming approaches, and show how to compute provably optimal solutions for geometric instances with up to 300 edges. Additionally, we compare the performance of different meta-heuristics for even larger instances.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms; Theory of computation → Computational geometry; Applied computing → Operations research

**Keywords and phrases** Graph scanning, angular metric, makespan, energy, bottleneck, complexity, approximation, algorithm engineering, mixed-integer programming, constraint programming

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.4

**Related Version** *Full Version*: `http://arxiv.org/abs/2103.14599` [11]

**Supplementary Material** *Software*: `https://github.com/ahillbs/minimum_scan_cover`
archived at `swh:1:dir:7524ee7b5783c34cf8bb187a116438dcb5c7beb8`

**Funding** Work at TU Braunschweig was partially supported under grant FE407/21-1, "Computational Geometry: Solving Hard Optimization Problems" (CG:SHOP).

## 1 Introduction

For many aspects of wireless communication, the relative direction, i.e., the angle of visibility between different locations, plays a crucial role. A particularly striking example occurs in the context of inter-satellite communication, which requires focused transmission, with communication partners facing each other with directional, paraboloid antennas or laser beams. This makes it impossible to exchange information with multiple partners at once. Moreover, a change of communication partner requires a change of heading, which is costly in the context of space missions with limited resources, making it worthwhile to invest in good schedules. Problems of this type do not only arise from long-distance communication. They also come into play when astro- and geophysical measurements are to be performed, in which groups of spacecraft can determine physical quantities not just at their current locations, but also along their common line of sight; see [21] for a description.

In previous theoretical work [15], we considered an optimization problem arising from this context: How can we schedule a given set of intersatellite communications, such that the overall timetable is as efficient as possible? In the problem Minimum Scan Cover with Angular Costs (MSC), the task is to establish a collection of connections between a given set of locations, described by a graph $G = (V, E)$ that is embedded in space. For any connection (or scan) of an edge, the two involved vertices need to face each other; changing the heading of a vertex to cover a different connection takes an amount of time proportional to the corresponding rotation angle. In [15], the goal considered was to minimize the time until all tasks are completed, i.e., compute a geometric schedule of minimum makespan.

Given the importance of conserving energy on space (or drone) missions, this Minimum Makespan Scan Cover (MSC-MS) is not the only important objective: In Minimum Total Energy Scan Cover (MSC-TE), the goal is to minimize the sum of all rotation angles; in Minimum Bottleneck Energy Scan Cover (MSC-BE), the task is to limit the energy used by any one vertex by minimizing the maximum total rotation at one vertex.

In this paper, we complement the previous theoretical results on MSC-MS (hardness and approximation) [15] by presenting an NP-hardness proof and a 2-approximation for MSC-TE and MSC-BE for bipartite graphs in two dimensions. For one-dimensional instances of MSC-TE and MSC-BE, we show a polynomial time algorithm and an upper bound independent of the chromatic number, which shows a fundamental difference to MSC-MS. Most importantly, we provide a comprehensive study of practical methods for all three objective functions. We compare three different mixed-integer programming (MIP) and two constraint programming (CP) approaches, and show how to compute provably optimal solutions for geometric instances with up to 300 edges. Additionally, we evaluate the practical performance of approximation algorithms and heuristics for even larger instances.

### 1.1 Previous Work

The use of directional antennas has introduced a number of geometric questions. The paper at hand expands on previous work of Fekete, Kleist, and Krupke [15], who investigated MSC-MS and identified a close connection to graph coloring and the (directed) cut cover number. More precisely, MSC-MS in 1D and 2D is in $\Theta(\log \chi(G))$, which implies that even in 1D, there exists no constant-factor approximation for MSC-MS. For 2D, they present a 4.5-approximation for bipartite instances and show inapproximability for a constant better than $3/2$. This yields an $O(c)$-approximation for $k$-colored graphs with $k \leq \chi(G)^c$.

Further problems involving directional antennas have been considered by Carmi et al. [12], who study the $\alpha$-MST problem. This problem arises from finding orientations of directional antennas with $\alpha$-cones, such that the connectivity graph yields a spanning tree of minimum weight, based on bidirectional communication. They prove that for $\alpha < \pi/3$, a solution may not exist, while $\alpha \geq \pi/3$ always suffices. See Aschner and Katz [7] for more recent hardness proofs and constant-factor approximations for some specific values of $\alpha$.

Many other geometric optimization problems deal with turn cost. Arkin et al. [5, 6] show hardness of finding an optimal milling tour with turn cost, even in relatively constrained settings, and give approximation algorithms. The complexity of finding an optimal cycle cover in a 2-dimensional grid graph was stated as *Problem 53* in *The Open Problems Project* [14] and shown to be NP-hard in [16], which also provides constant-factor approximations; practical methods and results are given in [17], and visualized in the video [9].

Finding a fastest roundtrip for a set of points in the plane for which the travel time depends only on the turn cost is called the Angular Metric Traveling Salesman Problem. Aggarwal et al. [1] prove hardness and provide an $O(\log n)$ approximation algorithm. For the abstract version on graphs in which "turns" correspond to weighted changes between edges, Fellows et al. [19] show that the problem is fixed-parameter tractable in the number of turns, the treewidth, and the maximum degree. Fekete and Woeginger [18] consider the problem of connecting a set of points by a tour in which the angles of successive edges are constrained.

MSC-MS is a special case of scheduling in which the cost of a current job depends on the sequence of the already processed ones; e.g., Allahverdi et al. [2, 3, 4] provide a comprehensive overview, especially on practical work. In the context of earth observation, Li et al. [22] and Augenstein et al. [8] describe MIPs and heuristics to schedule image acquisition and downlink for satellites for which rotation and setup costs are taken into account.

## 1.2 Preliminaries and Problem Definitions

For all considered versions of Minimum Scan Cover (MSC), the *input* consists of a (straight-line) embedded (not necessarily crossing-free) graph $G = (V, E)$ with a finite vertex set $V \subset \mathbb{R}^2$. We refer to the elements of $V$ as *points* when their specific locations in $\mathbb{R}^2$ are relevant; if we focus on graph properties, we may also refer to them as *vertices*. We denote the undirected edge between $u, v \in V$ by $uv$. For $v \in V$, we let $N(v) = \{u \in V : uv \in E\}$ be all vertices adjacent to $v$, and $E(v) = \{uv : u \in N(v)\}$ be all edges incident to $v$. For two adjacent edges $uv, vw \in E(v)$, let $\alpha(uv, vw) \in [0, 180°]$ denote the smaller angle between the lines supporting the segments $uv$ and $vw$. The *output* for each problem is a *scan cover* $S : E \to \mathbb{R}^+$, such that for all pairs of adjacent edges $e, e'$, we have $|S(e) - S(e')| \geq \alpha(e, e')$. The geometric interpretation of a scan cover is that all points $v \in V$ have a *heading* that can change over time, and that if $S(uv) = t$ then $u$ and $v$ *face* each other at time $t$. In this case, we say that the edge $uv$ is *scanned* at time $t$. Thus, the above condition on $S$ guarantees that $S$ complies with the necessary rotation time if rotation speed is bounded by 1.

A *rotation scheme* describes the geometric change of headings of the vertices over a time interval of length $T$, i.e., it is a map $r \colon V \times [0, T] \mapsto [0°, 360°]$. The *total rotation angle* of a vertex $v$ in $r$ is the total amount that $v$ rotates over $[0, T]$. For a given scan cover $S$, we are particularly interested in edges that are scanned consecutively. Therefore, let $\nu_S(e, e') = 1$ if $e$ and $e'$ share exactly one vertex $v$ and the edge $e'$ is scanned directly after $e$ at $v$; otherwise $\nu_S(e, e') = 0$.

**The Problems.** We consider the following three problems, defined by their respective objectives. For a given graph $G = (V, E)$ with vertices in the plane, find a scan cover $S$ with

- Minimum Makespan (MSC-MS): $\min \max_{e \in E} S(e)$

- Minimum Total Energy (MSC-TE): $\min \sum_{v\in V} \sum_{e,e'\in E(v)} \alpha(e,e')\cdot \nu_S(e,e')$
- Minimum Bottleneck Energy (MSC-BE): $\min \max_{v\in V} \sum_{e,e'\in E(v)} \alpha(e,e')\cdot \nu_S(e,e')$

Concentrating on the expensive and algorithmically challenging part of efficient rotations between the edges, we do not fix the initial heading of the satellites. In fact, all algorithms can be easily adapted to handle fixed initial headings. Furthermore, for every of the three objectives, an $f$-approximation can be converted into a $(f+1)$-approximation for the problem variant with fixed initial headings.

For a vertex $v$, we denote by $\Lambda(v)$ the minimum angle, such that a cone of this angle with apex $v$ contains all edges in $E(v)$. We call such a cone a $\Lambda$-cone of $v$ and call the complement of such a cone an *outer* cone of $v$. A $\Lambda$-*cover* is a scan cover for which every vertex $v$ rotates in a single direction, either clockwise or counterclockwise, with a total rotation angle equal to $\Lambda(v)$. Note that different vertices can rotate in different directions. A $\Lambda$-cover minimizes both the MSC-TE and MSC-BE objectives.

### 1.3 Outline and Results

This paper consists of a theoretical part (Section 2) and a practical part (Section 3). Our theoretical results complement the work on MSC-MS [15] by hardness and approximation results for the two new objectives MSC-TE and MSC-BE. In Section 2, we show that both problems can be solved efficiently in 1D; on the other hand, we prove that they are NP-hard in 2D, even for bipartite graphs. Finally, we complement the hardness results by providing 2-approximations for bipartite graphs and $O(\log n)$-approximations for general graphs. Our practical study in Section 3 considers optimal solutions in Section 3.1 and heuristic solutions in Section 3.2. For optimal solutions, we develop three mixed integer linear programs (MIPs), as well as two constraint programs (CPs) and evaluate their practical performance on a suite of benchmark instances. Solving instances of MSC-TE and MSC-BE to provable optimality turned out to be quite difficult; for MSC-MS, we were able to solve instances with up to 300 edges, based on one CP. In addition, we compared the solution quality of four (meta-)heuristics and the approximation algorithms on larger instances with up to 800 edges. In our experiments, a genetic algorithm and the intermediate solution after timeout of one CP produced the best solutions. All omitted proofs can be found in the full version [11].

## 2 Complexity Results

Fekete, Kleist, and Krupke studied the computational complexity of MSC-MS [15]. In this section, we provide new results for MSC-BE and MSC-TE.

For MSC-MS in 1D, when all vertices are placed on a line, there exists no constant-factor approximation unless $P = NP$ [15]. In contrast, we show that MSC-TE and MSC-BE in 1D can be solved efficiently.

▶ **Theorem 1.** *MSC-TE and MSC-BE in 1D are in $P$. Moreover, denoting by $k$ the number of vertices with neighbors to both sides, the objective value is $0$ for $k = 0$, while for $k > 0$ it is $180° \cdot k$ for MSC-TE and $180°$ for MSC-BE.*

Next we show that for 2D instances of MSC-TE and MSC-BE, there does not exist an efficient algorithm, unless $P = NP$. Specifically, we show that MSC-TE and MSC-BE are NP-hard in 2D, even when the underlying graph $G = (V, E)$ is bipartite. Our proof is based on the observation that if a $\Lambda$-cover exists, any scan cover optimal for MSC-TE is a $\Lambda$-cover. If additionally all vertices have the same $\Lambda(v)$, any scan cover optimal for MSC-BE is a $\Lambda$-cover. We show finding a $\Lambda$-cover is NP-hard via a reduction from the NP-complete problem Monotone Not-all-equal 3-satisfiability (MNAE3SAT) [23, 27], defined as follows: Given a set of Boolean variables $X$ and a set of clauses $\mathcal{C}$ with at most 3 literals from $X$ all of which are not negated, is there a 0/1-assignment to the variables in $X$, such that for each clause in $\mathcal{C}$, not all variables have the same value?

Given an instance $I$ of the MNAE3SAT, we construct an MSC instance $G_I$ (with the same $\Lambda(v)$ for all vertices) that has a $\Lambda$-cover if and only if $I$ has a valid variable assignment. Recall that in a $\Lambda$-cover, the edges of each vertex are scanned in either clockwise or counter-clockwise order. We encode variable assignment by the rotation direction of the vertices in $G_I$ in a $\Lambda$-cover. A variable is encoded by a subgraph that contains a set of vertices that have the same rotation direction in a $\Lambda$-cover, and a clause by a subgraph that contains three vertices that cannot all have the same rotation direction in a $\Lambda$-cover. We connect variables to clauses via wires, which are encoded by a subgraph that contains two vertices that have the same rotation direction in a $\Lambda$-cover. See Figure 1 for an example of the construction.

▶ **Theorem 2.** *MSC-TE and MSC-BE in 2D are NP-hard, even for bipartite graphs.*

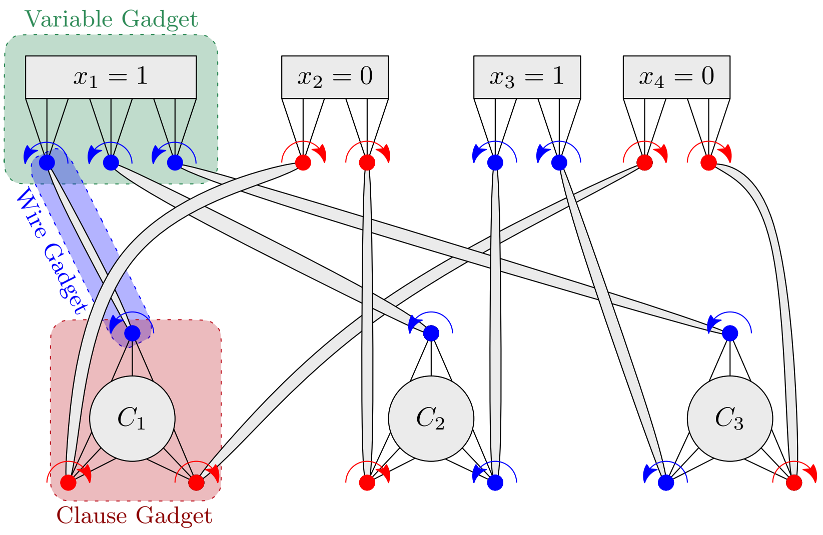


**Figure 1** The constructed graph $G_I$ for the instance $(x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_3 \vee x_4)$ of MNAE3SAT. The gadgets are drawn symbolically; also shown are the directions of the connector vertices corresponding to the satisfying assignment $x_1 = x_3 = 1, x_2 = x_4 = 0$.

The construction in the proof of Theorem 2 establishes a gap between optimal and suboptimal solutions, which implies a constant-factor approximation lower bound for MSC-BE.

▶ **Corollary 3.** *MSC-BE in 2D is NP-hard to approximate within a factor of* 1.04*, even for bipartite graphs.*

Next, we complement the 4.5-approximation algorithm for MSC-MS in bipartite graphs in the plane [15] by presenting an approximation algorithms for both remaining objectives.

▶ **Theorem 4.** *There exists a 2-approximation algorithm for MSC-BE and MSC-TE for each bipartite graph $G = (V_1 \uplus V_2, E)$ embedded in the plane.*

**Proof.** Defining $V := V_1 \cup V_2$, the values $\max_{v\in V} \Lambda(v)$ and $\sum_{v\in V} \Lambda(v)$ are clearly lower bounds on the value of a scan cover minimizing MSC-BE and MSC-TE, respectively.

We use the following geometric property based on alternating angles that is also used in [15]: Starting with opposite headings, two vertices face their edge at the same time when both start a full clockwise rotation simultaneously. Defining start headings $r(v, 0) := 0°$ for $v \in V_1$ and $r(v, 0) := 180°$ for $v \in V_2$, the clockwise rotation scheme induces a scan cover $S$ by defining the scan time $S(e)$ of edge $e$ as the time when its two vertices face each other.

We now show that in the rotation scheme $r'$ induced by $S$, i.e., every vertex $v$ starts to head towards its edge first scanned in $S$ and then follows the order on $E(v)$ defined by $S$, the total rotation angle of each vertex $v$ is at most $2\Lambda(v)$. To this end, we consider three types of vertices; for an illustration consider Figure 2.

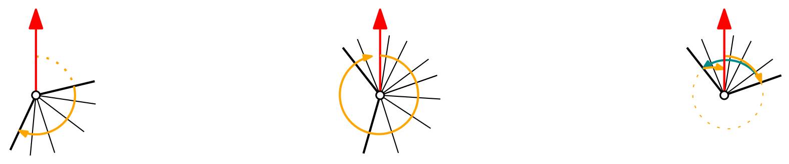

**(a)** Case (a). **(b)** Case (b). **(c)** Case (c).

**Figure 2** Illustration for the rotation scheme $r'$ in the proof of Theorem 4.

**(a)** Case: $r(v, 0)$ lies outside the $\Lambda$-cone of $v$. Then all edges of $v$ are scanned by a clockwise rotation, one after the other. Hence, $v$ has a total rotation angle of $\Lambda(v)$.

**(b)** Case: $r(v, 0)$ lies inside the $\Lambda$-cone of $v$ and $\Lambda(v) \geq 180°$. Going over all edges clockwise takes at most a full rotation of $360° \leq 2\Lambda(v)$.

**(c)** Case: $r(v, 0)$ lies inside the $\Lambda$-cone of $v$ and $\Lambda(v) < 180°$. Let $e_1$ and $e_2$ denote the bounding edges of the $\Lambda$-cone such that $S(e_1) \leq S(e_2)$. By definition, the minimal angle of $e_1$ and $e_2$ is $\Lambda(v) < 180°$. Splitting the $\Lambda$-cone of $v$ into two halves at $r(v, 0)$, $v$ scans the edges in each half in clockwise direction, rotating an angle of $\Lambda(v)$ counterclockwise between $e_1$ and $e_2$. It follows that the total rotation angle of $v$ is at most $2\Lambda(v)$.

As the total rotation angle is at most $2\Lambda(v)$ for each vertex $v$, MSC-BE and MSC-TE are upper bounded by $2 \cdot \max_{v\in V} \Lambda(v)$ and $2 \cdot \sum_{v\in V} \Lambda(v)$. Together with the lower bounds provided above, this shows that this scan cover is a 2-approximation for *either* objective. ◀

▶ **Corollary 5.** *Let $G = (V_1 \cup V_2, E)$ be a bipartite graph embedded in the plane such that the points of $V_1$ and $V_2$ can be separated by a line. Then an optimal MSC-BE and MSC-TE of $G$ can be found in polynomial time.*

**Proof.** We follow the same technique as in the proof of Theorem 4. We may assume without loss of generality that the separating line is vertical and that the points of $V_1$ lie left of the line. Then, with the above definitions, every vertex is in case (a), i.e., the total rotation angle for each vertex $v$ is $\Lambda(v)$. Consequently, the resulting scan cover is optimal for both MSC-BE and MSC-TE. ◀

The insights from Theorem 4 yield an approximation algorithm for $k$-colored graphs.

▶ **Corollary 6.** *For MSC-TE and MSC-BE of $k$-colored graphs embedded in 2D, there exists an $O(\log(k))$-approximation.*

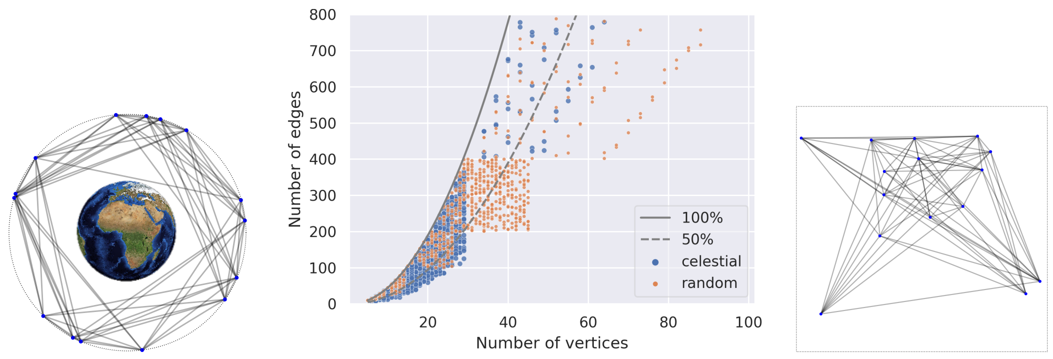
**Proof.** The edges of a $k$-colored graph $G$ can be covered by $\lceil \log_2(k) \rceil$ bipartite graphs $G_i$ [25]. For each $G_i$, we use the 2-approximation of Theorem 4. Clearly, for both objectives, the optimal scan time of $G$ is lower bounded by the optimal scan time of every subgraph. Consequently, scanning all $G_i$ takes at most $\sum_i 2 \cdot OPT(G_i) \leq 2\lceil \log_2(k) \rceil \cdot OPT(G)$, where $OPT$ denotes the optimum scan time for the respective objective. For adjusting the headings between the scan covers of the bipartite graphs, we need $(\lceil \log_2(k) \rceil - 1)$ transition phases each of which needs at most $OPT(G)$. Hence, the total scan time is upper bounded by $(3\lceil \log_2(k) \rceil - 1) \cdot OPT(G)$. ◀

## 3 Experiments

For our experimental evaluation, we considered two types of benchmark instances in 2D, which we call *random* and *celestial*. Random instances are generated by placing $n$ points chosen uniformly at random from the unit square, with each edge chosen with probability $p$. Note that the visible area of a satellite constellation on the same altitude in low Earth orbit is fairly close to a set of co-planar points and hence the square (or plane) serves as a reasonable approximation.

Celestial instances are inspired by real-world instances of satellites in a shared orbit, in which they maintain their relative positions while orbiting around a central body like Earth, as long as no explicit orbit-changing maneuvers are carried out. They are characterized by a set of points on a circle and a central circular obstacle. The points on the circle are chosen uniformly at random; an edge exists if and only if its vertices *see* each other, i.e., the edge does not intersect the central obstacle. Examples and the distribution of the nearly 2000 instances with up to 800 edges used for our experiments can be seen in Figure 3.

All experiments were run on Intel Core i7-3770 with 3.4 GHz and 32 GB of RAM.



**(a)** A celestial instance. **(b)** Instance distribution. **(c)** A random instance.

**Figure 3** Examples of instances with 15 vertices and $\sim 70$ edges: (a) celestial, (c) random, and (b) the instance distribution. Auxiliary lines in (b) indicate graphs with edge densities 50% (dashed) and 100% (solid) of complete graphs.

### 3.1 Exact Algorithms

We developed three mixed integer programs (MIPs) and two constraint programs (CPs) to solve instances to provable optimality. Note that not every program solves all three problems. An experimental evaluation is given at the end of this section. While we focus on 2-dimensional geometric instances, all formulations are applicable to all metric cost functions.

### 3.1.1 Mixed Integer Program 1 (MSC-MS, MSC-TE, MSC-BE)

Our first MIP, denoted by MIP-1, uses two types of variables. The first type are real variables $t_e \geq 0$ for all $e \in E$. The second type are Boolean variables $x_{(e,e')}$ for all ordered edge pairs $(e,e') \in E^2$. In a computed solution, the variables $t_e$ define a scan cover in which $S(e) := t_e$ and the value of $x_{(e,e')}$ corresponds to $\nu_S(e,e')$. Because $\nu_S(e,e') = 0$ if $|e \cap e'| \neq 1$, we directly set $x_{(e,e')} := 0$ in these cases. Consequently, the objective functions can be expressed by substitution of $S(e)$ with $t_e$ and $\nu_S(e,e')$ with $x_{(e,e')}$. Note that a min-max objective can be implemented by a single additional real variable and one additional constraint for each term in the objective.

We introduce a set of constraints to guarantee that the $t$-variables and the $x$-variables arise from a valid scan cover. Because the angle function $\alpha$ fulfills the triangle inequality, it suffices to ensure the time difference of the $t$-variables for all $x_{(e,e')} = 1$. We know that $M_1 := \log_2 n \cdot 360°$ is an upper bound on the minimal makespan for a graph $G$ in 2D with $n$ vertices [15]. Moreover, a makespan of $M_2 := |E| \cdot 180°$ allows to scan each edge individually, and thus an optimal scan cover of MSC-BE and MSC-TE can be realized in this makespan. Therefore, by inserting the correct $M_i$, we can enforce feasible scan times by using the Big-M method.

$$\forall v \in V, \forall (e,e') \in E(v) \times E(v), e \neq e' \colon \quad t_{e'} \geq t_e + \alpha(e,e') - (1 - x_{(e,e')}) \cdot M_i. \tag{1}$$

This leaves us with ensuring that the $x$-variables correspond to a feasible scan cover. First, for every vertex $v$, an incident scanned edge $e$ has at most one predecessor edge and one successor edge in the scan order.

$$\forall v \in V, e \in E(v) \colon \quad \sum_{e' \in E(v), e' \neq e} x_{(e,e')} \leq 1 \quad \text{and} \quad \sum_{e' \in E(v), e' \neq e} x_{(e',e)} \leq 1 \tag{2}$$

Second, the total number of scanned edges at vertex $v$ is $|E(v)|$, i.e., the number of consecutively scanned edge pairs, is $|E(v)| - 1$.

$$\forall v \in V \colon \quad \sum_{e,e' \in E(v) \times E(v), e \neq e'} x_{(e,e')} = |E(v)| - 1 \tag{3}$$

Together, Equations (2) and (3) enforce that every vertex has exactly one first and one last scanned edge in the induced scan order. Because Equation (1) enforces that the scan times obey the rotation times, there are no cycles in the sequence defined by $x$ if all angles are positive. This fact is very similar to the Miller-Tucker-Zemlin formulation of the TSP [24]. In the presence of 0°-angles, we dynamically add the following constraint similar to the Dantzig formulation [13] to separate these cycles.

$$\forall v \in V, \forall S \subsetneq E(v), S \neq \emptyset \colon \quad \sum_{e \in S, e' \in E(v) \setminus S} x_{(e,e')} + x_{(e',e)} \geq 1 \tag{4}$$

### 3.1.2 Mixed Integer Program 2 (MSC-MS)

The abstract definition of the MSC [15] can be directly implemented as a MIP, because absolute values can be implemented using a Boolean variable. Some modern solvers like Gurobi actually provide this functionality directly. Like for MIP-1 (Section 3.1.1), we have a real-valued variable $t_e \geq 0$ for each $e \in E$ that states its scan time. We try to keep the maximum value assigned to any $t_e, e \in E$ as low as possible. For every two incident edges

$vw$ and $vu$, we only have the constraint that $t_{vw}$ and $t_{vu}$ have to be at least the time apart that $v$ needs to rotate between these two. This results in the following MIP-2.

$$\begin{aligned} &\min && \max_{e\in E} t_e && && (5)\\ &\text{s.t.} && |t_{vw} - t_{vu}| \geq \alpha(vu, vw) && \forall vw, vu \in E && (6)\\ & && t_e \geq 0 && \forall e \in E && (7) \end{aligned}$$

The main difference to MIP-1 is that we do not keep a record of the actually performed rotations. As a consequence, MIP-2 can only be used for MSC-MS. However, on the positive side, we do not need to dynamically add additional cycle constraints.

### 3.1.3 Mixed Integer Program 3 (MSC-TE, MSC-BE)

The third MIP (defined by Equations (2)–(4) and (8)), denoted by MIP-3, is a variant of MIP-1 (Section 3.1.1) in which the $t$-variables and the corresponding Big-M based constraint (Equation (1)) are removed. As a consequence, we may use it for MSC-BE and MSC-TE, as they only need the $x$-variables.

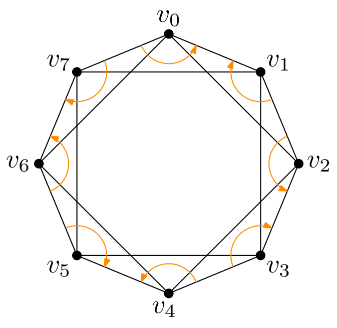
It is possible that the scan orders at the individual vertices are cycle free, but that the overall schedule has a deadlock when the vertices wait for each other, see Figures 4a and 4b. We therefore prohibit directed cycles in the scan order defined by the $x$-variables (if not already separated by Equation (4)) dynamically via callbacks for every newly found integral solution. Violated constraints can be found via a simple DFS search.

$$\forall k \in \mathbb{N}_{|V|}, \forall (e_0, e_1, \ldots, e_{k-1}) \in E^k: \quad x_{(e_{k-1},e_0)} + \sum_{i=0,1,\ldots k-2} x_{(e_i,e_{i+1})} \leq k-1 \qquad (8)$$

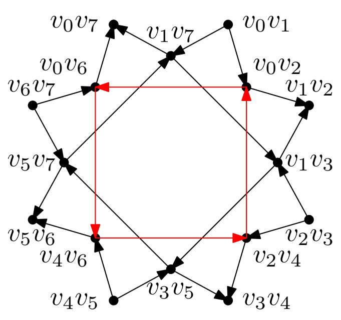
Note that these cycles can also happen in MIP-1, but only with zero rotation costs between the involved edges. Thus, they are irrelevant for the solution, as all of these edges can be scanned at once.

### 3.1.4 Constraint Program 1 (MSC-MS)

Our first constraint program (denoted by CP-1) has the same formulation as MIP-2. The only difference between the CP version and the MIP version lies in the employed solver. In particular, absolute values can be modeled directly.



**(a)** Rotation scheme.



**(b)** Cycle in scan order.

**Figure 4** A globally infeasible edge order fulfilling Equations (2) and (3), i.e., it is cycle-free at each vertex: (a) its rotations scheme (b) the resulting edge order that contains a cycle. An arc $(e, e')$ in this graph corresponds to an $x_{(e,e')} = 1$.

### 3.1.5 Constraint Program 2 (MSC-TE, MSC-BE)

Our second constraint program (defined by Equations (2), (3), and (9)), denoted by CP-2, is similar to MIP-3 described in Section 3.1.3. However, MIP-3 adds Equations (4) and (8) dynamically, which our CP does not support. Because adding all these constraints directly results in a prohibitively large formulation, we instead use a conditional variant of the Miller-Tucker-Zemlin [24] formulation to eliminate cycles in the scan order. Different from MIPs, we do not need the Big M method for CPs, but can implement conditional constraints directly. More precisely, we add the variables $o_e \in \mathbb{N}_{|E|}, e \in E$ that state the cycle-free scan order of the edges, which is enforced by the constraints

$$\forall (e,e') \in E \times E\colon \qquad o_{e'} - o_e \geq 1 \quad \text{if } x_{(e,e')} = 1. \tag{9}$$

### 3.1.6 Experimental Evaluation of Exact Algorithms

We used *Gurobi* (v9.0.1) for solving the MIPs and *CP-SAT* of Google's *or-tools* (v7.7.7810) for solving the CPs. CP-SAT, which is based on a SAT solver, requires all coefficients and variables to be integral for computational efficiency. We therefore convert the floating point values to integral values including the first eight floating point digits (rounded, decimal). While this weakens the accuracy, we calculated a theoretical maximal deviation of less than $1 \times 10^{-4}\,\%$, which we consider negligible and comparable to the accuracy of the MIP solver.

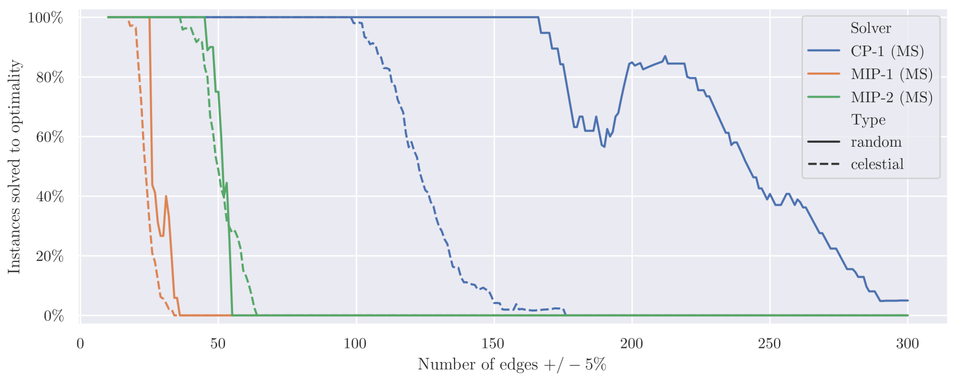
We considered all solvers for the three objectives on the two instance types described in the preliminaries. We evaluated how many instances of which size could still be solved to provable optimality within a time limit of 900 sec; see Figure 5. For MSC-MS, CP-1 has a clear lead, solving 50 % of the instances with $242 \pm 5\%$ edges for random instances, and $125 \pm 5\%$ edges for celestial instances. In our experiments, neither MIPs was able to solve any instance with more than 70 edges to provable optimality. For MSC-TE, MIP-1 and MIP-3 performed better than CP-2, but all solvers could barely solve instances with more than 30 edges. While MIP-1 has a more direct objective without auxiliary constraints and variables as needed for MSC-MS, its actual performance was slightly worse. For MSC-BE, CP-2 performed considerably better; for celestial instances, it can solve instances nearly twice as large ($\geq 50\,\%$ at $48 \pm 5\%$ edges) than the MIPs. Surprisingly, MIP-1 was slightly better than CP-2 for random instances, being able to solve 50 % of the instances with $61 \pm 5\%$ edges. Overall, CPs appear to be considerably more effective than MIPs, and random instances show to be easier to solve than celestial ones.

## 3.2 Approximations and Heuristics

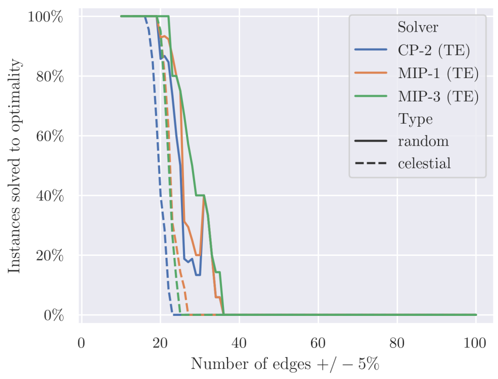
For larger instances (beyond the size that was solvable to provable optimality), we developed additional methods based on approximation algorithms and heuristics that provided good (but not provably optimal) solutions.

### 3.2.1 Bipartite Approximation Algorithms with Coloring Partition

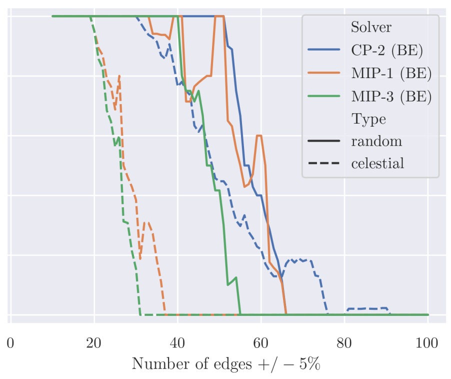
The constant-factor approximation algorithms for bipartite graphs extend to general graphs by partitioning them into bipartite graphs and applying the corresponding approximation algorithm to each of the bipartite subgraphs. More specifically, assigning a vector over $\{0,1\}$ with $\lceil \log_2 k \rceil$ bits to each color class of a $k$-colored graph induces a covering of its edge set with $\lceil \log_2 k \rceil$ bipartite graphs; for more details see Motwani and Naor [25]. For MSC-MS, this even preserves the approximation factor [15]. We use the well-engineered



**(a)** Makespan.



**(b)** Total Energy.



**(c)** Bottleneck Energy.

**Figure 5** Performance of the exact solver measured in how many instances with $m \pm 5\%$ edges can be solved to provable optimality within 900 sec. The bump for CP-1 starting at 200 can be explained by the instance distribution that at this point includes more instances with lower degree.

*dsatur* heuristic [10] for the graph coloring problem, which is shipped with the *pyclustering*-package [26]. Concatenating the solutions of the bipartite graphs yields a feasible scan cover; here we use a greedy approach to minimize the transition costs. We denote this method by APX.

### 3.2.2 (Meta-)Heuristics

We also considered a number of (meta-)heuristics for optimizing the three objectives.

**Greedy:** Scan the first edge regarding a given or random order and then scan the edge that increases the objective the least, until all edges are scanned. If multiple edges are equally good, the first one regarding the order is selected. Many edges can be inserted without extra cost and thus the initial edge order has a strong influence on the result.

**Iterated Local Search (ILS):** This simple but potentially slow heuristic considers for a given start solution (in this case of *Greedy*) all possible swaps of edges; the locally best swap is carried out, until no further improvement is possible.

**Simulated Annealing (SA):** This common variation of *Iterated Local Search* performs swaps according to a probability based on the Boltzmann function $\textbf{Boltzmann}(T, s_1, s_2) = e^{1/T \cdot (s_2 - s_1)}$, where $s_1$ is the objective value of the current best solution, $s_2$ is the objective value of the considered solution, and $T \in \mathbb{R}^+$ is the current *temperature*. The temperature decreases over time and with it the likelihood of a worse solution being used. If the objective does not improve for some time, the temperature is increased in order to escape the local minimum. Due to randomization, we can run multiple searches in parallel. We terminate if the solution has not improved for some time.

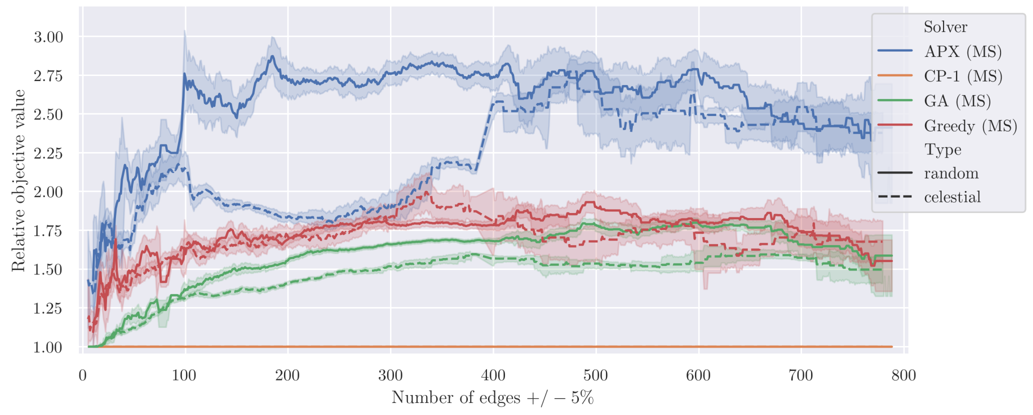
**Genetic Algorithm (GA):** We start with an initial population of 200 solutions generated by a randomized Greedy. A solution is encoded by assigning each edge a fractional number between 0.0 and 1.0, similar to [20]. The scan order is determined by sorting the edges by these numbers. In each round, we build a new population by selecting the best 10% of the old population (*elitism*) and then fill the rest of the population by crossovers of the old generation. For a crossover, we select two solutions of the old generation with a probability matching their objective values (*uniform selection*) and for each edge we choose with equal probability either the number from the first or second solution (*uniform crossover*). If by chance, two edges get the same number, we randomly change one of them without influencing the order. Of the new generation of solutions, 3% are selected for mutation. A mutation applies Greedy with a probability of 60% (the old order is used as initial edge order) or changes each edge with a 3% probability to a new random number. This is repeated until we either reach a time limit of 900 sec, 300 generations, or 60 generations without improvement. The best solution found during this process is then returned.

### 3.2.3 Experimental Evaluation of Approximations and Heuristics

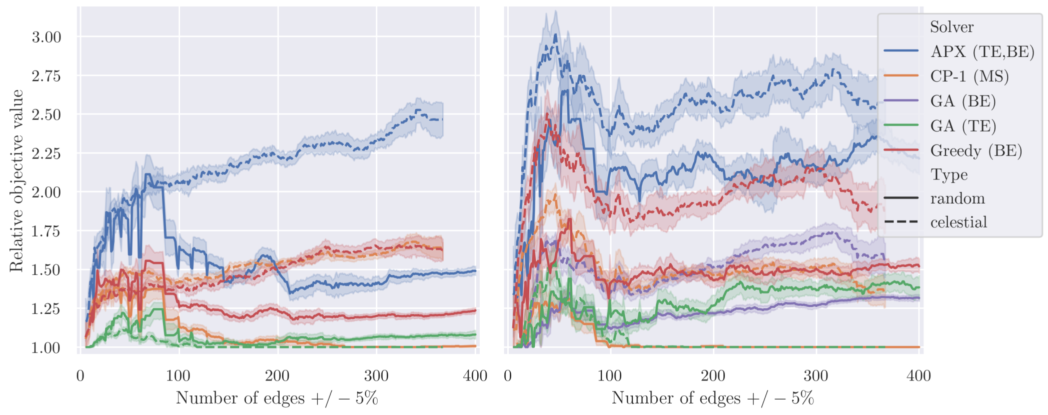
Figure 6 shows experimental results for heuristically solving instances with up to 800 edges with a 900 sec time limit (at which point the current solution is returned). For MSC-MS, CP-1 yields the best results even for larger instances (where it is aborted by the time limit) by a margin of 25 % to 50 % to the next best algorithm, GA. For MSC-TE, the genetic algorithm turned out to be the best approach for celestial instances by a margin of over 50 % for the larger instances. Surprisingly, CP-1 (optimizing for MSC-MS) yields slightly better solutions than the genetic algorithm for random instances of MSC-TE. The most interesting results are for MSC-BE. Here, CP-1 achieves the best results by a margin of over 20 % for random instances, and GA (TE) the best results for celestial instances by a margin of over 40 %. The excellent performance of CP-1 can be explained by a strong correlation of MSC-MS and MSC-BE for random graphs, as shown in Figure 7. The fact that GA (TE) is actually better in optimizing MSC-BE than GA (BE) can be explained by the weaker gradients of bottleneck objectives, because only a small part of the solution (the most expensive vertex) actually contributes to the value. However, the initial bump, at which the exact solver of MSC-BE still yields (better) solutions, indicates that these solutions could be far from optimal and that there may still be room for improvement.

Overall, either CP-1 or GA (TE) yields the best solutions. CP-1 is especially strong on random instances for all three objectives. The approximation algorithm is usually among the worst. For MSC-MS, the algorithm performs a full rotation for nearly all instances, as $\max_{v \in V} \Lambda(v)$ is usually above 180°. Note that the factor can be worse than the approximation factor 4.5 (resp. 2), because these are not bipartite graphs.

In Figure 7 (first row, fourth and last column) we can additionally see that for MSC-MS the objective correlates strongly with the number of edges for celestial instances and with the average degree for random instances. Total energy seems to primarily correlate with the number of edges for both types; our random instances are on average twice as expensive. For MSC-BE, only random graphs seem to have a significant correlation to MSC-MS and the average degree.



**(a)** Makespan.



**(b)** Total energy. **(c)** Bottleneck energy.

**Figure 6** Relative performance of the non-exact methods, measured by the obtained objective value divided by the best known value. We used the same instances for the exact solver, so the better denominator creates a small bump for smaller instance sizes, in particular for MSC-BE. Except for CP-1, the exact solvers did not yield good solutions for larger instances, if any at all, and are thus excluded for readability. The plots show the mean and the corresponding 95 % confidence interval. We highlight the difference between the two instance types by using different styles for the lines. Note that because these are relative values, a comparison of the performance over the different objectives is not possible. ILS and SA are excluded for readability and perform only slightly better than Greedy.

## 4 Conclusion and Open Problems

We studied problems of minimum scan cover with three different practically relevant objective functions, providing both theoretical and practical contributions: complexity and algorithmic results for the new objectives (MSC-TE and MSC-BE), and practical methods for computing provably optimal solutions for smaller and near-optimal solutions for larger instances.

In particular, we developed multiple MIP and CP formulations and demonstrated that instances of MSC-MS can be solved reliably for instances with more than 100 edges using constraint programming which performs much better than our MIP approaches. While this approach generalizes also to 3D, we only tested 2D instances; it is open whether these results also carry over to 3D. MSC-TE and MSC-BE can only be solved to optimality for much smaller instances. For solving larger instances without guarantee of optimality, we

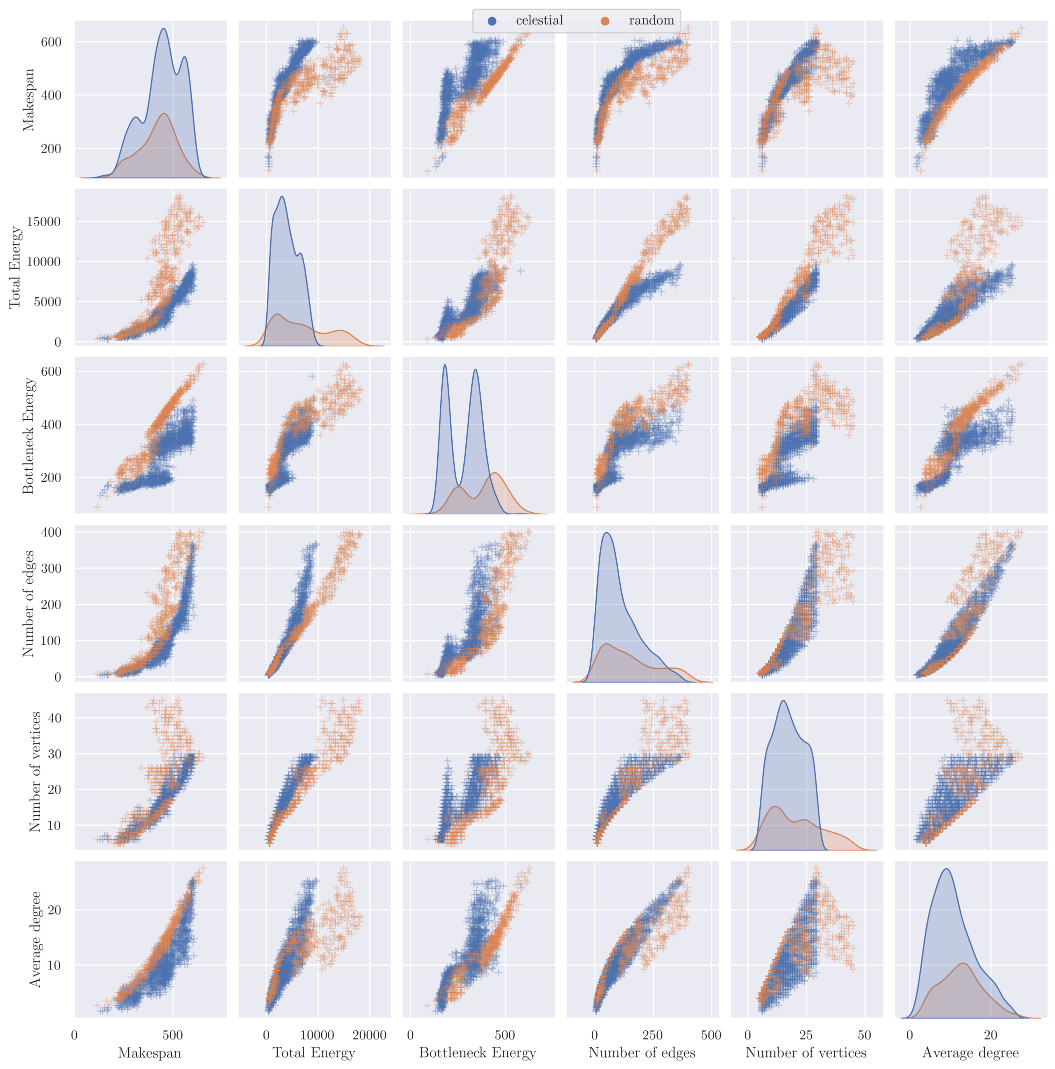


**Figure 7** Correlation and distribution of the best known objectives and instance properties. The diagonal shows the density distribution of the x-values. The scatter plots have a point for every existing value pair, which allows to detect correlations.

evaluated approximation algorithms and a spectrum of meta-heuristics. Within the given time limit, CP-1 provided the best solutions for all MSC-MS instances, and even the random instances for MSC-TE and MSC-BE, despite only optimizing for MSC-MS. For celestial instances of MSC-TE and MSC-BE, the genetic algorithm optimizing for MSC-TE provides the best solutions. However, the results indicate perspectives for improving the optimization of MSC-BE.

At this point, fully dynamic instances (in which the vertices change their relative positions to each other over time, such as for satellites with different orbit parameters) are yet to be explored. These promise to be even more challenging, due to bigger gaps between optimal and suboptimal solutions, resulting from possibly long delays when a limited communication window has been missed.

## References

1 Alok Aggarwal, Don Coppersmith, Sanjeev Khanna, Rajeev Motwani, and Baruch Schieber. The angular-metric traveling salesman problem. *SIAM J. Comp.*, 29(3):697–711, 1999.

2 Ali Allahverdi. The third comprehensive survey on scheduling problems with setup times/costs. *Eur. J. Oper. Res.*, 246(2):345–378, 2015.

3 Ali Allahverdi, Jatinder N.D. Gupta, and Tariq Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27(2):219–239, 1999.

4 Ali Allahverdi, C.T. Ng, T.C. Edwin Cheng, and Mikhail Y. Kovalyov. A survey of scheduling problems with setup times or costs. *Eur. J. Oper. Res.*, 187(3):985–1032, 2008.

5 Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S. B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. In *Symp. Disc. Alg. (SODA)*, pages 138–147, 2001.

6 Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S.B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. *SIAM J. Comp.*, 35(3):531–566, 2005.

7 Rom Aschner and Matthew J. Katz. Bounded-angle spanning tree: modeling networks with angular constraints. *Algorithmica*, 77(2):349–373, 2017.

8 Sean Augenstein, Alejandra Estanislao, Emmanuel Guere, and Sean Blaes. Optimal scheduling of a constellation of Earth-imaging satellites, for maximal data throughput and efficient human management. In *Int. Conf. Automated Planning & Scheduling*, pages 345–352, 2016.

9 Aaron T. Becker, Mustapha Debboun, Sándor P. Fekete, Dominik Krupke, and An Nguyen. Zapping Zika with a mosquito-managing drone: Computing optimal flight patterns with minimum turn cost. In *Symp. Comp. Geom. (SoCG)*, pages 62:1–62:5, 2017.

10 Daniel Brélaz. New methods to color the vertices of a graph. *Comm. ACM*, 22(4):251–256, 1979.

11 Kevin Buchin, Sándor P. Fekete, Alexander Hill, Linda Kleist, Irina Kostitsyna, Dominik Krupke, Roel Lambers, and Martijn Struijs. Minimum scan cover and variants – theory and experiments, 2021. `arXiv:2103.14599`.

12 Paz Carmi, Matthew J. Katz, Zvi Lotker, and Adi Rosén. Connectivity guarantees for wireless networks with directional antennas. *Comp. Geom.*, 44(9):477–485, 2011.

13 George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Op. Res. Soc. of America*, 2(4):393–410, 1954.

14 Erik D. Demaine, Joseph S. B. Mitchell, and Joseph O'Rourke. The Open Problems Project. URL: `http://cs.smith.edu/~orourke/TOPP/`.

15 Sándor P. Fekete, Linda Kleist, and Dominik Krupke. Minimum scan cover with angular transition costs. In *Symp. Comp. Geom. (SoCG)*, volume 164, pages 43:1–43:18, 2020. `doi:10.4230/LIPIcs.SoCG.2020.43`.

16 Sándor P. Fekete and Dominik Krupke. Covering tours and cycle covers with turn costs: Hardness and approximation. In *Int. Conf. Algor. Complexity (CIAC)*, pages 224–236, 2019.

17 Sándor P. Fekete and Dominik Krupke. Practical methods for computing large covering tours and cycle covers with turn cost. In *Alg. Engin. Exp. (ALENEX)*, pages 186–198, 2019.

18 Sándor P. Fekete and Gerhard J. Woeginger. Angle-restricted tours in the plane. *Comp. Geom.*, 8:195–218, 1997.

19 Mike Fellows, Panos Giannopoulos, Christian Knauer, Christophe Paul, Frances A. Rosamond, Sue Whitesides, and Nathan Yu. Milling a graph with turn costs: A parameterized complexity perspective. In *Worksh. Graph Theo. Conc. Comp. Sci. (WG)*, pages 123–134, 2010.

20 M. Gholami, M. Zandieh, and A. Alem-Tabriz. Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns. *Int. J. Adv. Manufact. Tech.*, 42(1-2):189–201, 2009.

21 Haje Korth, Michelle F. Thomsen, Karl-Heinz Glassmeier, and W. Scott Phillips. Particle tomography of the inner magnetosphere. *J. Geophys. Res.: Space Phys.*, 107(A9):SMP–5, 2002.

22 Guoliang Li, Lining Xing, and Yingwu Chen. A hybrid online scheduling mechanism with revision and progressive techniques for autonomous Earth observation satellite. *Acta Astronautica*, 140:308–321, 2017.

23 László Lovász. Coverings and colorings of hypergraphs. In *Southeastern Conf. Combin., Graph Th., Comput. (SEICCGTC)*, pages 3–12, 1973.

24 Clair E. Miller, Albert W. Tucker, and Richard A. Zemlin. Integer programming formulation of traveling salesman problems. *J. ACM*, 7(4):326–329, 1960.

25 Rajeev Motwani and Joseph (Seffi) Naor. On exact and approximate cut covers of graphs. Technical report, Stanford University, Stanford, CA, USA, 1994.

26 Andrei Novikov. PyClustering: Data mining library. *J. Open Source Softw.*, 4(36):1230, 2019.

27 Thomas J. Schaefer. The complexity of satisfiability problems. In *Symp. Th. Comp. (STOC)*, pages 216–226, 1978. doi:10.1145/800133.804350.

# Three Is Enough for Steiner Trees

**Emmanuel Arrighi**
University of Bergen, Norway

**Mateus de Oliveira Oliveira**
University of Bergen, Norway

## Abstract

In the Steiner tree problem, the input consists of an edge-weighted graph $G$ together with a set $S$ of terminal vertices. The goal is to find a minimum weight tree in $G$ that spans all terminals. This fundamental NP-hard problem has direct applications in many subfields of combinatorial optimization, such as planning, scheduling, etc. In this work we introduce a new heuristic for the Steiner tree problem, based on a simple routine for improving the cost of sub-optimal Steiner trees: first, the sub-optimal tree is split into three connected components, and then these components are reconnected by using an algorithm that computes an optimal Steiner tree with 3-terminals (the roots of the three components). We have implemented our heuristic into a solver and compared it with several state-of-the-art solvers on well-known data sets. Our solver performs very well across all the data sets, and outperforms most of the other benchmarked solvers on very large graphs, which have been either obtained from real-world applications or from randomly generated data sets.

**2012 ACM Subject Classification** Theory of computation → Theory of randomized search heuristics

**Keywords and phrases** Steiner Tree, Heuristics, 3TST

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.5

**Supplementary Material** *Software*: `https://github.com/AutoProving/3TST`
archived at `swh:1:dir:740b703cb9dceee55ef1571f49d1b58701a2082f`

**Funding** *Emmanuel Arrighi*: Research Council of Norway (Grant no. 274526).
*Mateus de Oliveira Oliveira*: Research Council of Norway (Grant no. 288761) and Sigma2 Network (NN9535K).

## 1 Introduction

In the *Steiner tree* problem, we are given an undirected graph $G$ whose edges are weighted with non-negative values, and a subset of vertices $S$, whose elements are called *terminals*. The goal is to find a minimum-weight tree in $G$ whose nodes span all terminal in $S$. This is a fundamental NP hard problem [14], which has been studied since the seventies [11] and which has found applications in several fields of research such as planning [16], social networks [17], sensor networks [18], community detection [7], VLSI circuit design [13], as well as in numerous applications in industry [6].

Since Steiner tree is an NP-hard problem, most research surrounding this problem has been devoted both to the task of developing heuristics that work reasonably well in practice, and to the task of developing approximation algorithms that provide approximation guarantees within polynomial time. In particular, a short list of heuristic paradigms that have been used to attack the Steiner-tree problem include simulated annealing [19], genetic algorithms [5], logic programming [20] and constraint solving [8]. On the other hand, when it comes to approximation algorithms, the approximation ratio guarantee achievable by algorithms running in polynomial time was gradually improved from 2 [27] to 1.39 [4] in a span of two and a half decades[27, 29, 1, 30, 21, 15, 12, 24, 25, 4]. It is worth noting that unless $P = NP$, the Steiner tree problem in general graphs cannot be approximated within a factor of $1 + \epsilon$ for sufficiently small $\epsilon > 0$ [2].

In this work, we introduce a new heuristic for the Steiner tree problem and show that on large graphs, it outperforms several state of the art algorithms. Our heuristic has two main components. First, we devise a method that can be used to quickly compute a good initial Steiner tree. The second component is based on an improvement procedure that takes a Steiner tree as input and tries to output a lighter Steiner tree. Essentially, this procedure is executed until a specific time limit is up. It is worth noting that our improvement strategy is similar in spirit to an improvement procedure used in a celebrated approximation algorithm due to Robins and Zelikovsky [25].

This improvement procedure can be explained in two high-level steps. First, given a (potentially suboptimal) Steiner tree $T_0$, one appropriately split it into three subtrees $T_1$, $T_2$ and $T_3$ such that, all terminals are contained in $T_1 \cup T_2 \cup T_3$. Then those three subtrees are reconnected together by solving an instance of the Steiner tree problem with three terminals. This gives a new Steiner tree $T_0'$. As the Steiner tree problem with three terminals can be solve exactly and efficiently, the weight of $T_0'$ is at most the weigh t of $T_0$.

We observe that there are two crucial differences between the optimization procedure used in our heuristic and the one used in the algorithm of [25]. The first is that their algorithm is run in a complete graph where for each two vertices $v$ and $u$, the weight of the edge $\{v, u\}$ is the weight of the shortest path between $v$ and $u$ in the original graph, while our algorithm is run without the need to compute shortest paths between all possible pairs of vertices. The second difference is that in Robins and Zelikovsky's algorithm, the split is chosen to be the optimal one, while in our algorithm we replace optimality by a greedy selection strategy. Building the complete graph and looking for the optimum splitting is costly and cannot be done on large graphs. Therefore, the algorithm of [25] cannot handle large real world instances. By doing something that does not need such large structures our approach can handle large instances. As a consequence, our optimization procedure performs especially well on large graphs. We also note that this optimization procedure can also be used to improve the weight of sub-optimal Steiner trees output by other solvers.

To validate our new heuristic, we implement a solver in C++ and benchmark it against several state of the art solvers for the Steiner tree problem on well known data sets. These solvers implement several paradigms, such as genetic algorithms, linear programming algorithms, local search algorithms as well as algorithms with approximation guarantees. The data sets were obtained from a variety of sources, such as established real-world benchmarks for the Steiner tree problem, data sets of common use in the field of road networks, and a synthetic data set where instances are generated at random. Our solver obtained very good results in most data sets. In particular our solver was able to obtain solutions that are on par with those obtained by solvers that employed large scale mixed-linear programming suites such as SCIP [10]. Our solver was also able to handle very large instances, with millions of vertices and edges, while most of the solvers failed in these instances. A detailed exposition of these results can be found in Section 4.

## 2 Preliminaries

In this section, we set notation for basic graph-theoretic concepts used in the description of our algorithm. We let $\mathbb{N}$ denote the set of *natural numbers*. For a finite set $V$, we let $\mathcal{P}(V, 2) = \{\{u, v\} : u, v \in V, u \neq v\}$ be the set of unordered pairs of elements from $V$.

An *undirected graph* is a pair $G = (V, E)$ where $V$ is a set of *vertices* and $E \subseteq \mathcal{P}(V, 2)$ is a set of *undirected edges*. We may write $V(G)$ to denote the vertex-set of $G$ and $E(G)$ to denote the edge-set of $G$. An *edge-weighted* graph is a graph $G = (V, E)$ together with a *cost function* $\mathrm{cost} : E \to \mathbb{N}$. We let $\mathrm{cost}(G)$ be the sum of the costs of all edges in $G$.

We say that a graph $H$ is a *subgraph* of a graph $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. For each subset $X \subseteq V(G)$, the subgraph of $G$ induced by $X$ is the graph $G[X]$ with vertex set $X$ and edge set $E(G) \cap \mathcal{P}(X, 2)$.

A *walk* in a graph $G$ is a sequence of vertices $v_1, \ldots, v_k$ such that for each $i$ in $\{1, \ldots, k-1\}$, $\{v_i, v_{i+1}\} \in E(G)$. A *path* in $G$ is a walk in which all vertices are distinct. We let $\text{dist}(v, v')$ be the minimum number of edges in a path between $v$ and $v'$. We say that $G$ is *connected* if for each two vertices $v_1$ and $v_2$ there is a path between $v_1$ and $v_2$. A *cycle* is a walk $v_1, \ldots, v_k$ such that $v_1 = v_k$ and $v_i \neq v_j$ for $i, j \leq k$ and $i \neq j$. A graph is *acyclic* if it contains no cycle.

A *tree* is a connected acyclic graph $T$. A *rooted tree* is a tree $T$ together with a distinguished vertex $r$. If $T$ is a rooted tree with root $r$, and $v \in V(T)$ is such that $r \neq v$, then the parent of $v$ is the unique neighbour $v'$ of $v$ such that $\text{dist}(r, v') < \text{dist}(r, v)$. Note that the root $r$ does not have a parent. Each neighbour $v'$ of $v$ with $\text{dist}(r, v') > \text{dist}(r, v)$ is called a *child* of $v$. A *leaf* of $T$ is a vertex with no child. A *descendant* of a vertex $v$ is a vertex $v'$ such that any path between $r$ and $v'$ contains $v$. We consider $v$ to be a descendant of itself. The *subtree of $T$ rooted at $v$* is the subgraph of $T$ induced by the set of descendants of $v$.

Given a graph $G$, a spanning tree for $G$ is a tree $T$ such that $T$ is a subgraph of $G$ and $V(T) = V(G)$. Given a connected edge-weighted graph $G$, and a vertex $v \in v(G)$, a *shortest-path tree* for $G$ rooted at $v$ is an edge-weighted spanning tree $T$ of $G$ rooted at $v$ such that for each vertex $u \in V(G)$, the distance between $v$ and $u$ in $G$ is equal to the distance between $v$ and $u$ in $T$.

Let $G$ be an undirected edge-weighted graph and let $S \subseteq V(G)$ be a subset of vertices of $G$ whose elements are called *terminals.* A *Steiner tree* in $G$ is a subgraph $T$ of $G$ such that $T$ is a tree and $S \subseteq V(T)$. We note that $T$ may contain non-terminal vertices. We call the vertices in $V(T) \backslash S$, *Steiner points.* The cost of a tree, $\text{cost}(T)$, is the sum of the costs of its edges.

Let $G$ be a graph and $H$ be a connected subgraph of $G$. The *contraction* of $H$ in $G$, written $G/H$, is the graph obtained from $G$ by first deleting all vertices of $H$, then by adding a new vertex $v_H$, and finally by connecting $v_H$ to a vertex $u \in V(G) \backslash V(H)$ in $G/H$ if and only if there is an edge between $u$ and some vertex from $V(H)$ in $G$. The weight of an edge between $v_H$ and $u$ is the minimum weight of an edge connecting a vertex of $H$ to $u$. We note that in practice, since our graphs are weighted, contraction of a subgraph $H$ will be simulated by simply setting the weights of the edges of $H$ to 0, and therefore, the topology of the original graph remains unchanged.

## 3 Our Heuristics

In this section, we describe the main components of the heuristic used in our Steiner tree solver. There are three main components. A pre-processing component, which simplifies the input graph, a greedy procedure that constructs an initial Steiner tree, an optimization function that takes a given Steiner tree as input and outputs another Steiner tree that is at least as light as the original one. This optimization procedure is then repeated until it stabilizes, or until the time is up. Once the solution can not be improved, our solver starts again with a new starting Steiner tree. It repeats this procedure until it receives a timeout signal. These three components are described in more details below.

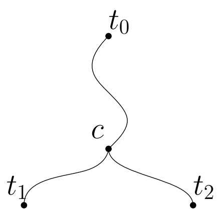


**Figure 1** A Steiner tree with three terminals $t_0$, $t_1$ and $t_2$ is a union of shortest paths between a center vertex $c$ and $t_0$, $t_1$ and $t_2$ respectively.

## 3.1 Preprocessing

During the preprocessing step, we modify the input graph by applying two standard rules [28, 22] with the goal of eliminating redundancies. Once a solution is obtained in the modified graph, this solution can be easily converted into a solution to the original graph. The two preprocessing rules we apply are the following.

1. The first rule removes non-terminal vertices of degree 1 from the graph. These vertices are redundant because if a Steiner tree contains such a vertex, then one can safely delete it from the tree and still obtain a valid Steiner tree.
2. The second rule eliminates non-terminal vertices of degree 2. More precisely, let $u$ be a non-terminal vertex of degree 2 connected to vertices $v_1$ and $v_2$ by edges $e_1 = \{u, v_1\}$ and $e_2 = \{u, v_2\}$ respectively. Then we delete the vertex $u$ and the edges $e_1$ and $e_2$ from the graph. If the graph has an edge of cost $c$ connecting $v_1$ and $v_2$, then we update the cost of this edge to $\min(c, \mathrm{cost}(e_1) + \mathrm{cost}(e_2))$. Otherwise, we just add a new edge $e = \{v_1, v_2\}$ of cost $\mathrm{cost}(e_1) + \mathrm{cost}(e_2)$ to the graph. This rule is repeated until no vertex of degree 2 is left.

This preprocessing step can be done in time $O(n)$ where $n$ is the number of vertices. Note that if a solution to the modified graph contains an edge $e = \{u, v\}$ that is not present in the original graph, then one can obtain a solution to the original graph by replacing each such edge $e$ by a path between $u$ and $v$ in which all internal vertices have degree 2.

## 3.2 Minimum Steiner trees with 2 or 3 terminals

A fact that we will use often both in the construction of an initial Steiner tree and in our optimization procedure is the fact that Steiner trees with two or three terminals can be computed very quickly by using elementary algorithms. Indeed, a Steiner tree with two terminal vertices $t_0$ and $t_1$ is simply a shortest path between these two vertices. On the other hand, it can be shown that if $T$ is a Steiner tree with 3 terminals $\{t_0, t_1, t_2\}$ then there is a *center vertex* $c$ such that $T$ is obtained by taking the union of the shortest paths $p_0, p_1$ and $p_2$ between $c$ and the terminals $t_0$, $t_1$ and $t_2$ respectively. We call $c$ the center of $T$ (Figure 1). We observe that $c$ can be one of the terminals. Therefore, to construct such a Steiner tree, we can iterate through the vertices of $G$ and set as the center the vertex that minimizes the sum of the lengths of the shortest paths $p_0, p_1$ and $p_2$. In this paper, we will call this procedure $\mathsf{3Steiner}(G, t_0, t_1, t_2)$. We note that $\mathsf{3Steiner}(G, t_0, t_1, t_2)$ is a deterministic procedure that produces an optimal Steiner tree with three terminals, and runs in time $O(n + m\log(m))$ where $n$ the number of vertices and $m$ the number of edges.

## 3.3 Constructing an Initial Solution

Once the preprocessing procedure has been applied, our algorithm proceeds to construct suitable initial solutions. We actually implement two initialization functions. Both functions take as input a triple $(G, S, r)$ consisting of a graph $G$, a set of terminals $S$ and a root vertex $r$ as input, and return a Steiner tree with terminals $S$ rooted at $r$. We note that the root can be an arbitrary vertex in the graph, but in our implementation we always choose this root to be a terminal.

The first function, $\mathsf{DetInitialST}(G, S, r)$, is used to construct a reasonable first-solution when our algorithm is executed for the first time. This function is completely deterministic. At each step, the function $\mathsf{DetInitialST}(G, S, r)$ maintains the following data:

1. a partial Steiner tree $T$ spanning some of the terminals;
2. a graph $G/T$ obtained from $G$ by contracting $T$ to its root $r$; and
3. a shortest-path tree $D$ for $G/T$ rooted at $r$.

In the beginning, $T$ contains only the root $r$, $G/T = G$, and $D$ is simply the shortest-path tree for $G$ rooted at $r$. After this initialization has taken place, the algorithm enters in a loop, where at each iteration, two new terminals $t_1$ and $t_2$ are incorporated to the tree. Each iteration consists of three steps.

1. First, one applies a function $\mathsf{SelectTerminals}$ that selects terminals $t_1, t_2$ that will be added to the tree. This function proceeds as follows. First, it sets $t_1$ as the terminal with *greatest* distance to the root vertex $r$ in the graph $G/T$. Note that that the contraction of $T$ is simulated by setting the costs of its edges to 0 in the graph $G$. Subsequently, a shortest path from $r$ to $t_1$ is contracted, and $t_2$ is selected as the terminal with the *greatest* distance to $r$.
2. Once the terminals $t_1$ and $t_2$ have been determined, one calls the function $\mathsf{3Steiner}$ to compute the minimum Steiner tree $T'$ in $G/T$ with respect to the terminal set $\{r, t_1, t_2\}$.
3. Finally, the two trees $T$ and $T'$ are *merged*. This merging process consists in taking the spanning tree of the union $T \cup T'$.

The three steps above are repeated until a Steiner tree spanning at least $|S| - 1$ terminals in $S$ has been obtained. Suppose that some last terminal $t \in S$ is not spanned by $T$, and that $v$ is a vertex in $T$ of minimum distance[1] to $t$. Then the tree $T$ is updated by incorporating to it the shortest path between $v$ and $t$. The algorithm described above is specified more formally in Algorithm 1. The procedure $\mathsf{DetInitialST}(G, S, r)$ has time complexity $O(|S| \cdot (n + m\log(m)))$, where $n$ the number of vertices, $m$ the number of edges of $G$.

Once we have a starting Steiner tree, we will improve it by applying the optimization procedure described in Subsection 3.4. Since this optimization procedure may converge to a local minimum, we will repeat the optimization process with respect to several initial Steiner trees. Nevertheless, from this point on, each initial Steiner tree will be selected using a much cheaper procedure, which we call $\mathsf{RandomInitialST}(G, S, r)$. This procedure simply selects random path between some terminal $t_1$ in $S$ and the root vertex $r$. Subsequently, it selects a random path between some terminal $t_2$ and some vertex in the first path, then a random path between some terminal $t_3$ and some vertex in the previous paths and so on, until all terminals have been selected. Each random path is selected by performing a random walk in the graph starting at the terminal to be added.

[1] More precisely $\mathrm{dist}(v, t) = \min_{u \in T} \mathrm{dist}(u, t)$, where the distance function is computed with respect to $G$.

**Algorithm 1** DetInitialST$(G, S, r)$.

```
Input: An edge weighted graph $G$, a set of terminals $S$, a vertex $r$
Output: A Steiner tree in $G$ connecting all terminals in $S$
$T \leftarrow r$
while there are two terminals in $S$ not spanned by $T$ do
    $G' \leftarrow G/T$
    $D \leftarrow \mathsf{ShortestPathTree}(G', r)$
    $t_1, t_2 \leftarrow \mathsf{SelectTerminals}(G', D)$
    $T' \leftarrow \mathsf{3Steiner}(G', r, t_1, t_2)$
    $T \leftarrow \mathsf{SpanningTree}(T \cup T')$
end
if some terminal $t \in S$ is not spanned by $T$ then
    Let $v$ be the vertex of $T$ with smallest distance to $t$
    Set $T \leftarrow T \cup p$ where $p$ is a shortest path between $v$ and $t$ in $G$
end
return $T$
```

## 3.4 Optimization Procedure

Once the preprocessing stage has been completed, and an initial Steiner tree has been computed using the procedure described in the previous subsection, our algorithm applies an optimization procedure that takes a Steiner tree $T$ rooted at a *terminal vertex* $r$ as input, and outputs a Steiner tree $T'$, also rooted at $r$, with equal or smaller weight than $T$. This optimization procedure is repeated until the time is up or until it has stabilized. Alternatively, the procedure can be halted by an external algorithm even if it has not stabilized. In this case the best Steiner tree computed so far is given as the result.

Intuitively, this optimization procedure works in two stages. In the first stage, we split the Steiner tree $T$ into three subtrees $T_1$, $T_2$ and $T_r$, where $T_1$ and $T_2$ are rooted at vertices $v_1$ and $v_2$ respectively, and $T_r$ is rooted at $r$. Subsequently, we select a vertex $v_r$ among the leaves of $T_r$ and reconnect the three subtrees by finding a suitable Steiner tree with respect to terminals $\{v_1, v_2, v_r\}$.

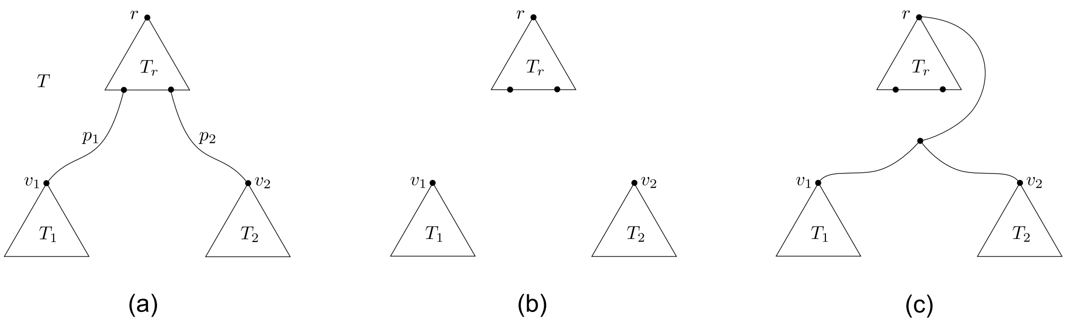


**Figure 2** (a) A Steiner tree $T$, a pair $\{v_1, v_2\}$ of vertices in SelectCut$(T)$, and Steiner paths $p_1$ and $p_2$. (b) The internal vertices of $p_1$ and $p_2$ are removed. This results into three trees $T_r$, $T_1$ and $T_2$. (c) $\{r, v_1, v_2\}$ are connected using an optimal 3-terminal Steiner tree obtained using 3Steiner$(G, r, v_1, v_2)$ function.

Before describing the details of the procedure, we need to define the concept of a *relevant vertex*. Let $T$ be Steiner tree of $G$ a rooted at a vertex $r$. We say that a vertex $v \in V(T)$ is *relevant* for $T$ if $v$ is a terminal or if $v$ has at least 2 children in $T$. A path $p$ in $T$ is a *Steiner path* if the two endpoints of $p$ are relevant for $T$ and if the remaining vertices of $p$ are Steiner points of degree 2 in $T$. Note that each middle vertex of a Steiner path has a unique child. Let $v$ and $v'$ be relevant vertices. We say that $v'$ is a *relevant child* of $v$ if these two vertices are the endpoints of a relevant path in $T$ and if $\text{dist}(r, v) < \text{dist}(r, v')$.

The algorithm starts by applying a simple routine that prunes the input Steiner tree. More precisely, this routine processes the input tree by removing every Steiner point that does not have a terminal as descendant. Such Steiner points do not connect the root to any terminal, and therefore can be safely removed. The resulting tree is still a Steiner tree and every leaf is a terminal.

Subsequently, the algorithm executes a procedure $\mathsf{Improve}(G, S, T)$ that takes a graph $G$, a set of terminals $S$ and a tree $T$ as input, and tries to modify $T$ with the goal of reducing its cost by proceeding as follows.

1. First, we construct a set $\mathsf{SelectCut}(T)$ containing all pairs of the form $\{v_1, v_2\}$ where both $v_1$ and $v_2$ are relevant vertices, and the unique path connecting $v_1$ to $v_2$ in $T$ has no relevant vertex, other than possibly the root (which is always relevant).
2. Now, for each pair of vertices $(v_1, v_2)$ in the list $\mathsf{SelectCut}(T)$ built in the previous step, we call a function $\mathsf{Cut}(T, v_1, v_2)$ that cuts the tree above each of the vertices $v_1$ and $v_2$. More precisely, for each $i \in \{1, 2\}$, one deletes from $T$ the internal vertices of the *unique* Steiner path $p_i$ that starts at $v_i$ that is contained in the unique path between $v_i$ and $v_r$ in $T$. Such Steiner paths $p_1$ and $p_2$ are always guaranteed to exist because the root is a relevant vertex. This process splits the original tree into three disjoint subtrees $T_r$, $T_1$, $T_2$ (Figure 2.(b)).
3. Subsequently, the algorithm contracts each of the three subtrees into a single vertex. More precisely, $T_r$ is contracted to $r$, $T_1$ is contracted to $v_1$, and $T_2$ is contracted to $v_2$. We let $G'$ be the contracted graph. We note that in practice, the contraction of a subtree is simulated by setting the cost of each edge of the subtree to 0.
4. Finally we apply the subroutine $\mathsf{3Steiner}(G', r, v_1, v_2)$ to computes an optimal 3-terminal Steiner tree with terminal set $\{r, v_1, v_2\}$. This tree, together with the three subtrees $T_r$, $T_1$ and $T_2$ give rise to a graph $T_r \cup T_1 \cup T_2 \cup \mathsf{3Steiner}(G', r, v_1, v_2)$ whose weight is at most the weight of the input tree $T$. The algorithm then returns a spanning-tree of this graph.

A summary of the algorithm is provided below (Algorithm 2). Let $t$ be the number of terminals in the graph, and $\Delta$ be the maximum degree of the graph $G$. We note that relevant vertices are either terminals or have degree at least 2, therefore, there are at most $O(t)$ relevant vertices. And for each relevant vertices, the algorithm generates at most $\Delta^2$ pair of vertices. Therefore, the time complexity of the function $\mathsf{Improve}(G, S, r)$ is $O(t \cdot \Delta^2 \cdot (n + m \log(m)))$, where $n$ is the number of vertices and $m$ is the number of edges in $G$.

## 4 Experimental results

We have implemented our heuristic algorithm in C++ and compared it with six state-of-the art solvers for the Steiner tree problem, including solvers that competed at the PACE challenge 2018 [3]. We refer to our solver as 3TST, an acronym for *3-Terminal Steiner Tree*. The remaining solvers in our benchmark are named according to the surnames, or initials of their respective authors. These solvers are listed below.

**Algorithm 2** Improve$(G, S, T)$.

```
Input: An edge weighted graph $G$, a set of terminals $S$ and a Steiner tree $T$ in $G$ spanning $S$
Output: A Steiner tree $T'$ of cost at most cost$(T)$ spanning $S$.
for $\{v_1, v_2\} \in$ SelectCut$(T)$ do
    $G' \leftarrow G$
    if $v_1 \in T$ and $v_2 \in T$ then
        $(T_r, T_1, T_2) \leftarrow$ Cut$(T, v_1, v_2)$
        $G' \leftarrow G'/T_r/T_1/T_2$        ($G'$ is obtained by contracting $T_r, T_1$ and $T_2$)
        $T \leftarrow$ SpanningTree$(T_r \cup T_1 \cup T_2 \cup$ 3Steiner$(G', r, v_1, v_2))$
    end
end
return $T$
```

1. Grandcola's Solver[2] implements a local search algorithm.
2. HTKME Solver[3] combines a star contraction algorithm from [9] with several auxiliary heuristics.
3. HGSSB Solver[4] performs a shortest path heuristic followed by a local optimization step.
4. RCLG Solver[5] implements an evolutionary algorithm.
5. KR Solver[6] reduces the Steiner tree problem to a linear programming problem.
6. AO solver[7] is based on a local optimization heuristic.

We used the original implementation of each of these solvers in our benchmark, without any modification in the code. We benchmarked all the solvers on different data sets, some of which are well established datasets for the Steiner tree problem (PACE2018 dataset [3], Vienna dataset), and some of which are well known datasets in the field of networks (Urban Road Networks set [23], Network repository [26]). Finally, we also compared the solvers on synthetic data sets obtained by generating random $d$-regular graphs for distinct values of $d$.

For each graph considered in our benchmark, we run each solver with a time limit of 30 minutes. When the time limit was reached, each solver received a Unix signal SIGTERM, and had 30 seconds to output a solution before being killed. This is the same experimental setting used in the PACE challenge 2018, whose theme was the Steiner tree problem. Each solution is associated with a score, which is defined as the relative distance of the solution to the best solution found during the whole experiments. If the value of the solution is $v$ and the best solution is $b$ then the score is the ratio $\frac{v-b}{b}$. The score of a solver on a data set is the sum of the scores over all graphs in the data set. With this measure, the lower the score the better is the performance of the solver. In particular, a solver that gets a score of 0 in a given instance is the best solver on that instance.

For some instances of some data sets, some solvers did not output a feasible solution. In these cases, we assigned a default value for the instance. To avoid penalizing excessively a solver on such instances, we have defined the default value as the weight of a Steiner tree obtained by computing a minimum spanning tree of the input graph and subsequently by pruning this tree in such a way that each leaf is a terminal.

[2] http://www.dil.univ-mrs.fr/~gcolas/sgls.c
[3] https://github.com/goderik01/PACE2018
[4] https://github.com/maxhort/Pacechallenge-TrackC/
[5] https://github.com/HeathcliffAC/SteinerTreeProblem
[6] https://github.com/dRehfeldt/scipjack/
[7] https://github.com/SteinerGardeners/TrackC-Version1

**Table 1** Summary: ratios for all algorithms and all data sets. The smallest the value the better is the solver on a given data set. A value of 0 means that the solver was the best in all instances of the data set. Values in bold are the smallest values on the dataset among all the solvers. The superscript number in the column of our solver give the rank of our solver on that data set. For example $0.0397^2$ means that our solver is the second best solver on the data set. (*) means that for some instances, the solver did not output a feasible solution. NC means that the solver could not find a solution for any of the instances of the data set.

| Set/Solvers | AO | Grandcolas | HGSSB | HTKME | KR | RCLG | 3TST |
|---|---|---|---|---|---|---|---|
| PACE 2018 | 1.3756 | 1.6682 | 3.6119 | 0.6589 | 0.1994 | **0.1755** | $0.7130^3$ |
| Geo Original | 0.2579 | 0.4889 | 1.0904 | 0.1032 | **2.2048e-05** | 0.0707 | $0.0397^2$ |
| Geo Prepro. | 0.0661 | 0.0639 | 0.9802 | 0.0723 | **0.0009** | 0.0543 | $0.0472^2$ |
| I Simple | 0.0081 | 0.0354 | 0.2454 | 0.0054 | **0.** | 0.0102 | $0.0056^3$ |
| I Advanced | 0.0294 | 0.0393 | 0.4100 | 0.0146 | **0.** | 0.0128 | $0.0199^4$ |
| 3-regular | 0.3081 | 0.1656 | 1.1526 | 0.2324 | **0.** | 0.0610 | $0.1252^3$ |
| 4-regular | 0.3027 | 0.2385 | 1.1380 | 0.3453 | 5.0182* | **0.0616** | $0.1414^2$ |
| 5-regular | 0.4151 | 0.2642 | 1.0573 | 0.4176 | 9.8697* | **0.0123** | $0.1325^2$ |
| 6-regular | 0.3472 | 0.1668 | 1.1315 | 0.4133 | 8.2781* | **0.0304** | $0.1729^3$ |
| 7-regular | 0.4414 | 0.2909 | 1.0943 | 3.2680* | 8.7525* | **0.0332** | $0.2179^2$ |
| 8-regular | 0.4854 | 0.2840 | 1.1815 | 1.1467* | 3.1947* | **0.0997** | $0.3123^3$ |
| 9-regular | 0.3554 | 0.2205 | 1.0640 | 2.9180* | 2.4698* | **0.1008** | $0.2762^3$ |
| 10-regular | 0.6159 | 0.2950 | 1.2266 | 4.5041* | 4.0639* | **0.0942** | $0.3585^3$ |
| 20-regular | 0.5437 | 0.2385 | 8.4318* | 13.2870* | 6.4683* | **0.0671** | $0.4591^3$ |
| City Road | 4.5950* | 3.2082* | 10.8455* | 0.5466 | **0.** | 5.3683* | $0.9224^3$ |
| Big Road | 4.9210* | NC | NC | 4.1762* | 3.4190* | NC | $\mathbf{0.4513*}^1$ |

Each solver that uses a random procedure has an option to choose a particular seed with the goal of making a computation deterministic, and therefore reproducible. We used the same seed for all experiments (seed = 10). This seed was chosen before experiments were run. All our experiments were executed on Core$^{\text{TM}}$ i7-4770S computers with 16 Gb of RAM running Ubuntu$^{\text{TM}}$ 16.04.

In all figures and tables, our implementation is called 3TST. Table 1 summarises all experiments. This table gathers the sum of ratios obtained by each algorithm on each data set. The symbol (*) following an entry in the table is used to indicate that for some graphs in the data set, the solver did not output a feasible solution. *NC* means that the solver did not output a feasible solution for any of the graph in the data set. We can see that our implementation (3TST) obtains good results in most data sets. Additionally, it is worth noting that our implementation is the one that could find feasible solutions more often in the Big Road Networks data set, which contains graphs with millions of nodes.

## PACE-Challenge

Graphs of the PACE Challenge 2018 dataset were selected by the organizers of the competition from the hard instances of the well known Steinlib and Vienna data sets. The average number of vertices is $27K$, the average number of edges is $48K$, and the average number of terminals is 1114, with a median at 360.5. Finally, most of these instances have treewidth above 40.

Figure 3 shows the score of each solver on each instance. Instances are sorted by increasing number of vertices. On the first half of the instances, our implementation provide decent solution but not as good as RCLG, KR or HTKME which are among the four first in the PACE Challenge 2018. And Figure 4 show a focus on the second half without the HGSSB solver for clarity because it has quite large ratio compare to the other solvers. On those larger instances with smaller average degree, our implementation is very good and almost on par with KR which is the best solver on this part of the data. We note that the implementation of KR is based on the SCIP Optimization Suite, a state-of-the-art tool for mixed integer programming [10]. We also note that the maximum ratio of our solver on these instances was 0.008, while in most instances this ratio was much smaller.

## Vienna set

Graphs in the Vienna set were generated from real-world telecommunication networks at the University of Vienna. This dataset is split into several types of instances. We realized our benchmark in the so called $I$-Instances sub-dataset, which contains 85 instances representing deployment areas from various Austrian cities, but they also include rural areas with smaller population density and very sparse infrastructure. The underlying graphs contain between 7K and 178K nodes, 9K and 239K edges, and between 38 and 4991 terminals. I-instances are available after simple preprocessing that eliminates non-terminal nodes of degrees 1 and that contracts non-terminal nodes of degree 2.

Figure 5 shows the score of each solver on each instance of the I simple preprocessed instances data set without the HGSSB solver for clarity because it has quite large ratio compare to the other solvers. Instances are sorted by increasing number of vertices. We can see a similar behaviour as for the PACE Challenge instances. On small instances, our implementation gives decent solutions and show its strength on larger instance where it give very good solutions. The instances of this data set are small enough so that the KR Solver, which is base on an exact solver, manage to give the best solution in all case. On such data set solutions given by KR Solver can be seen as the ground truth.

## $d$-regular graphs

We generated random $d$-regular graphs using the random generator from the python package Networkx. The number of vertices were chosen uniformly at random from the range $[10000; 200000]$. The weights on the edge follow a normal distribution with mean uniformly chosen from the range $[2000; 10000]$ and standard deviation uniformly chosen from the range $[200; 2000]$. Negative weights were set to 0. The number of terminals was chosen uniformly at random between 2% and 10% of the number of vertices. Terminals were chosen uniformly at random from the vertices. We generated 10 graphs for each $d$ in the set $\{3, 4, 5, 6, 7, 8, 9, 10, 20\}$.

Figure 6 and Table 1 show the evolution of the ratio for each solver with respect to the degree $d$ of the vertices of the graphs. The best solver in these datasets was the solver RCLG, which implements a genetic algorithm. The ratios obtained by our solver (3TST) alternated between the second best and third best. This ratio varied from 0.1252 for 3-regular graphs, to 0.4591 for 20-regular graphs. We note that starting from $k = 4$, the solver KR, which reduces the Steiner tree problem to mixed integer-programming, started failing to give feasible solutions for some instances.

**Table 2** Big road networks: ratios for all algorithms on the big road networks data set. The smallest the value the better is the solver on a given data set. A value of 0 means that the solver was the best the instance. The superscript number in the column of our solver give the rank of our solver on that instance. For example $0.0891^2$ means that our solver is the second best solver on the instance. NC means that the solver could not find a solution for the instance.

| Set/Solvers | AO | Grandcolas | HGSSB | HTKME | KR | RCLG | 3TST |
|---|---|---|---|---|---|---|---|
| Instance 1 | NC | NC | NC | NC | NC | NC | $0.^1$ |
| Instance 2 | NC | NC | NC | 0. | NC | NC | $0.0891^2$ |
| Instance 3 | NC | NC | NC | NC | NC | NC | $0.^1$ |
| Instance 4 | NC | NC | NC | NC | 0. | NC | $0.1219^2$ |
| Instance 5 | NC | NC | NC | NC | 0. | NC | NC |

## City Road Networks

This well known data set contains graphs associated with road networks for 80 of the most populated urban areas in the world. As the original graphs were not connected we filtered each instance by taking only the largest connected component of each graph. Since these graphs do not come originally with information about terminal nodes, we selected these terminals at random. First, we selected a number $r$ uniformly at random in the range between 2% and 10% of the number of vertices. Subsequently, we selected $r$ distinct vertices uniformly at random among the vertices of the graph. The graphs contain between $2K$ and $685K$ nodes, $3K$ and $924K$ edges and between 246 and 53275 terminals

Figure 7 shows the score of each solver on each instance of the City Road Networks set. Instances are sorted by increasing number of vertices. We can see that on the first half of the instances, almost all solvers manage to give really good solution. As the size of the instances grow, the solver KR, which reduces Steiner tree to mixed-integer programming, starts to be the dominant best solver. Nevertheless, our solver still outputs solutions with a very good ration (of at most 0.2).

## Big Road Networks

In this data set was used to push the solvers to their limits. We selected 9 unweighted road networks with more than 1 million nodes. As in the previous data set, the number of terminals was chosen uniformly at random between 2% and 10% of the number of vertices. Since no solver could output a feasible solution for the 4 largest graphs we only show results for the remaining five. These graphs contain between $1087K$ and $6686K$ nodes, $1541K$ and $7013K$ edges, and between $52K$ and $661K$ terminals

Table 2 shows the ratio of each solver on each of these five instances. On this data set only KR, RCLG and our algorithm (3TST) managed to output some solution for some of the instances. On the 9 graphs, KR output 2 solutions, HTKME 1 solution, and our implementation 4 solutions. This dataset highlights one of the strengths of our solver, which is the ability to handle very large instances and still give good solutions, when compared with other solvers.

## 5 Conclusion

In this work, we introduced a simple combinatorial heuristic algorithm for the Steiner tree problem. Our heuristic is similar in spirit to the classic approximation algorithm of Robin and Zelikovsky [25], that works by replacing sub-trees of a prospective solution with Steiner trees on a small set of terminals. In our case, we use a routine that splits a prospective solution Steiner tree into three disjoint subtrees, and that reconnects these subtrees by taking the union with a 3-terminal Steiner tree, where the terminals are the roots of the subtrees. We note that one distinguishing feature of our algorithm is that it is well suited for large graphs, since it does not require the book-keeping of the distances between all pairs of vertices in the graph. Indeed we almost only need to keep track of of the edges of a slightly pre-processed version of the input graph, where non-terminal vertices of degree 1 are removed, and edges containing non-terminal vertices of degree 2 are contracted.

Our experimental results have shown that our algorithm fits well the category of a general purpose Steiner tree heuristic, since it was able to obtain good solutions in all benchmarked datasets when compared with other solvers. We note that the best solver in some datasets was built upon a state-of-the art mixed-integer programming package. In some other datasets, the best solver was based on genetic algorithms. On the other hand, our algorithm essentially consists in the application of a single simple replacement routine that is applied multiple times until the time limit is reached. Still the solutions obtained by our solvers were very competitive, often being the second best in the benchmarks and with a very small ratio $(v-b)/b$ where $v$ is the weight of our solution and $b$ the weight of the best solver. It is also worth noting that our algorithm was able to handle graphs with millions of vertices, while most of the other solvers failed in all these big instances. Finally, it is worth noting that one possible application of our Steiner-tree improvement sub-routine is as a black-box that can be used to improve the solution output by other solvers.

## References

1 Piotr Berman and Viswanathan Ramaiyer. Improved approximations for the steiner tree problem. *J. Algorithms*, 17(3):381–408, 1994.

2 Marshall W. Bern and Paul E. Plassmann. The steiner problem with edge lengths 1 and 2. *Inf. Process. Lett.*, 32(4):171–176, 1989.

3 Édouard Bonnet and Florian Sikora. The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration. In *Proc. of IPEC 2018*, volume 115, pages 26:1–26:15, 2019.

4 Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1):6:1–6:33, 2013.

5 Goutam Chakraborty. Genetic algorithm approaches to solve various steiner tree problems. In *Steiner Trees in Industry*, pages 29–69. Springer, 2001.

6 Xiuzhen Cheng, Yingshu Li, Ding-Zhu Du, and Hung Q Ngo. Steiner trees in industry. In *Handbook of combinatorial optimization*, pages 193–216. Springer, 2004.

7 Mung Chiang, Henry Lam, Zhenming Liu, and H. Vincent Poor. Why steiner-tree type algorithms work for community detection. In *Proc. of (AISTATS 2013)*, volume 31, pages 187–195, 2013.

8 Diego de Uña, Graeme Gange, Peter Schachte, and Peter J Stuckey. Steiner tree problems with side constraints using constraint programming. In *Proc. of the 30th AAAI Conference on Artificial Intelligence*, 2016.

9 Pavel Dvorák, Andreas Emil Feldmann, Dusan Knop, Tomás Masarík, Tomas Toufar, and Pavel Veselý. Parameterized approximation schemes for steiner trees with small number of steiner vertices. In *Proc. of (STACS 2018)*, volume 96, pages 26:1–26:15, 2018.

10 Ambros Gleixner, Leon Eifler, Tristan Gally, Gerald Gamrath, Patrick Gemander, Robert Lion Gottwald, Gregor Hendel, Christopher Hojny, Thorsten Koch, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Jan Merlin Viernickel, Stefan Vigerske, Dieter Weninger, Jonas T.

Witt, and Jakob Witzig. The SCIP Optimization Suite 5.0. ZIB-Report 17-61, Zuse Institute Berlin, December 2017. URL: `http://nbn-resolving.de/urn:nbn:de:0297-zib-66297`.

**11** S Louis Hakimi. Steiner's problem in graphs and its implications. *Networks*, 1(2):113–133, 1971.

**12** Stefan Hougardy and Hans Jürgen Prömel. A 1.598 approximation algorithm for the steiner problem in graphs. In *Proc. of the 10th Symposium on Discrete Algorithms (SODA 1999)*, pages 448–453, 1999.

**13** Rostam Joobbani. *An artificial intelligence approach to VLSI routing*, volume 9. Springer Science & Business Media, 2012.

**14** Richard M. Karp. Reducibility among combinatorial problems. In *Proc. of Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, 1972.

**15** Marek Karpinski and Alexander Zelikovsky. New approximation algorithms for the steiner tree problems. *J. Comb. Optim.*, 1(1):47–65, 1997.

**16** Emil Keyder and Hector Geffner. Trees of shortest paths vs. steiner trees: Understanding and improving delete relaxation heuristics. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 1734–1739, 2009.

**17** Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *Proc. of 15th International Conference on Knowledge Discovery and Data Mining (KDD 2009)*, pages 467–476, 2009.

**18** Sookyoung Lee and Mohamed F. Younis. Recovery from multiple simultaneous failures in wireless sensor networks using minimum steiner tree. *J. Parallel Distrib. Comput.*, 70(5):525–536, 2010.

**19** M Lundy. Applications of the annealing algorithm to combinatorial problems in statistics. *Biometrika*, 72(1):191–198, 1985.

**20** Mohamed El Bachir Menai. A logic-based approach to solve the steiner tree problem. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 73–79. Springer, 2009.

**21** Hans Jürgen Prömel and Angelika Steger. Rnc-approximation algorithms for the steiner problem. In *Proc. of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1997), Proceedings*, volume 1200 of *LNCS*, pages 559–570, 1997.

**22** Daniel Rehfeldt, Thorsten Koch, and Stephen J. Maher. Reduction techniques for the prize collecting steiner tree problem and the maximum-weight connected subgraph problem. *Networks*, 73(2):206–233, 2019. `doi:10.1002/net.21857`.

**23** Urban Road Networks. Urban road network data, January 2016. `doi:10.6084/m9.figshare.2061897.v1`.

**24** Gabriel Robins and Alexander Zelikovsky. Improved steiner tree approximation in graphs. In *Proc. of the 11th Symposium on Discrete Algorithms (SODA 2000)*, pages 770–779, 2000.

**25** Gabriel Robins and Alexander Zelikovsky. Tighter bounds for graph steiner tree approximation. *SIAM J. Discrete Math.*, 19(1):122–134, 2005.

**26** Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: `http://networkrepository.com`.

**27** Hiromitsu Takahashi. An approximate solution for the steiner problem in graphs. *Math. Japonica.*, 6:573–577, 1990.

**28** Eduardo Uchoa, Marcus Poggi de Aragão, and Celso C. Ribeiro. Preprocessing steiner problems from VLSI layout. *Networks*, 40(1):38–50, 2002. `doi:10.1002/net.10035`.

**29** Alexander Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5):463–470, 1993.

**30** Alexander Zelikovsky. Better approximation bounds for the network and euclidean steiner tree problems. *University of Virginia, Charlottesville, VA*, 1996.

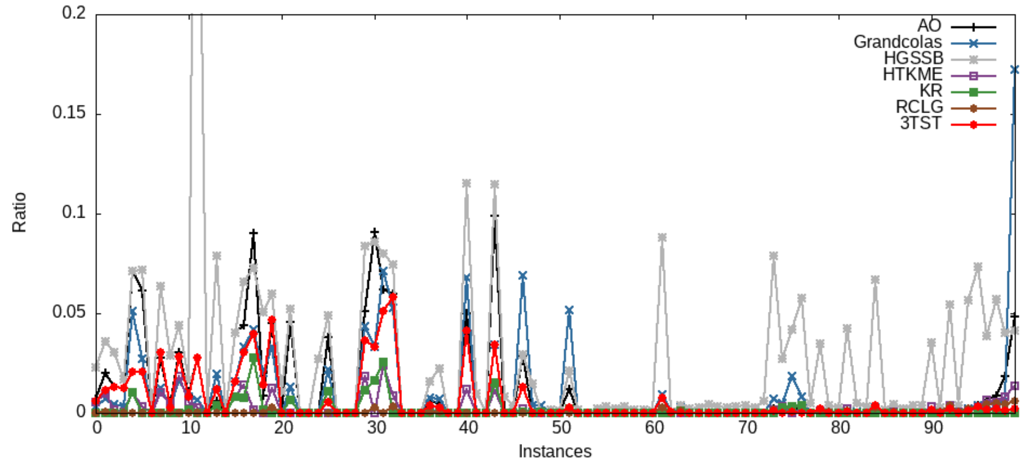


**Figure 3** PACE Challenge: Show the ratio obtained by each solver on each instance of the PACE Challenge data set. Instances are sorted by increasing number of vertices.

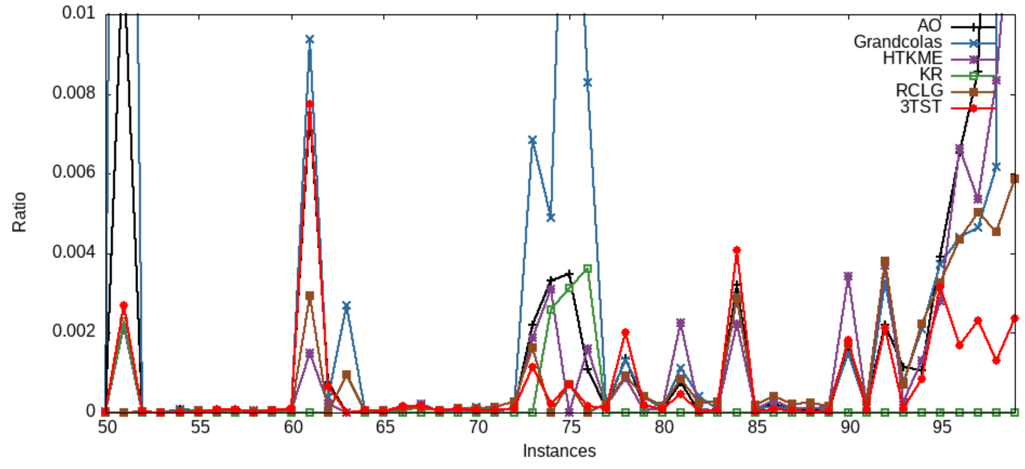


**Figure 4** PACE Challenge: Show the ratio obtained by each solver on the 50 largest instances of the PACE Challenge data set. Instances are sorted by increasing number of vertices.

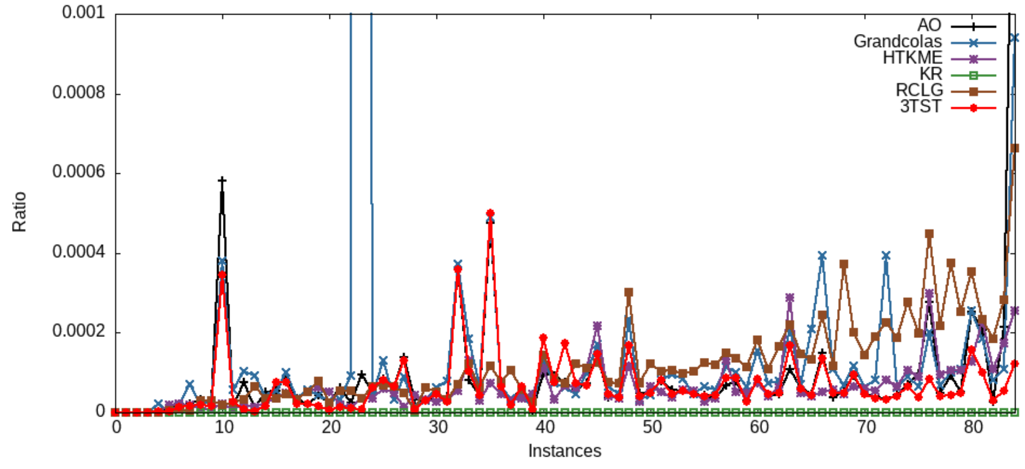


**Figure 5** I simple: Show the ratio obtained by each solver on each instance of the I simple preprocessed instances data set. Instances are sorted by increasing number of vertices.

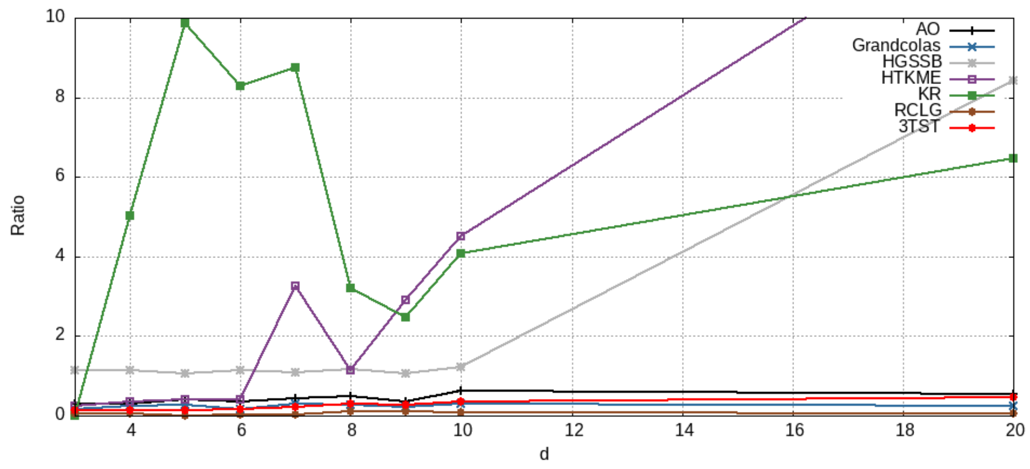


**Figure 6** *d*-regular: Show the ratio obtained by each solver on *d*-regular random graph. Show the evolution of the ratio with respect to increasing values of *d*.

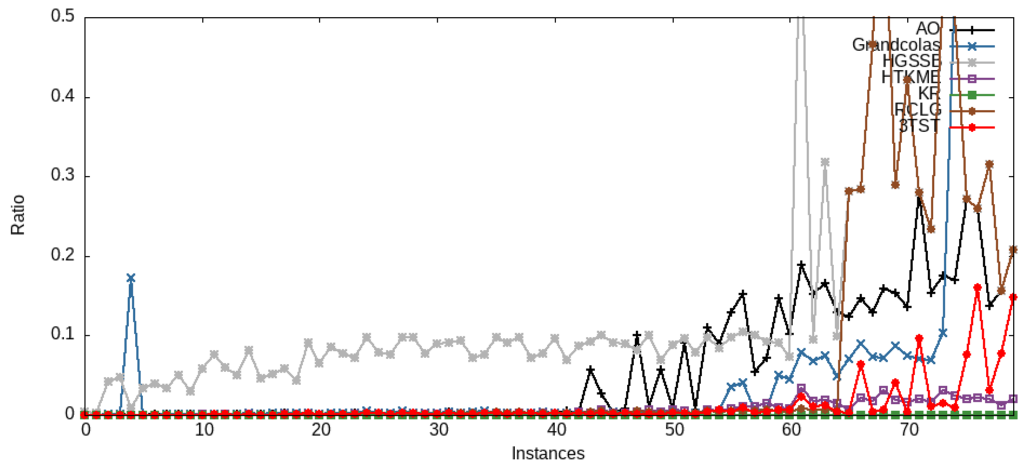


**Figure 7** City road networks: Show the ratio obtained by each solver on each instance of the City road networks data set. Instances are sorted by increasing number of vertices.

# A Fast and Tight Heuristic for A* in Road Networks

**Ben Strasser**
Stuttgart, Germany

**Tim Zeitz**
Karlsruhe Institute of Technology, Germany

## Abstract

We study exact, efficient and practical algorithms for route planning in large road networks. Routing applications often require integrating the current traffic situation, planning ahead with traffic predictions for the future, respecting forbidden turns, and many other features depending on the exact application. While Dijkstra's algorithm can be used to solve these problems, it is too slow for many applications. A* is a classical approach to accelerate Dijkstra's algorithm. A* can support many extended scenarios without much additional implementation complexity. However, A*'s performance depends on the availability of a good heuristic that estimates distances. Computing tight distance estimates is a challenge on its own. On road networks, shortest paths can also be quickly computed using hierarchical speedup techniques. They achieve speed and exactness but sacrifice A*'s flexibility. Extending them to certain practical applications can be hard. In this paper, we present an algorithm to efficiently extract distance estimates for A* from Contraction Hierarchies (CH), a hierarchical technique. We call our heuristic CH-Potentials. Our approach allows decoupling the supported extensions from the hierarchical speed-up technique. Additionally, we describe A* optimizations to accelerate the processing of low degree nodes, which often occur in road networks.

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

**Keywords and phrases** route planning, shortest paths, realistic road networks

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.6

**Supplementary Material** *Software (Source Code)*: `https://github.com/kit-algo/ch_potentials` archived at `swh:1:dir:816552e5789f95afc8d6a2afe5b3dc40de537165`

## 1 Introduction

The past decade has seen a plethora of research on route planning in large street networks [1]. Routing a user through a road network can be formalized as the shortest path problem in weighted graphs. Nodes represent intersections. Roads are modeled using edges. Edges are weighted by their traversal times. The problem can be solved with Dijkstra's algorithm [20]. Unfortunately, on continental sized networks, it is too slow for many applications. Thus, speed-up techniques have been developed. One popular example are Contraction Hierarchies (CH) [25]. They have been used successfully in many real world applications. A CH exploits the inherent hierarchy of road networks. In a preprocessing step, additional shortcut edges are inserted, which allow skipping unimportant parts of the network at query time. Another popular example is Multi-Level-Dijkstra (MLD) [38] also known as CRP [14]. It is also used in practice [35]. MLD also uses shortcut edges. Both approaches achieve speed-ups of at least three orders of magnitude over Dijkstra's algorithm.

Unfortunately, for many real world applications, this basic graph model is too simplistic. For realistic routing, many additional features need to be considered. This includes turn costs and restrictions, live traffic, user preferences, and traffic predictions. Some applications may have additional application-specific requirements. Extending Dijkstra's algorithm to support

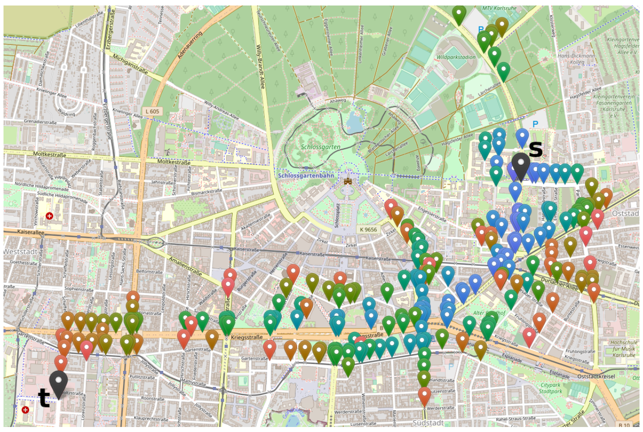


**Figure 1** Nodes explored by A*. Color indicates the node removal order from the queue. Blue was removed first. Next is green. Red was removed last.

these features is usually easy. Extending hierarchical speed-up techniques is also possible. However, the algorithm development is vastly more complex. For every feature, dedicated research paper(s) exist that extend CH. Supporting the combination of several features is even harder. For example, we are not aware of any work combining all features mentioned above. In this paper, we describe an algorithmic building block, that allows handling the combination of all above mentioned features – and probably more.

Our approach decouples extensions from the hierarchical speed-up technique by utilizing the A* algorithm [31]. A* is a goal-directed variant of Dijkstra's algorithm. See Figure 1 for an example of nodes traversed during an A* search. A* uses a *heuristic* to guide the search towards the goal. A heuristic is function that maps a node $v$ onto an estimate of the distance from $v$ to the goal. A*'s running time crucially depends on how tight this estimate is. Further, evaluating the heuristic must be fast. In this paper, we describe CH-Potentials, a fast heuristic with tight estimates. Internally, the heuristic uses a CH. Fortunately, this is an implementation detail from the perspective of the A*. To support a new feature, we only need to modify the A* algorithm. The heuristic core containing the CH remains untouched. Extending A* is vastly easier than extending a CH. This enables us to design algorithms for a multitude of features. In addition, we describe query optimizations for handling of low-degree nodes, common in road networks. These low degree optimizations are applicable to Dijkstra's algorithm and A*.

The rest of the paper is organized as follows. In Section 2, we discuss related works on goal directed search and extensions for realistic applications for hierarchical techniques. CH-Potentials, our new distance estimation function is introduced in Section 3. Section 4 discusses our improvements for the handling of low-degree nodes. In Section 5, we demonstrate CH-Potential's flexibility, by describing how to apply the approach to different practical applications. Finally, in Section 6, we present an experimental evaluation of our approach.

## 2 Related Work

There is a lot of work that extends hierarchical speed-up techniques to more complex settings [1]. For example, in [26] turn information is integrated into CH. A considerable amount of research and engineering effort has been put into studying the combination of traffic predictions with CH. Several papers [3, 4, 32, 5] and an entire dissertation [2] have been published on the subject. Different variants with trade-offs regarding exactness, query speed and space consumption were proposed [5]. Recently, a new approach has been published [40] which simultaneously achieves competitive results in all three aspects but only at the cost of considerable implementation complexity. CRP (Customizable Route Planning) [14] is an engineered variant of MLD [38] which was developed to allow updating weights without invalidating the entire preprocessing. For this, a faster, second preprocessing phase is introduced. It can be run regularly to update weights. In theory, this enables the integration of live traffic and user preferences. In practice, live traffic feed data is imperfect. Computing "good" routes without undesired detours due to artifacts in the data requires additional algorithmic extensions [16]. CRP also supports turn costs. Integrating traffic predictions into CRP was studied in [9]. On continental sized networks, TD-CRP can only compute approximate shortest distances (rather than paths). In [19], CH is extended to Customizable CH (CCH). CCH also has a second preprocessing phase where weights can be altered. Supporting turn costs in CCH was studied in [11]. Other extensions studied include electric vehicle routing [8, 22] or multi-criteria optimization [23, 24]. While these works show that it is possible to extend hierarchical approaches, they also show that it is non-trivial. Further, in every extension the flexibility available at query time is fairly limited. Combining these hierarchical extensions is an unsolved problem.

CH-Potentials is not the first work to combine hierarchical approaches and A* [6, 28, 7]. However, previous works mostly focused on accelerating hierarchical approaches further rather than exploiting A*'s flexibility.

ALT [27, 29] and CPD-Heuristics [10] are the two techniques with high conceptual similarity to CH-Potentials. ALT has been combined with shortcuts [6] and also extended for dynamic graphs [17] and time-dependent routing [36, 15]. CPD-Heuristics are a combination of A* and Compressed Path Databases (CPD). A CPD can quickly compute the first edge of a shortest path between any two nodes. In [10], SRC [39] is used as CPD. For every distance estimation, a shortest path to the target is computed, whose length is used as the heuristic value. Unfortunately, the employed CPD's quadratic preprocessing running time is problematic on large street networks. In [12] the weighted graph is embedded into Euclidean space using FastMap. The Euclidean distance is then used as a distance estimate for A*.

## 3 Algorithm

In this section, we first discuss the framework in which CH-Potentials can be used. Then, we describe the building blocks of CH-Potentials: Contraction Hierarchies and PHAST, a CH extension. Finally, we introduce the CH-Potentials heuristic.

### 3.1 Formal Setup: Inputs, Outputs, and Phases

In this paper, we consider different applications, with slightly different problem models. The goal is always to quickly answer many shortest path queries. For the purpose of describing our framework, we establish a shared notation: Input to each query are nodes $s$ and $t$, and a graph $G_q$ with query weights $w_q$. However, the precise formal inputs of the query and what exactly $w_q$ represents depends on the application. In the simplest case, $w_q$ will be scalar edge weights. However, this is not a requirement. It can be any function that computes a weight for an edge. This function can also take additional parameters from the state of the search. For example, in the case of live-traffic, $w_q$ represents scalar edge weights. However, values of $w_q$ might change between queries. In the case of traffic predictions, $w_q$ is a function which maps the edge entry time to the traversal time and the query takes an additional departure time parameter.

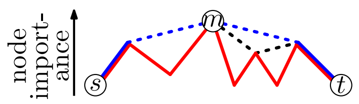


**Figure 2** Solid lines are edges in $G$. Dotted lines are shortcuts. Red is shortest $st$-path in $G$. Blue is equaly long up-down $st$-path in $G^+$. $m$ is the mid node.

To enable quick shortest path computations, we consider a two phase setup with an additional off-line preprocessing phase before the on-line query phase. The input to the preprocessing phase is a graph $G_\ell$ with lower bound weights $w_\ell$ and a node mapping function $\phi$. $w_\ell(e)$ must be a scalar value for every edge $e$ of $G_\ell$. We require that $w_q(u, v)$ is greater or equal to the shortest distance $\text{dist}_\ell(\phi(u), \phi(v))$ from $\phi(u)$ to $\phi(v)$ in $G_\ell$. The output of the preprocessing is auxiliary data that enables an efficient heuristic function $h_t(x)$. $h_t(x)$ is the exact distance from $\phi(x)$ to $\phi(t)$ in $G_\ell$. In the applications considered in this paper, $w_\ell$ is always the freeflow travel time.

The query phase uses this heuristic in an A* search between nodes $s$ and $t$ on $G_q$ and $w_q$. The exact implementation of this A* search depends on the application. Our approach only provides the heuristic $h_t$ for the A* search. In contrast, the preprocessing phase remains the same for all applications.

Our heuristic is always *feasible* [31], i.e. $w_q(u, v) - h_t(u) + h_t(v) \geq 0$ holds for all edges. By requirement and because of the triangle inequality the following holds:

$$w_q(u, v) - h_t(u) + h_t(v) \geq \text{dist}_\ell(\phi(u), \phi(v)) - \text{dist}_\ell(\phi(u), \phi(t)) + \text{dist}_\ell(\phi(v), \phi(t)) \geq 0$$

Thus, A* will always determine the correct shortest distances.

## 3.2 Contraction Hierarchy (CH)

**Algorithm 1** CH backward search.

```
Data: $B[x]$: tentative distance from $x$ to target $t$
Data: Min. priority queue $Q$, also called open list
$B[x] \leftarrow +\infty$ for all $x \neq t$; $B[t] \leftarrow 0$;
Make $Q$ only contain $t$ with weight 0;
while not $Q$ empty do
    $y \leftarrow$ pop minimum element from $Q$;
    for $xy$ is down-edge in $G_\ell^+$ do
        if $B[x] > w_\ell(xy) + B[y]$ then
            $B[x] \leftarrow w_\ell(xy) + B[y]$;
            Add $x$ or decrease $x$'s key in $Q$ to $B[x]$;
```

A CH is a two phase technique to efficiently compute exact, shortest paths. For details, we refer to [25, 19]. In this section, we give an introduction.

A CH places nodes into levels. No edge must connect two nodes within one level. Levels are ordered by "importance". The intuition is that dead-ends are unimportant and at the bottom while highway bridges are very important and at the top. An edge goes *up* when it goes from a node in a lower level to higher level. *Down* edges are defined analogously. An *up-down path* is a path where only one node $m$ is more important than both its neighbors. $m$

is called the *mid* node. An *up path* is a path where the last node is the mid node. Similarly, the first node is the mid node of a *down path*. In the preprocessing phase, a CH adds *shortcut* edges to the input graph $G$ to obtain $G^+$. This is done by repeatedly contracting unimportant nodes and adding shortcuts between its neighbors. After the preprocessing, for every pair of nodes $s$ and $t$ there exists a shortest up-down $st$-path in $G^+$ with the same length as a shortest path in $G$. See Figure 2 for a proof sketch. From every shortest path (red) in $G$, an up-down path of equal length in $G+$ (blue) exists. Thus, we can restrict our search to up-down paths in $G^+$. The search is bidirectional. The forward search starts from $s$ and only follows up-edges. Similarly, the backward search starts at $t$ and only follows down-edges in reversed direction. The two searches meet at the mid node. Pseudo-code for the backward search, i.e., the path from $m$ to $t$, is presented in Algorithm 1. The forward search works analogously. A CH query is fast, if the number of nodes reachable via only up- or down-nodes is small. On road networks, this is the case [25, 14]. On graphs with low treewidth, this is also the case [19, 30].

Using the CH query algorithm, we can already give a simple heuristic. The heuristic evaluation $h_t(x)$ performs a CH-query from $x$ to $t$. This yields tight estimates but a high overhead for the heuristic evaluation. While a single CH query is fast, answering one for every node explored in the $A^*$ search is slow. Fortunately, we can do better.

## 3.3 PHAST based Heuristic

**Algorithm 2** PHAST basic all-to-one search.

```
Data: $P[x]$: tentative distance from $x$ to $t$
Execute Algorithm 1;
for all CH levels $L$ from most to least important do
    for all up edges $xy$ in $G^+_\ell$ with $x$ in $L$ do
        if $P[x] < P[y] + w_\ell(xy)$ then
            $P[x] \leftarrow P[y] + w_\ell(xy)$;
```

PHAST [13] is a CH extension that computes distances from all nodes to one target node. The preprocessing phase remains unchanged. The query phase is split into two steps. The first step is analogue to the CH query: From $t$, all reachable nodes via reversed down-edges are explored. Algorithm 1 shows this first step. The second step iterates over all CH levels from top to bottom. In each iteration, all up-edges starting within the current level are followed in reverse. After all levels are processed, the shortest distances from all nodes to $t$ were computed. Pseudo-code is provided in Algorithm 2. Using PHAST, we can also compute a tight A* heuristic. In the query phase, we first run PHAST to compute the distances from every node to $t$ with respect to $w_\ell$ and store the result in an array $H$. Next, we run $A^*$ and implement the heuristic as a lookup in the array $H$.

The $H$ lookup and by extension the $A^*$ search is indeed fast. However, the PHAST step before the search is comparatively expensive. The reason is that the distances towards $t$ are computed for *all* nodes. Ideally, we only want to compute the distances from the nodes explored in the $A^*$ search.

**Algorithm 3** CH-Potentials Algorithm.

```
Data: $B[x]$: tentative distance from $x$ to $t$ as computed by Algorithm 1
Data: $P[x]$: memoized potential at $x$, $\perp$ initially
Function Pot($x$):
    if $P[x] = \perp$ then
        $P[x] \leftarrow B[x]$;
        for all up edges $xy$ in $G_\ell^+$ do
            $P[x] \leftarrow \min\{P[x], w_\ell(xy) + \texttt{Pot}(y)\}$;
    return $P[x]$;
```

## 3.4 CH-Potentials

Fortunately, the PHAST computation can be done lazily using memoization as depicted in Algorithm 3. In a first step, we run the backward CH search from $t$ to obtain an array $B$. $B[x]$ is the minimum down $xt$-path distance or $+\infty$, if there is no such path. $B$ is computed as shown in Algorithm 1.

To compute the heuristic $h_t(x)$, we recursively compute for all up-edges $(x, y)$ the heuristic $h_t(y)$. Next, we compute the minimum distance over all up-down paths that contain at least one up-edge using $d = \min_y\{w_\ell(x, y) + h_t(y)\}$. As not all shortest up-down paths contain an up-edge, we set $h_t(x) = \min\{B[x], d\}$. This calculation is correct, as it computes the minimum up-down $xt$-path distance in $G_\ell^+$, which corresponds to the minimum $xt$-path distance in a CH. A* with this heuristic is the basic CH-Potentials algorithm.

## 4 Low Degree A* Improvements

Preliminary experiments showed, that a significant amount of query running time is spent in heuristic evaluations and queue operations. We can reduce both by keeping some nodes out of the queue, as the heuristic needs to be evaluated when a node is pushed into the queue. Avoiding pushing low degree nodes into the queue is the focus of this section. The techniques discussed here are a lazy variant of the ideas used in TopoCore [18].

We modify A* by processing low degree nodes consecutively without pushing them into the queue. Our algorithm uses the undirected degree $d(x)$ of a node $x$. Formally, $d(x)$ is the number of nodes $y$ such that $(x, y) \in E$ or $(y, x) \in E$.

Analogous to A*, our algorithm stores for every node $x$ a tentative distance $D[x]$. Additionally, it maintains a minimum priority queue. Diverging from A*, not all nodes can be pushed but every node has a tentative distance.

## 4.1 Skip Degree Two Nodes

Our algorithm differs from A* when removing a node $x$ from the queue. A* iterates over the outgoing arcs $(x, y)$ of $x$ and tries to reduce $D[y]$ by relaxing $(x, y)$. If A* succeeds, $y$'s weight in the queue is set to $D[y] + h_t(y)$. Our algorithm, however, behaves differently, if $d(y) \leq 2$. Our algorithm determines the longest degree two chain of nodes $x, y_1, \dots, y_k, z$ such that $d(y_i) = 2$ and $d(z) > 2$. If our algorithm succeeds in reducing $D[y_1]$, it does not push $y_1$ into the queue. Instead, it iteratively tries to reduce all $D[y_i]$. If it does not reach $z$, then only $D$ is modified but no queue action is performed. If $D[z]$ is modified and $d(z) > 2$, $z$'s weight in the queue is set to $D[z] + h_t(z)$.

As the target node $t$ might have degree two, our algorithm cannot rely on stopping, when $t$ is removed from the queue. Instead, our algorithm stops as soon as $D[t]$ is less than the minimum weight in the queue.

## 4.2 Skip Degree Three Nodes

We can also skip some degree three nodes. Denote by $x, y_1, \dots, y_k, z$ a degree two chain as described in the previous section. If $d(z) > 3$ or $z$ is in the queue, our algorithm proceeds as in the previous section. Otherwise, there exist up to two degree chains $z, a_1, \dots, a_p, b$ and $z, \alpha_1, \dots, \alpha_q, \beta$ such that $a_1 \neq y_k \neq \alpha_1$. Our algorithm iteratively tries to reduce all $D[a_i]$ and $D[\alpha_i]$. If it reaches $\beta$, $\beta$'s weight in the queue is set to $D[\beta] + h_t(\beta)$. Analogously, if $b$ is reached, $b$'s weight is set to $D[b] + h_t(b)$. If $b$ respectively $\beta$ are not reached, our algorithm does nothing.

## 4.3 Stay in Largest Biconnected Component

A lot of nodes in road networks lead to dead-ends. Unless the source or target is in this dead-end, it is unnecessary to explore these nodes.

In the preprocessing phase, we compute the subgraph $G_C$, called *core*. $G_C$ is induced by the largest biconnected component of the undirected graph underlying $G$. We do this using Tarjan's algorithm [43]. For every node $v$ in the input graph $G$, we store the attachment node $a_v$ to the core. For nodes in the core, $a_v = v$. We exploit that all attachment nodes are single node separators and the problem can be decomposed along them.

The query phase is divided into two steps. In the first step, we apply A* with CH-Potentials to $G_C$ combined with the component that contains $s$. This can be achieved implicitly by removing edges from $G_C$ into other components during preprocessing. If $t$ is part of $G_C$ or in the same component as $s$, this A* search finds it. Otherwise, we find $a_t$. In that case, we continue by searching a path from $a_t$ towards $t$ restricted to $t$'s biconnected component. The final result is the concatenation of both paths.

## 5 Applications

We describe some extended routing problems and how to apply CH-Potentials to them. Unless stated otherwise, $G_q$ and $G_\ell$ are the same graph and only $w_q$ changes for the queries.

## 5.1 Avoiding Tunnels and/or Highways

Avoiding tunnels and/or highways is a common feature of navigation devices. Implementing this feature with CH-Potentials is easy. We set $w_\ell$ to the freeflow travel time. If an edge is a tunnel and/or a highway, we set $w_q$ to $+\infty$. Otherwise, $w_q$ is set to the freeflow travel time.

## 5.2 Forbidden Turns and Turn Costs

The classical shortest path problem allows to freely change edges at nodes. However, in the real world, turn restrictions, such as a forbidden left or right turn, exist. Also, taking a left turn might take longer than going straight. This can be modeled using turn weights [26, 14, 11]. A *turn weight* $w_t$ maps a pair of incident edges onto the turning time or $+\infty$ for forbidden turns. For CH-Potentials, we use zero as lower bound for every turn weight in the heuristic. Thus, the graph $G_\ell$ and weights $w_\ell$ for preprocessing is the unmodified input graph without turn weights.

A path with nodes $v_1, v_2, \dots v_k$ has the following *turn-aware weight*:

$$w_\ell(v_1, v_2) + \sum_{i=2}^{k-1} w_t(v_{i-1}, v_i, v_{i+1}) + w_\ell(v_i, v_{i+1})$$

The objective is to find a path between two given edges with minimum turn-aware weight. The first term $w_\ell(v_1, v_2)$ is the same for all paths, as it only depends on the source edge. It can thus be ignored during optimization.

We solve this problem by constructing a *turn-expanded* graph as $G_q$. Edges in the input graph $G_\ell$ correspond to *expanded nodes* in $G_q$. For every pair of incident edges $(x, y)$ and $(y, z)$ in $G_\ell$, there is an *expanded edge* in $G_q$ with expanded weight $w_t(x, y, z) + w_\ell(y, z)$. A sequence of expanded nodes in the expanded graph $G_q$ corresponds to a sequence of edges in the input graph $G_\ell$. The weight of a path in $G_q$ is equal to the turn-aware weight of the corresponding path in $G_\ell$ minus the irrelevant $w_\ell(v_1, v_2)$ term. Thus, the turn-aware routing problem can be solved by searching for shortest paths in $G_q$.

In this scenario, preprocessing and query use different graphs $G_\ell$ and $G_q$. We define the node mapping function $\phi$ as $\phi(x, y) = y$. Obviously, $w_q(xy, yz) = w_t(x, y, z) + w_\ell(y, z) \geq \text{dist}_\ell(\phi(x, y), \phi(y, z))$ and this approach yields a feasible heuristic. Sadly, the undirected graph underlying $G_q$ is always biconnected, if the input graph is strongly connected. The optimization described in Section 4.3 is therefore ineffective. With this setup, CH-Potentials support turns without requiring turn information in the CH.

## 5.3 Predicted Traffic or Time-Dependent Routing

The classical shortest path problem assumes that edge weights are scalars. However, in practice, travel times vary along an edge due to the traffic situation. Recurring traffic can be predicted by observing the traffic in the past. It is common [5, 9, 40] to represent these predictions as *travel time functions*. An edge weight is no longer a scalar value but a function that maps the entry time onto the traversal time.

In this setting, the query weight $w_q$ is a function from $E \times \mathbb{R}$ to $\mathbb{R}^+$. $w_q(e, \tau)$ is the travel time through edge $e$ when entering it at moment $\tau$. The input to the extended problem consists of a source node $s$ and a target node $t$, as in the classical problem formulation. Additionally, the input contains a source time $\tau_s$. A path with edges $e_1, e_2 \dots e_k$ is weighted using $\alpha_k$, which is defined recursively as follows:

$$\alpha_1 = w_q(e_1, \tau_s)$$
$$\alpha_k = \alpha_{k-1} + w_q(e_1, \alpha_{k-1})$$

The objective is to find a path to $t$ that minimizes $\alpha_k$.

If all travel time functions fulfill the *FIFO property*, this problem can be solved using a straight forward extension of Dijkstra's algorithm [21]. The necessary modification to A* is analogous. Without the FIFO property the problem becomes NP-hard [37]. The FIFO property states that it is not possible to arrive earlier by departing later. Formally stated, the following must hold $\forall e \in E, \tau \in \mathbb{R}, \delta \in \mathbb{R}^+ : w_q(e, \tau) \leq w_q(e, \tau + \delta) + \delta$. Our implementation stores edge travel times using piece-wise linear functions. The A* search uses the tentative distance $\tau$ at a node $x$ when to evaluating the travel time of outgoing edges $(x, y)$. This strategy is very similar to TD-ALT [36, 17].

For the preprocessing, we set $w_\ell(e) = \min_\tau w_q(e, \tau)$, that is the minimum travel time. By keeping travel time functions out of the CH, we avoid a lot of algorithmic complications compared to [5, 9, 40, 15] which have to create shortcuts of travel time functions.

## 5.4 Live and Predicted Traffic

Beside predicted traffic, we also consider live traffic. Live traffic refers to the current traffic situation. It is important to distinguish between predicted and live traffic. Live traffic data is more accurate for the current moment than predicted data. It is possible that it differs significantly from predicted traffic, if unexpected events like accidents happen. However, just using live traffic data is problematic for long routes as traffic changes while driving. At some point, one wants to switch from live traffic to the predicted traffic. In this section, we first describe a setup with only live traffic and then combine it with predicted traffic.

To support only live traffic, we set $w_\ell$ to the freeflow travel time. $w_q$ is set to the travel time accounting for current traffic. As traffic only increases the travel time along an edge, $w_\ell$ is a valid lower bound for $w_q$. In a real world application, values from $w_q$ could be updated between queries. This is all that is necessary to apply CH-Potentials in a live traffic scenario.

To combine live traffic with predicted traffic, we define a modified travel time function $w_q$ that is then used as query weights. Denote by $w_p(e, \tau)$ the predicted travel time along edge $e$ at moment $\tau$. Further, $w_c(e)$ is the travel time according to current live traffic. Finally, we denote by $\tau_{\text{soon}}$ the moment when we switch to predicted traffic. In our experiments, we set $\tau_{\text{soon}}$ to one hour in the future. We need to make sure that the modified travel time function fulfills the no-waiting property. For this reason, we cannot make a hard switch at $\tau_{\text{soon}}$. Our modified travel time function linearly approaches the predicted travel time. Formally, we set $w_q(e, \tau)$ to $w_c(e)$, if $\tau \leq \tau_{\text{soon}}$. Otherwise, we check whether $w_p(e, \tau_{\text{soon}}) < w_c(e)$ is true. If it is the case, we set $w_q(e, \tau)$ to $\max\{w_c(e) + (\tau_{\text{soon}} - \tau), w_p(e, \tau)\}$. Otherwise, we set $w_q(e, \tau)$ to $\min\{w_c(e) - (\tau_{\text{soon}} - \tau), w_p(e, \tau)\}$. In our implementation, we to not modify the representation of $w_p$ but evaluate the formulas above at each travel time evaluation. We set $w_\ell$ again to the freeflow travel time.

With this setup, CH-Potentials support a combination of live and predicted traffic. We did not make any modification, that would hinder a combination with other extensions. Further adding tunnel and/or highway avoidance or turn-aware routing is simple. This straight-forward integration of complex routing problems is the strength of the CH-Potentials.

### 5.4.1 Three-Phase Setups

Supporting live traffic is also possible with a three-phase setup: A slow preprocessing phase, a faster *customization* phase, and fast queries. The customization phase is run regularly and incorporates updates to the weights into the auxiliary preprocessing data. CRP [14] and CCH [19] follow this setup. Luckily, a CCH is just a CH with some additional properties. The CH in CH-Potentials can be replaced by a CCH without further modifications. Thus, CCH-Potentials could also support a three-phase setup. However, evaluating CCH-Potentials is beyond the scope of this paper. We focus on evaluating CH-Potentials as a simple building block in the two-phase setup.

## 5.5 Temporary Driving Bans

Truck routing differs from car routing due to night driving bans and other restrictions. In [33], a preliminary version of CH-Potentials[1] is used for such a scenario. The work considers

[1] In [33], CH-Potentials are used as a blackbox referring to an early ArXiv preprint [41] of ours. Our submitted paper is the finished version of the ArXiv preprint. [33] does not describe any of the contributions of this paper.

**Table 1** Instances used in the evaluation.

| | Nodes $[\cdot 10^6]$ | Edges $[\cdot 10^6]$ | Preprocessing [s] |
|---|---|---|---|
| OSM Ger | 16.2 | 35.4 | 295.2 |
| TDEur17 | 25.8 | 55.5 | 292.6 |
| TDGer06 | 4.7 | 10.8 | 58.9 |

time-dependent blocked edges and waiting at parking locations. Further, a trade-off between arrival time and route quality is considered.

## 6 Evaluation

In this section, we present our experimental evaluation. Our benchmark machine runs openSUSE Leap 15.1 (kernel 4.12.14), and has 128 GiB of DDR4-2133 RAM and an Intel Xeon E5-1630 v3 CPUs which has four cores clocked at 3.7 Ghz and 4 × 32 KiB of L1, 8 × 256 KiB of L2, and 10 MiB of shared L3 cache. All experiments were performed sequentially. Our code is written in Rust and compiled with rustc 1.47.0-nightly in the release profile with the target-cpu=native option. The source code of our implementation and the experimental evaluation can be found on Github[2].

**Inputs and Methodology.** Our main benchmark instance is a graph of the road network of Germany obtained from Open Street Map[3]. To obtain the routing graph, we use the import from RoutingKit[4]. The graph has 16M nodes and 35M edges. For this instance, we have proprietary traffic data provided by Mapbox[5]. The data includes a live traffic snapshot from Friday 2019/08/02 afternoon and comes in the form of 320K OSM node pairs and live speeds for the edge between the nodes. It also includes traffic predictions for 38% of the edges as predicted speeds for all five minute periods over the course of a week. We exclude speed values which are faster than the freeflow speed computed by RoutingKit. Additionally, we have two graphs with proprietary traffic predictions provided by PTV[6]. The PTV instances are not OSM-based. One is an old instance of Germany with traffic predictions from 2006 for 7% of the edges and the other one a newer instance of Europe with predictions for 27% of the edges. Table 1 contains an overview over our instances. In this table, we further include the sequential running time necessary to construct the CH. We report preprocessing running times as averages over 10 runs. For queries, we perform 10 000 point-to-point queries where both source and target are nodes drawn uniformly at random and report average results.

**Experiments.** The performance of A* depends on the tightness of the heuristic. CH-Potentials computes optimal distance estimates with respect to $w_\ell$. However, for most applications, there will be a gap between $w_q$ and $w_\ell$ (otherwise one could use CH without A*). We evaluate the impact of the difference between $w_q$ and $w_\ell$ on the performance of A*.

[2] `https://github.com/kit-algo/ch_potentials`
[3] `https://download.geofabrik.de/europe/germany-200101.osm.pbf`
[4] `https://github.com/RoutingKit/RoutingKit`
[5] `https://mapbox.com`
[6] `https://ptvgroup.com`

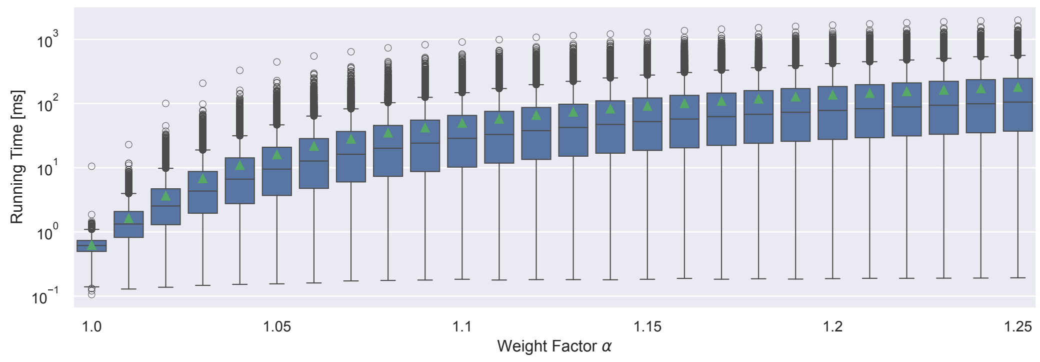


**Figure 3** Running times on a logarithmic scale for queries on OSM Ger with scaled edge weights $w_q = \alpha \cdot w_\ell$. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the diamond the mean. The whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

**Table 2** Average query running times and number of queue pushs with different heuristics and optimizations on OSM Ger with $w_q = 1.05 \cdot w_\ell$.

| | BCC | Deg2 | Deg3 | Zero | ALT | CH-Pot. | Oracle |
|---|---|---|---|---|---|---|---|
| Running time [ms] | ✗ | ✗ | ✗ | 1 947.4 | 279.5 | 50.6 | 32.0 |
| | ✓ | ✗ | ✗ | 1 253.1 | 217.3 | 36.3 | 23.9 |
| | ✓ | ✓ | ✗ | 713.6 | 117.3 | 18.8 | 11.8 |
| | ✓ | ✓ | ✓ | 558.9 | 88.3 | 15.7 | 9.5 |
| Queue pushs [$\cdot 10^3$] | ✗ | ✗ | ✗ | 8 120.7 | 859.4 | 138.0 | 138.0 |
| | ✓ | ✗ | ✗ | 6 326.5 | 684.1 | 114.0 | 114.0 |
| | ✓ | ✓ | ✗ | 2 915.7 | 301.3 | 42.1 | 42.1 |
| | ✓ | ✓ | ✓ | 1 689.8 | 178.4 | 26.0 | 26.0 |

The lower bound $w_\ell$ is set to the freeflow travel time. The query weights $w_q$ are set to $\alpha \cdot w_\ell$, where $\alpha \geq 1$. Increasing $\alpha$ degrades the heuristic's quality. Figure 3 depicts the results. Clearly, $\alpha$ has significant influence on the running time. Average running times range from below a millisecond to a few hundred milliseconds depending on $\alpha$. Up to around $\alpha = 1.1$ the running time grows quickly. For $\alpha > 1.1$, the growth slows down. This illustrates both the strengths and limits of our approach and goal directed search in general. CH-Potentials can only achieve competitive running times if the application allows for a sufficiently tight lower bounds at preprocessing time.

We observe that the running times for a fixed $\alpha$ vary strongly. This is an interesting observation, as with uniform source and target sampling, nearly all queries are long-distance. The query distance is thus not the reason. After some investigation, we concluded that this is due to non-uniform road graph density. Some regions have more roads per area than others. The number explored A* nodes depends on the density of the search space area. As the density varies, the running times vary.

Table 2 depicts the performance of A* with different heuristics and optimizations. We compare CH-Potentials to three other heuristics. First, the Zero heuristic where $h(x) = 0$ for all nodes $x$. This corresponds to using Dijkstra's algorithm. Second, we compare against our own implementation of ALT [29]. We use 16 landmarks generated with the avoid strategy [29]

**Table 3** CH-Potentials performance for different route planning applications. We report average running times and number of queue pushes. We also report the average length increase, that is how much longer the final shortest distance is compared to the lower bound. Finally, we report the average running time of Dijkstra's algorithm as a baseline and the speedup over this baseline.

| | | Running time [ms] | Queue [$\cdot 10^3$] | Length incr. [%] | Dijkstra [ms] | Speedup |
|---|---|---|---|---|---|---|
| OSM Ger | Unmodified ($w_q = w_\ell$) | 0.6 | 0.5 | 0.0 | 1 952.8 | 3 243.1 |
| | Turns | 2.8 | 6.0 | 1.1 | 4 244.4 | 1 540.8 |
| | No Tunnels | 25.9 | 40.7 | 5.3 | 1 990.1 | 76.9 |
| | No Highways | 342.9 | 518.5 | 42.4 | 1 843.9 | 5.4 |
| | Live | 127.3 | 192.1 | 14.8 | 1 884.1 | 14.8 |
| | TD | 195.5 | 163.1 | 17.6 | 3 186.2 | 16.3 |
| | TD + Live | 209.7 | 179.1 | 21.4 | 3 152.2 | 15.0 |
| | TD + Live + Turns | 508.5 | 765.5 | 22.7 | 6 179.5 | 12.2 |
| TDEur17 | TD | 89.7 | 81.5 | 3.9 | 3 479.9 | 38.8 |
| TDGer06 | TD | 4.5 | 6.4 | 3.1 | 602.4 | 135.4 |

and activate all during every query. Our ALT implementation is uni-directional. In this work, we do not consider bidirectional search as it creates problems for some settings, such as predicted traffic. Finally, we compare against a hypothetical *Oracle-A** heuristic. This heuristic has instant access to a shortest distance array with respect to $w_\ell$, i.e. it is faster than the fastest heuristic possible in our model. We fill this array before each query using a reverse Dijkstra search from the target node. Thus, the reported running times of Oracle-A* do *not* account for any heuristic evaluation. CH-Potentials compute the same distance estimates but the heuristic evaluation has some overhead. Comparing against Oracle-A* allows us to measure this overhead. Also, no other heuristic, which only has access to the preprocessing weights, can be faster than Oracle-A*.

We observe that the number of queue pushes roughly correlates with running time. Each optimization reduces both queue pushes and running times. All optimizations yield a combined speed-up of around 3. CH-Potentials outperform ALT by a factor of between six and seven and settle correspondingly fewer nodes. This is not surprising, since ALT computes worse distance estimates. In contrast, CH-Potentials already compute exact distances with respect to $w_\ell$. The number of popped nodes is the same for CH-Potentials and Oracle-A*. The only difference between CH-Potentials and Oracle-A* is the overhead of the heuristic evaluation. This overhead leads to a slowdown of around 1.6. Thus, CH-Potentials are already very close to the best possible heuristic in this model. This means that no competing algorithm such as ALT or CPD-Heuristics can be significantly faster.

Table 3 depicts the running times of CH-Potentials in various applications, such as those described in Section 5. We report speedups compared to extensions of Dijkstra's algorithm for each application respectively. We start with the base case where $w_q = w_\ell$. This is the problem variant solved by the basic CH algorithm. CH achieves average query running times of 0.16 ms on OSM Ger. CH-Potentials are roughly four times slower but still achieve a huge speedup of 3243 over Dijkstra. Such large speedups are typical for CH. This shows that CH-Potentials gracefully converges toward a CH in the $w_q = w_\ell$ special case.

In the other scenarios, the performance of CH-Potentials strongly depends on the quality of the heuristic. We measure this quality using the length increase of $w_q$ compared to $w_\ell$.

Forbidding highways results in the largest length increase and in the smallest speedup. The other extreme are turn restrictions. They have only a small impact on the length increase. The achieved speedups are therefore comparable to CH speedups. Mapbox live traffic has a length increase of around 15%, which yields running times of 127 ms. The length increase of Mapbox traffic predictions are about 18%, and results in a running time of 200 ms. The speedup in the predicted scenario is larger than in the live setting, as the travel time function evaluations slow down Dijkstra's algorithm. Combining predicted and live traffic results in a running time only slightly higher than for the predicted scenario. Further adding turn restrictions, increases the running times. This increase is mostly due to the BCC optimization of Section 4.3 becoming ineffective when considering turns. It is not due to the length increase of using turns. With everything activated, our algorithm still has a speedup of 12.2 over the baseline. Interestingly, the PTV traffic predictions have a much smaller length increase than the Mapbox predictions. This results in smaller running times of our algorithm.

## 7 Conclusion

In this paper, we introduced CH-Potentials, a fast, exact, and flexible two-phase algorithm based on A* and CH for finding shortest paths in road networks. The approach can handle a multitude of complex, integrated routing scenarios with very little implementation complexity. CH-Potentials provides *exact* distances with respect to lower bound weights known at preprocessing time as an A* heuristic. Thus, the query performance of CH-Potentials crucially depends on the availability of good lower bounds in the preprocessing phase. Our experiments show, that this availability highly depends on the application. We also show that the overhead of our heuristic is within a factor 1.6 of a hypothetical A*-heuristic that can instantly access lower bound distances. Achieving significantly faster running times could still be possible in variations of the problem setting.

Dropping the provable exactness requirement using a setup similar to anytime A* [45, 34] would be interesting. Another promising research avenue would be to investigate graphs other than road networks. A lot of research into grid maps exists including a series of competitions called GPPC [42]. Hierarchical techniques have been shown to work well on these graphs [44].

## References

1 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

2 Gernot Veit Batz. *Time-Dependent Route Planning with Contraction Hierarchies.* PhD thesis, Karlsruhe Institute of Technology (KIT), 2014.

3 Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

4 Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 166–177. Springer, May 2010. URL: `http://www.springerlink.com/content/u787292691813526/`.

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

6 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

7 Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. *Transportation Science*, 2019.

8 Moritz Baum, Julian Dibbelt, Thomas Pajor, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. Energy-optimal routes for battery electric vehicles. *Algorithmica*, 82(5):1490–1546, 2020. doi:10.1007/s00453-019-00655-9.

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

10 Massimo Bono, Alfonso Emilio Gerevini, Daniel Damir Harabor, and Peter J. Stuckey. Path planning with CPD heuristics. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1199–1205. ijcai.org, 2019. doi:10.24963/ijcai.2019/167.

11 Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. Customizable Contraction Hierarchies with Turn Costs. In Dennis Huisman and Christos Zaroliagis, editors, *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*, OpenAccess Series in Informatics (OASIcs), 2020. Accepted for publication.

12 Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T. K. Satish Kumar. The fastmap algorithm for shortest path computations. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1427–1433. ijcai.org, 2018. doi:10.24963/ijcai.2018/198.

13 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

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

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

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

17 Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.

18 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Fast exact shortest path and distance queries on road networks with parametrized costs. In Jie Bao, Christian Sengstock, Mohammed Eunus Ali, Yan Huang, Michael Gertz, Matthias Renz, and Jagan Sankaranarayanan, editors, *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 66:1–66:4. ACM, 2015. doi:10.1145/2820783.2820856.

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

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

21 Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

22 Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3637`.

23 Stefan Funke, André Nusser, and Sabine Storandt. On k-Path Covers and their Applications. In *Proceedings of the 40th International Conference on Very Large Databases (VLDB 2014)*, pages 893–902, 2014.

24 Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

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

26 Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 100–111. Springer, 2011.

27 Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

28 Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.

29 Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

30 Michael Hamann and Ben Strasser. Graph Bisection with Pareto Optimization. *ACM Journal of Experimental Algorithmics*, 23(1):1.2:1–1.2:34, 2018. `doi:10.1145/3173045`.

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

32 Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 83–93. Springer, May 2010.

33 Alexander Kleff, Frank Schulz, Jakob Wagenblatt, and Tim Zeitz. Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas. In Simone Faro and Domenico Cantone, editors, *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*, volume 160 of *Leibniz International Proceedings in Informatics*, 2020. `doi:10.4230/LIPIcs.SEA.2020.17`.

34 Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, pages 767–774. MIT Press, 2003. URL: `https://proceedings.neurips.cc/paper/2003/hash/ee8fe9093fbbb687bef15a38facc44d2-Abstract.html`.

35 Bing maps new routing engine. `https://blogs.bing.com/maps/2012/01/05/bing-maps-new-routing-engine/`. Accessed: 2020-01-25.

36 Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search on Time-Dependent Road Networks. *Networks*, 59:240–251, 2012. Best Paper Award.

37 Ariel Orda and Raphael Rom. Traveling without waiting in time-dependent networks is NP-hard. Technical report, Dept. Electrical Engineering, Technion-Israel Institute of Technology, 1989.

38 Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.

39 Ben Strasser, Daniel Harabor, and Adi Botea. Fast first-move queries through run-length encoding. In Stefan Edelkamp and Roman Barták, editors, *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press, 2014. URL: `http://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/view/8906`.

40 Ben Strasser, Dorothea Wagner, and Tim Zeitz. Space-efficient, Fast and Exact Routing in Time-dependent Road Networks. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA'20)*, Leibniz International Proceedings in Informatics, September 2020.

41 Ben Strasser and Tim Zeitz. A* with perfect potentials, 2019. `arXiv:1910.12526`.

42 Nathan R. Sturtevant, Jason M. Traish, James R. Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The grid-based path planning competition: 2014 entries and results. In Levi Lelis and Roni Stern, editors, *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*, page 241. AAAI Press, 2015. URL: `http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11290`.

43 Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972.

44 Tansel Uras and Sven Koenig. Identifying hierarchies for fast optimal search. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 878–884. AAAI Press, 2014. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8497`.

45 Rong Zhou and Eric A. Hansen. Multiple sequence alignment using anytime a$^*$. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 975–977. AAAI Press / The MIT Press, 2002. URL: `http://www.aaai.org/Library/AAAI/2002/aaai02-155.php`.

# Engineering Predecessor Data Structures for Dynamic Integer Sets

**Patrick Dinklage**
TU Dortmund University, Germany

**Johannes Fischer**
TU Dortmund University, Germany

**Alexander Herlez**
TU Dortmund University, Germany

## Abstract

We present highly optimized data structures for the dynamic predecessor problem, where the task is to maintain a set $S$ of $w$-bit numbers under insertions, deletions, and predecessor queries (return the largest element in $S$ no larger than a given key). The problem of finding predecessors can be viewed as a generalized form of the membership problem, or as a simple version of the nearest neighbour problem. It lies at the core of various real-world problems such as internet routing.

In this work, we engineer (1) a simple implementation of the idea of universe reduction, similar to van-Emde-Boas trees (2) variants of y-fast tries [Willard, IPL'83], and (3) B-trees with different strategies for organizing the keys contained in the nodes, including an implementation of dynamic fusion nodes [Pătraşcu and Thorup, FOCS'14]. We implement our data structures for $w = 32, 40, 64$, which covers most typical scenarios.

Our data structures finish workloads faster than previous approaches while being significantly more space-efficient, e.g., they clearly outperform standard implementations of the STL by finishing up to four times as fast using less than a third of the memory. Our tests also provide more general insights on data structure design, such as how small sets should be stored and handled and if and when new CPU instructions such as advanced vector extensions pay off.

**2012 ACM Subject Classification** Theory of computation → Predecessor queries

**Keywords and phrases** integer data structures, dynamic data structures, predecessor, universe reduction, y-fast trie, fusion tree, B-tree

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.7

**Related Version** *Full Version*: `https://arxiv.org/abs/2104.06740`

**Supplementary Material** *Software*: `https://github.com/pdinklag/tdc/tree/sea21-predecessor` archived at `swh:1:dir:40d4000cd5f3302b2d037401280dcf1bea2c8b31`

**Funding** *Patrick Dinklage*: Supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).
*Alexander Herlez*: Supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

## 1 Introduction

Finding the predecessor of an integer key in a set of keys drawn from a fixed universe is a fundamental algorithmic problem in computer science at the core of real-world applications such as internet routing [8]. It can be considered a generalized form of the membership problem or a simple version of the nearest neighbour problem. Navarro and Rojas-Ledesma [20] recently gave a thorough survey on the topic, recapping the past four decades of research.

Data structures for the predecessor problem are designed to beat the $\Omega(\lg n)$ lower time bound for comparison-based searching. While optimal data structures have been shown for static sets [21] that are known in advance and do not change, they do not necessarily translate to the most practical implementations. Dinklage et al. [10] face the symmetrical *successor* problem for a small universe and develop a simple data structure that accelerates binary search. Despite not optimal in theory, it is the most efficient in their setting.

In this work, we focus on the *dynamic* problem, where the set of integers can be changed at any time by inserting, deleting or updating keys. A prominent example of a dynamic predecessor data structure is the *van Emde Boas tree* [24], which, despite near-optimal query times in theory, has been proven irrelevant in practice due to its memory consumption [25]. Dementiev et al. [9] implemented a *stratified trie* as a heavily simplified practical variant of the van Emde Boas tree for keys drawn from a 32-bit universe. Nowadays, with 64-bit architectures dominating the landscape, the limitation to 32-bit keys can be considered significant. The authors gave no hints as to how the data structure can be altered to properly handle larger universes, and simply applying the same structure on a larger universe exceeds practical memory limitations quite quickly. Nash and Gregg [19] thoroughly evaluated various dynamic predecessor data structures in practice, including the aforementioned stratified tree. They also implemented AVL trees, red-black trees and B-trees, as well as the trie hybrid by Korda and Raman [17] and their self-engineered adaptation of *burst tries* [14] for integer keys, which outperform the other data structures regarding both speed and memory usage.

**Our contributions.** We engineer new practical solutions for the dynamic predecessor problem that are both faster and more memory efficient than the current best known to us. First, we apply the idea of *universe sampling* following [10] to the dynamic case. Second, we engineer *y-fast tries* [26], which, in our view, offer room for many practical optimizations. Finally, we implement *dynamic fusion nodes* [22], for which Pătraşcu and Thorup give a very practical description but no implementation. We embed them into B-trees and make use of modern CPU instructions to accelerate some key low-level operations.

We note that our data structures are designed in a way often not optimal in theory. A recurring observation that we made is that thanks to large CPU caches, naïve solutions for queries on small datasets often outperform sophisticated data structures on modern hardware, including linear scans of unsorted lists, or binary search in naïvely organized sorted lists, where updates potentially require all items to be shifted. This observation has been confirmed in the contexts of balanced parentheses [4, 11] and finding longest common extensions in strings [10, 15]. We make use of this and replace predecessor data structures for small input sets by sorted or unsorted lists without any auxiliary information.

This paper is organized as follows: we begin with definitions and notations in Section 2 and a description of our experimental setup in Section 3. Then, in Sections 4–6, we describe our engineered data structures and give individual experimental results. In Section 7, we conclude with a comparison of the best configurations with existing implementations.

## 2 Preliminaries

Let $S$ be a set of $n = |S|$ positive integers called *keys* drawn from a fixed universe $U := [u] = [0, u-1]$. For any $x \in U$, we call $\mathrm{pred}_S(x) = \max\{y \in S \mid y \leq x\}$ the *predecessor* of $x$, which is the largest key in $S$ no larger than $x$. We consider the dynamic scenario, where keys may be inserted into or deleted from $S$ and the data structure must be updated accordingly.

In our analysis, we use the word RAM model, where we assume that we can perform arithmetic operations on words of size $w = \Theta(\lg u)$ in time $\mathcal{O}(1)$ (by default, logarithms are to the base of two). Additionally, the binary logic operations OR ($\vee$), AND ($\wedge$) and XOR

($\oplus$) on words take constant time. With this, we can access the $i$-th bit in a word $x$, denoted by $x\langle i\rangle$, as well as the bits $i$ to $j$ (inclusively) of $x$, denoted by $x\langle i\,..\,j\rangle$, in constant time. We refer to bit positions in MSBF order, e.g., $x\langle 0\rangle$ is the most significant bit of $x$.

We also require some advanced operations on words to be answered in constant time. Let $\mathrm{msb}(x)$ denote the position of the most significant set bit of $x$ and $\mathrm{select}_1(x, k)$ the position of the $k$-th set bit in $x$. Another needed operation is counting the number of trailing zero bits of $x$. In theory, these queries can be answered in constant time using the folklore approach of precomputing universal tables of size $o(u^{1/c})$ bits, where we can look up the answers for all possible queries on a constant number of $c > 1$ blocks of size $w/c$. In practice, we can make use of special CPU instructions: LZCNT and TZCNT count the number of leading or trailing zeroes, respectively, and POPCNT reports the number of set bits in a word. These instructions are fairly widely spread, being implemented by current versions of the x86-64 (both Intel [16] and AMD [1]) instruction sets as well as ARM [3].

**Tries.** Tries are a long-known information retrieval data structure [12]. Here, we consider *binary* tries for strings over a binary alphabet. Consider a key $x \in U$ to be inserted into a binary trie. We navigate the trie top-down according to the bits of the binary representation of $x$ in MSBF order: when reading a 0-bit, we go to the current node's left child, and otherwise to the right child. Inserting a new element $x$ works accordingly, creating any node that does not yet exist. Since the binary trie for a set of keys drawn from $U$ has height $\lceil \lg u \rceil$, this takes total time $\mathcal{O}(\lg u)$. An example of a binary trie is shown in Figure 5a. Binary tries are suitable for solving the dynamic predecessor problem: to find $\mathrm{pred}_S(x)$, we navigate down the trie as if we were to insert it. If we reach a leaf labeled $x$, then $x \in S$ and it is its own predecessor. Otherwise, we eventually reach an inner node $v$ that is missing the left or right edge that we want to navigate, respectively. If the right edge is missing, the predecessor of $x$ is the label of the rightmost leaf in the left subtrie of $v$. Otherwise, if the left edge is missing, we first navigate back up to the lowest ancestor $v'$ of $v$ that has two children and where $v$ is in the right subtrie; then the predecessor is the label of the rightmost leaf in the left subtrie of $v'$. In either case, we can report the predecessor of $x$ in time $\mathcal{O}(\lg u)$. Deleting is done by locating a key's leaf, removing it, and navigating back up removing any inner node no longer connected to any leaves, all in time $\mathcal{O}(\lg u)$. The number $\mathcal{O}(n \lg u)$ of nodes in the binary trie can be reduced to $\mathcal{O}(n)$ by contracting paths of branchless inner nodes to single edges [18]. We call a trie *compact* if it does not contain any inner nodes with one child.

## 3 Methodology

We conduct the following three-step experiment for our data structures:
**(1)** insert $n$ keys drawn uniformly at random from $U$ into the initially empty data structure,
**(2)** perform *ten million* random predecessor queries for keys in the range of the inserted keys, guaranteeing that there is always a predecessor that is never trivially the maximum, and
**(3)** delete the $n$ keys from the data structure in the order in which they were inserted.

In preliminary experiments, we also considered distributions other than uniform, and also intermingling insertions, queries and deletions. Apart from statistical fluctuations, the results led to the same assertions and thus we solely consider the experiment described above.

For each data structure, we run five iterations using a different random seed each (but the same seeds for all data structures and in the same order). We measure running times by the wall clock time difference between start and finish of an iteration, as well as the RAM usage using custom overridden versions of `malloc` and `free` and compute the averages over the five iterations. Our code is written in C++17 and publicly available[1]; we compile using GCC version 9.3. For hash tables (Sections 4 and 5), we use a public[2] implementation of Robin Hood hashing [6] that is both faster and more memory efficient than the STL implementation (`std::unordered_map`). We conduct our experiments on Linux machines with an Intel Xeon E5-4640v4 processor (12 cores at 2.1 GHz, 12×32 kB L1, 12×256 kB L2, 30 MB L3 shared, line size 64 B) and 256 GB of RAM.

## 4 Dynamic Universe Sampling

A common technique used by predecessor data structures is known as *length reduction* [5], where we partition the universe into buckets of size $b \ll u$ and reduce the predecessor problem to the much smaller sub-universe $[b]$. For each bucket, we determine a representative, e.g., the minimum contained key, which is entered into a *top level* predecessor data structure. The buckets are maintained on the *bucket level*. When $\text{pred}(x)$ is queried for some $x \in U$, we first solve the predecessor problem on the top level to find the bucket that $x$ belongs into, and then reduce the query to a smaller one answered on the bucket level. The van Emde Boas tree [24] applies this approach recursively. In this work, we develop a dynamic version the two-level data structure by Dinklage et al. [10] that achieved very good practical results for the *static* predecessor problem.

We partition $U$ into buckets of size $b = 2^k$ for some $k > 0$. Let $i \in [u/b]$, then the $i$-th bucket can only contain keys from the interval $[bi, b(i+1)-1]$. We call a bucket *active* if it contains at least one key from $S$. Since $b = 2^k$, the number $i$ of the bucket that a key $x \in U$ belongs into is the number represented by the $\lceil \lg u \rceil - k$ highest bits of $x$. Hence, we only store the lowest $k$ bits for each key to reduce space usage in the buckets.

**Top level.** The top level maintains the set of active buckets. Consider a query for $\text{pred}_S(x)$, then it reports the *rightmost* active bucket $i$ such that $bi \le x$. The predecessor of $x$ is then contained in bucket $i$ if $x$ is greater than the bucket's current minimum. Otherwise, it is the current maximum key contained in the active bucket preceding $i$. Clearly, the top level requires a dynamic predecessor data structure on the set of active buckets, i.e., keys drawn from the universe $[u/b]$ represented by the keys' high bits. We explore two basic options. First, let $i_{\min}$ and $i_{\max}$ be the numbers of the leftmost and rightmost active buckets, respectively. We store $i_{\max} - i_{\min} = \mathcal{O}(u/b)$ pointers in an array such that the $i$-th entry points to the rightmost active bucket $i'$ with $i' \le i$. Predecessor queries can trivially be answered in time $\mathcal{O}(1)$ using a lookup, but updates may take time $\mathcal{O}(u/b)$ in the worst case as we may need to shift pointers and/or update pointers for succeeding non-active buckets. Furthermore, the array requires up to $\lceil (u/b) \lg(u/b) \rceil$ bits of space. Our alternative is a hash table $H$ containing only pointers to active buckets, identified by their numbers. Let $b'$ be the number of active buckets, then $H$ requires $\mathcal{O}(b' \lg(u/b))$ bits of space. Updates can be done in $\mathcal{O}(1)$ expected time, but since the order of buckets in $H$ is arbitrary, queries may require to perform up to $b'$ lookups: when a key belongs in bucket $i$, we look up $i$ in $H$; if that bucket is not active, we find no result and look up $i-1$, and so on. This takes up to $\mathcal{O}(b')$ expected time.

[1] Our code is published at `https://github.com/pdinklag/tdc/tree/sea21-predecessor`. Make sure to check out the *sea21-predecessor* branch, which contains instructions in the readme.

[2] Robin Hood hashing by Martin Ankerl: `https://github.com/martinus/robin-hood-hashing`.

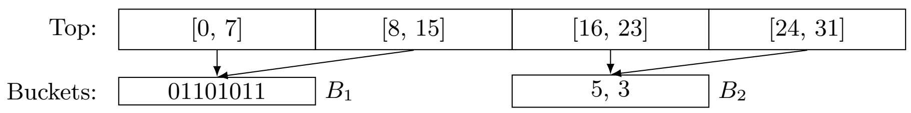


**Figure 1** Hybrid universe sampling data structure for $S = \{1, 2, 4, 6, 7, 19, 21\}$ with $w = 5$, $b = 8$ and $\theta_{\min} = \theta_{\max} = 3$ . The top level holds bucket pointers for the partitioned universe. Since there are no keys from the intervals $[8, 15]$ and $[24, 31]$ contained in $S$, their pointers point to the respective preceding buckets. Bucket $B_1$ is represented as a bit vector of length $b$ such that each 1-bit corresponds to a key contained in $S$. Bucket $B_2$, on the other hand, only contains two keys that are represented as an unsorted list of keys relative to the left interval boundary.

**Bucket level.** On the bucket level, we first look at two basic data structures. We only store the lowest $k = \lg b$ bits of the contained keys, called *truncated keys* in the following, as the high bits are already defined by the bucket number. Let $S_i \subseteq S$ be the set of truncated keys contained in the $i$-th bucket. We can store them in a bit vector $B_i \in \{0,1\}^b$ where $B_i[x] = 1$ if $x \in S_i$ and $B_i[x] = 0$ otherwise. Updates are then done in constant time by setting or clearing the respective bit in $B_i$. To compute $\mathrm{pred}_{S_i}(x)$, we scan $B_i$ linearly in time $\mathcal{O}(b/\lg u)$ using word packing. However, $B_i$ always requires $b$ bits of space. Let $n'$ be the *current* number of keys contained in a bucket and consider the case where $n' < b/\lg b$. An alternative is storing an unsorted list of $n'$ keys: this requires only $n' \lg b < b$ bits of space and retains $\mathcal{O}(1)$ time insertions, and predecessor queries and deletions take time $\mathcal{O}(n') = \mathcal{O}(b/\lg b) = \mathcal{O}(b/\lg u)$.

We now consider a hybrid of the two basic bucket structures. Let $\theta_{\min}$ and $\theta_{\max}$ be thresholds with $0 < \theta_{\min} \leq \theta_{\max} < b$. We maintain a bucket as an unsorted list of keys as long as $n' < \theta_{\max}$. If, after inserts, the bucket grows beyond $\theta_{\max}$ keys, we rebuild it to a bit vector. If, after deletions, the bucket size falls below $\theta_{\min}$ keys, we revert to an unsorted list. Rebuilding the bucket to a bit vector or unsorted list, respectively, takes time $\mathcal{O}(b)$. Let $\theta_{\min} := cb/\lg b$ and $\theta_{\max} := \theta_{\min} + c' \lg b$ for constants $c, c' > 0$. Then, $\Theta(\lg b)$ insertions need to occur before we switch to a bit vector representation, followed by $\Theta(\lg b)$ deletions before reverting to an unsorted list. We can thus amortize the time needed for one insertion and one deletion to $\mathcal{O}(b/\lg b)$. At all times, predecessor queries take at most $\mathcal{O}(b/\lg u)$ time and the bucket requires at most $b$ bits of space. Figure 1 shows an example.

**Experimental evaluation.** Following our considerations in Appendix B,

**(1)** we set $b := 2^{24}$ for buckets backed by *bit vectors*,

**(2)** we set $b := 2^{10}$ for buckets backed by *unsorted lists* and

**(3)** for *hybrid* buckets, we set $\theta_{\min} := 2^9$ and $\theta_{\max} := 2^{10}$ and try different sizes $b$.

Our results are shown in Figure 2. We first discuss the results for 32-bit keys. In most configurations, we achieve throughputs higher than $2^{22}$ operations per second for both updates (insertions and deletions) and predecessor queries. Furthermore, we achieve compression in that we require less than 32 bits per key, because we only store trunacted keys within the buckets. The compression increases with larger $n$ as for sufficiently large $n$, all buckets are active and more (truncated) keys are inserted into the same number of buckets. We observe that the top level organization, array versus hash table, barely appears to matter. The only difference is a slightly slower performance, but also lower memory consumption of the hash tables for smaller $n$, which was to be expected. However, as all buckets become active for larger $n$, these differences become negligible.

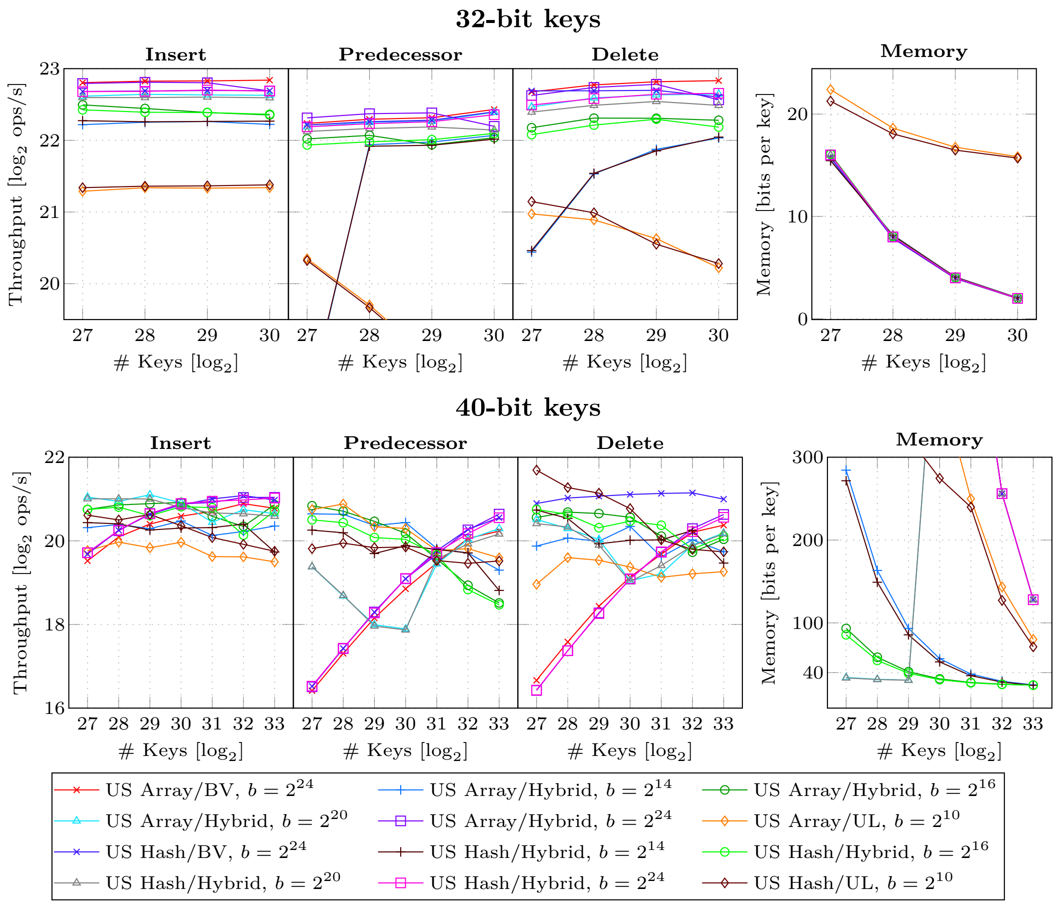


**Figure 2** Throughputs for the insert, predecessor and delete operations, as well as memory usage of the universe sampling data structures for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour. In the legend, *US* stands for universe sampling, *BV* stands for buckets backed by bit vectors, *UL* for unsorted lists. Missing points indicate throughputs lower than $2^{20}$ operations per second or exceeding of 300 bits consumed per key, respectively, and are omitted for clarity.

We now look at the three types of bucket organization. The clear outliers are where we implement buckets as unsorted lists of size at most $2^{10}$: here, all operations are between 2–4 times slower than the rest, and the number of inserted keys visibly affects the performance of queries and deletions negatively, which is due to linear scans facing a higher bucket fill rate. Hybrid buckets and those backed by bit vectors appear to be on par especially for large $n$, as the hybrid representation eventually switches to bit vectors. As expected, the bit vector representation achieves higher compression than the unsorted list representation.

Now, we discuss the results for 40-bit keys. The memory consumption is obviously different: while we still achieve compression below 40 bits per key for large $n$, the top level now contains up to $2^{30}$ active buckets (for buckets of size $2^{10}$), causing a big memory overhead that can only be compensated for sufficiently large $n$. Hybrid buckets may cause an explosion of memory consumption when they switch to bit vectors, as can be seen for bucket size $2^{24}$ at $2^{30}$ keys. For $2^{33}$ keys, we can see how it slowly starts to compensate. Regarding performance, similar observations as for 32-bit keys can be made, except that the top level organization now does matter: the smaller the buckets, the more linear scans weigh in, such that the hash table approach becomes faster for updates but slower for queries. Since this data structure is not suitable for large universes, we omit experiments for 64-bit keys.

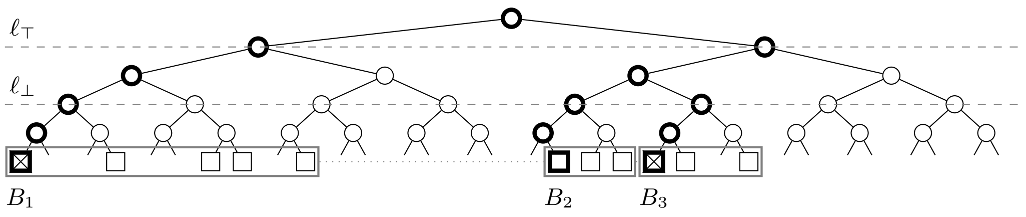


**Figure 3** Our y-fast trie for $w = 5$ and $S = \{3, 6, 7, 9, 17, 18, 19, 21, 23\}$ with $t = 2$, $c = 2$ and $\gamma = 1$. Edge and leaf labels of the conceptual trie are omitted for the sake of clarity: left edges are labeled by 0, right edges by 1 and leaves are labeled by the corresponding keys. Keys contained in $S$ are shown as squares, where representatives of buckets have a thick border. Representatives marked with an X are deleted: they are still representatives of buckets, but no longer contained in $S$ themselves. Buckets are shown as rectangles around the contained keys. Nodes on paths that lead to representatives are contained in the x-fast trie's LSS and are drawn with a thick border; other nodes are not contained in the LSS. Levels $\ell_\top$ and $\ell_\bot$ are highlighted by dashed lines.

## 5 Y-Fast Tries

The *x-fast* trie by Willard [26] is conceptually a variation of the binary trie where
**(1)** the keys of $S$ are doubly-linked in ascending order and
**(2)** if a node does not have a left (right) child, then the corresponding pointer is replaced by a *descendant* pointer that directly points to the smallest (largest) leaf descending from it.
The trie is stored in the *level-search data structure* (LSS). We say that a node of the trie is on level $\ell$ if it has depth $\ell$. For each level $\ell$ of the trie, the LSS stores an entry for every node $v$ that exists on level $\ell$, which we identify by the bit sequence $B_v \in \{0, 1\}^\ell$ that encodes the path in the trie from the root to $v$. Specifically, the LSS associates $B_v$ to $v$'s descendant pointers. We can find $\text{pred}_S(x)$ in expected time $\mathcal{O}(\lg \lg u)$ as follows: we first binary search the $\lceil \lg u \rceil$ levels of the trie to find the bottom-most node $v$ on the path leading to $x$ if it were contained in $S$. On each level $\ell$ that we inspect, we query the bit prefix $x\langle 0 .. \ell - 1\rangle$ in the LSS in $\mathcal{O}(1)$ expected time to test if we are done. From $v$, by construction, we can take a descendant pointer to the predecessor or successor of $x$. Updates of the x-fast trie require $\mathcal{O}(\lg u)$ expected time as in the worst case, the LSS needs to be updated for every level following an insertion or deletion. The total memory consumption of the x-fast trie is $\mathcal{O}(n \lg u)$ words.

The *y-fast trie* improves this to $\mathcal{O}(n)$ words: we partition $S$ into $\Theta(n/\lg u)$ buckets of $\Theta(\lg u)$ keys each and determine a *representative* for each bucket, e.g., the minimum contained key. Then, we build an x-fast trie over only the representatives, which occupies $\mathcal{O}(n)$ words of memory. For each bucket, we construct a binary search tree for the keys contained in it, consuming $\mathcal{O}((n/\lg u) \cdot \lg u) = \mathcal{O}(n)$ words. When looking for the predecessor of $x$, we can locate its bucket using the x-fast trie over the representatives in expected time $\mathcal{O}(\lg \lg u)$ and within the buckets, searching and updating can be done in time $\mathcal{O}(\lg \lg u)$. The sampling of representatives also improves the amortized expected update times to $\mathcal{O}(\lg \lg u)$.

**Implementation.** Let $t = \Theta(\lg u)$, $\gamma > 0$, and $c > 2\gamma$ be parameters. We partition $S$ into buckets of size variable in $[\gamma t, ct]$. We name the minimum key contained in a bucket its representative and only representatives are contained in the x-fast trie, which has height $\lceil \lg u - \lg t \rceil$ as the lowest $\lg t$ bits of the keys are maintained in the buckets. Within a bucket, we store keys in a sorted or unsorted list rather than a binary search tree. This increases the asymptotic time needed for updates and predecessor queries, but the additional memory costs for structures such as binary trees, albeit asymptotically constant, would be too high in practice. Figure 3 shows an example of our y-fast trie that we describe in the following.

We try to keep $t$ small such that buckets fit into few consecutive cache lines and can be searched quickly. Because this directly affects the height of the x-fast trie maintaining the representatives, we speed up searches as follows: let $\ell_\top$ be the bottommost level where *all* possible nodes exist in the x-fast trie and let $\ell_\perp$ be the topmost level where *no branching nodes* exist in the x-fast trie. Consider an operation involving a key $x \in U$: we locate its bucket by a vertical binary search in the x-fast trie's LSS. Because all levels above $\ell_\top$ contain all possible nodes and because all nodes on levels below $\ell_\perp$ point to the same buckets as their respective ancestors on level $\ell_\perp$, we limit the binary search to the levels between $\ell_\top$ and $\ell_\perp$ and maintain $\ell_\perp$ and $\ell_\top$ under updates with no asymptotic extra cost. With this strategy, we can also save space by avoiding storage of any nodes on levels below $\ell_\perp$; the corresponding hash tables in the LSS simply remain empty. Intuitively, this cuts off trailing unary paths in the x-fast trie. Note that due to the sampling mechanism, we always have $\ell_\perp \leq \lg(\gamma t)$, so the $\Theta(\lg \lg u)$ bottommost levels are never stored.

To speed up deletions, we allow the representative of a bucket to be no longer contained in the bucket by itself. When it is deleted, we mark it as such, but it remains the bucket's representative and also remains in the x-fast trie. This strategy avoids the need of finding a new representative and updating the x-fast trie every time a representative is deleted. However, we must consider a special case when answering predecessor queries. Let $y_{\text{rep}}$ be the deleted representative of a bucket and $y_{\min} > y_{\text{rep}}$ the smallest key currently contained in the bucket, and consider the query $\text{pred}_S(x)$ with $y_{\text{rep}} \leq x < y_{\min}$. The x-fast trie will lead us to said bucket and $y_{\text{rep}}$ would be the predecessor of $x$. When we detect $y_{\text{rep}}$ as deleted, we follow a pointer to the preceding bucket, which must contain the predecessor of $x$.

Buckets are merged and split as in B-trees [7, chapter 18] to ensure their size stays within $[\gamma t, ct]$. To avoid the creation of a new bucket each time a new minimum is inserted into the data structure, we maintain a special bucket with representative $-\infty$ that we allow to become empty and will never be removed by a merge.

When using unsorted lists to maintain keys within a bucket, we can amortize insertion costs. Unless a split is required, inserting a key into a bucket simply means appending it in constant time. Thus, if we choose $c := \gamma + \Theta(t) > 2\gamma$, we can amortize the time needed for a split over $\Theta(t)$ constant-time insertions. This amortization leads to $\mathcal{O}((\lg u)/t) = \mathcal{O}(1)$ time needed for inserting a key into a bucket followed by a potential split, such that the amortized expected insertion time of the y-fast trie is $\mathcal{O}(\lg \lg u)$ as in the original. This cannot be achieved for deletions, as the key to be deleted needs to be located in time $\mathcal{O}(\lg u)$ first.

► **Example 1** (insertion)**.** Consider the y-fast trie in Figure 3 with $t = 2$ and $c = 2$. We insert the new key 8. The binary search in the x-fast trie's LSS is constrained only to the three levels between $\ell_\top$ and $\ell_\perp$ and leads to bucket $B_1$ with (deleted) representative 0. We insert $x$ by appending it to the unsorted list of keys. However, we then have $|B_1| = 5 > 4 = ct$, thus we have to split $B_1$. We create a new bucket $B_1'$ with representative 7 (the median) and move keys such that $B_1 := \{3, 6\}$ and $B_1' := \{7, 8, 9\}$. Even though 0 is no longer contained in $B_1$, it remains its representative. Finally, we enter key 7 into the x-fast trie, causing two new nodes to be added to the LSS. However, $\ell_\perp$ remains unchanged, as the newly added nodes form a unary path beginning at level $\ell_\perp$.

► **Example 2** (deletion)**.** Consider the y-fast trie in Figure 3 with $t = 2$ and $\gamma = 1$. We delete key 21, which we find in bucket $B_3$ as in Example 1. After deletion, we have $|B_3| = 1 < 2 = \gamma t$ (note how 20 is the representative, but is marked as deleted), thus we have to merge. As the only neighbour, we merge with bucket $B_2$ by moving key 23 such that $B_2 := \{17, 18, 19, 23\}$.

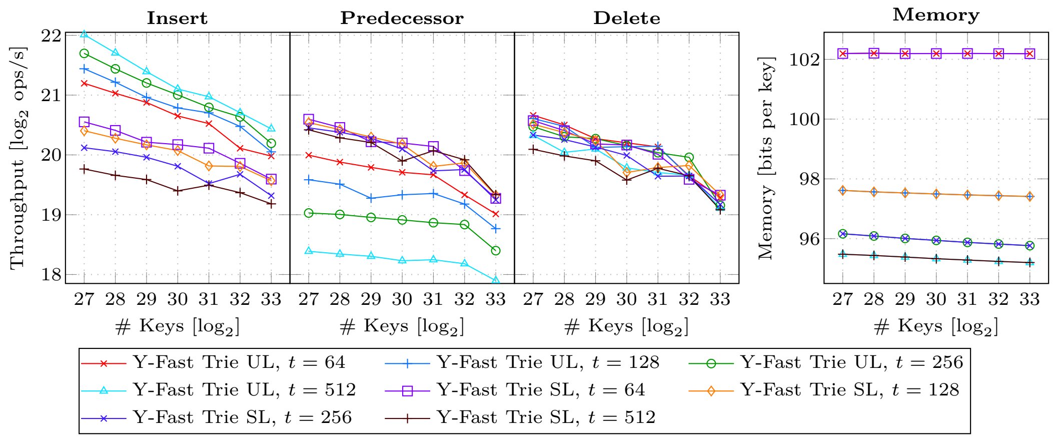


**Figure 4** Throughputs for the insert, predecessor and delete operations, as well as memory usage of the y-fast trie for $U = [2^{64}]$. Best viewed in colour. *UL* stands for unsorted, *SL* for sorted lists.

The former representative of $B_3$, key 20, is now removed from the x-fast trie. Observe how the path of nodes leading to $B_2$ now becomes a unary path starting at level $\ell_\top$. Because all unary paths then start at level $\ell_\top$, we set $\ell_\bot := \ell_\top$.

**Experimental evaluation.** In our experiments, we set $c := 2$ and $\gamma := 1/4$ and choose $t$ as powers of two to optimize memory alignments. Our results for $t := 64$ to 512 and 64-bit keys are shown in Figure 4. (Additional results are given in Figure 8 in Appendix A.)

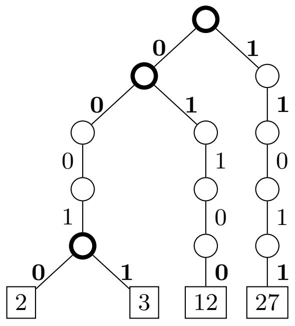
The fastest predecessor queries are achieved when buckets are organized as sorted lists and binary search is used to answer bucket-level queries. Conversely, insertions are are up to twice as fast in the unsorted list case, where new keys simply have to be appended without preserving any order. Regarding deletions, there is no substantial difference between using a sorted or unsorted list to organize the buckets: while we can find the item to be deleted using binary search when using a sorted list, we have to shift up to $t$ keys afterwards. As we use simple arrays for storage in either case, there is also no difference in memory consumption.

As expected, the bucket size of $t$ is a direct trade-off parameter for update versus query performance and memory usage, which is very visible when buckets are organized as unsorted lists. Here, insertions become faster as the bucket size grows since they are trivial on the bucket level and the LSS needs to be updated less often. However, larger buckets mean longer scans when answering predecessor queries. The bucket size is much less impactful on query performance when sorted lists are used, as the bucket-level query time is then only logarithmic in the bucket size. The memory consumption is also affected by the bucket size: larger buckets imply less levels in the LSS and thus less memory needed.

As a conclusion, unsorted lists may be preferable when fast insertions are required and the performance of predecessor queries is less important. For the general case, however, using sorted lists appears to be preferable, as all operations then have similar throughputs.

## 6 Fusion Trees

Pătraşcu and Thorup [22] introduce *dynamic fusion nodes* as a sorted list data structure for $|S| \leq k \leq \sqrt{w}$ keys that supports predecessor queries and updates in time $\mathcal{O}(1)$. It is based on the fusion node, originally described by Fredman and Willard [13], that simulates



**(a)** The binary trie for $S$. Branching nodes have thicker outlines and bits at distinguishing positions are written in bold.

$M = 11001$

| $x$ | binary | $\hat{x}$ | $\hat{x}^?$ | Branch | Free |
|---|---|---|---|---|---|
| 2 | 00010 | 000 | 000 | 000 | 000 |
| 3 | 00011 | 001 | 001 | 001 | 000 |
| 12 | 01100 | 010 | 01? | 010 | 001 |
| 27 | 11011 | 111 | 1?? | 100 | 011 |

**(b)** The keys $x \in S$, with binary representation and compressed versions $\hat{x}$ and $\hat{x}^?$ without and with don't cares according to [13] and [22], respectively. Branch and Free encode the matrix given by column $\hat{x}^?$ as described in [22]. The mask $M$ marks the distinguishing positions of $S$.

**Figure 5** Binary trie and compressed keys for $S = \{2, 3, 12, 27\}$ and $w = 5$.

navigation in a compact binary trie of $S$ represented by compressed keys. Given a key $x \in S$, we only consider those bits at *distinguishing* positions. A position $\ell < \lceil \lg u \rceil$ is a distinguishing position if there is at least one branch on level $\ell$ in the binary trie. For $k$ keys, there can be at most $k$ branches in the binary trie, and thus there can be at most $k$ distinguishing positions. A *compressed key* $\hat{x}$ consists of only the bits of $x$ at distinguishing positions moved to the $k$ least significant positions. We maintain the set of distinguishing positions in a *mask* $M$ of $w$ bits where the $k$ distinguishing bits are set and all other bits are clear. Figure 5b shows an example. From $x$, we can compute $\hat{x}$ in constant time by masking out unwanted bits using $M$, followed by multiplications to relocate the distinguishing bits. The $k$ compressed keys of $S$ can be stored in a $k \times k$ bit matrix $\hat{S}$ that fits into a single word. With this, we can compute $\mathrm{pred}_S(x)$ in time $\mathcal{O}(1)$ as Fredman and Willard describe in [13]. However, updates may cause a new position to become distinguishing after an insertion, or a position to be no longer distinguishing after a deletion. In these cases, their data structure needs to be rebuilt from scratch. To resolve this, Pătraşcu and Thorup introduce *don't care* bits (written ?) that indicate bits at distinguishing position that are, however, not used for branching in a specific compressed key. The data structure now contains a $k \times k$ matrix over the new alphabet $\{0, 1, ?\}$, which we encode using two $k \times k$ bit matrices that fit into one word each. Examples for this can be seen in Figure 5b. The notion of wildcards allows for updating the data structure in time $\mathcal{O}(1)$. We refer to Appendix C for a more detailed description and examples, including an elaboration of the deletion of keys not given in [22].

A *B-tree* is a self-balancing multiary tree data structure for representing a dynamic ordered set of items. With $B$ the maximum degree of a node, it is guaranteed to maintain height $\log_B n$, such that lookup – including predecessor – queries and updates can be done in time $\mathcal{O}(\log_B n)$. We consider B-trees a well-known folklore data structure and refer to [7, chapter 18] for a comprehensive introduction. Embedding fusion nodes into a B-tree, using the keys contained in the nodes as splitters, are the typical ingredients of a *fusion tree.*

**Implementation.** As we deal with 64-bit architectures ($w = 64$), we choose $k := 8$, such that a $k \times k$ bit matrix can be stored in a single word represented row-wise by an array $\hat{X} = [\hat{x}_0, \ldots, \hat{x}_7]$ of compressed keys. We keep $\hat{X}$ in ascending order, i.e., $\hat{x}_0 < \hat{x}_1 < \cdots < \hat{x}_7$.

The most important operation is key compression, writing only the bits of $x$ at distinguishing positions into a word $\hat{x}$. Instead of an approach based on sparse tables [23], we make use of the *parallel bits extract* (PEXT) instruction [16]. Let $M$ be the $w$-bit mask identifying

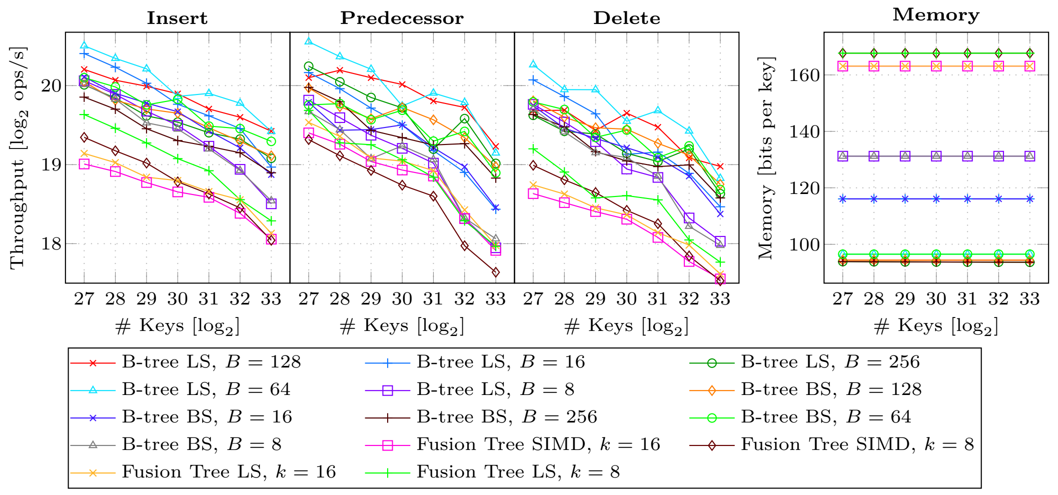


**Figure 6** Throughputs for the insert, predecessor and delete operations, as well as memory usage of the fusion trees and B-trees for $U = [2^{64}]$. Best viewed in colour. In the legend, *LS* stands for linear searched nodes, *BS* for binary search and *SIMD* for use of SIMD instructions.

distinguishing positions. Then, conveniently, $\hat{x} = \text{PEXT}(x, M)$. Another core operation is finding the rank $i$ of a compressed key $\hat{y}$ in $\hat{X}$. For this, we use the MMX SIMD instruction PCMPGTB [16], which performs a byte-wise greater-than comparison of two 64-bit words. First, we multiply $\hat{y}$ by the constant $(0^{k-1}1)^k$ to retrieve the word $\hat{y}^k$ containing $k$ copies of $\hat{y}$. Let $j$ be the smallest rank such that $\hat{x}_j > \hat{y}$. The instruction $\text{PCMPGTB}(\hat{X}, \hat{y}^k)$ returns the word $B_{\hat{X}>\hat{y}}$ where the $kj$ lowest bits are zero and the remaining bits are set (because $\hat{X}$ is ordered). Therefore, $j = \lfloor \text{TZCNT}(B_{\hat{X}>\hat{y}})/k \rfloor$ and finally $i = j - 1$. Alternatively, because $\hat{X}$ easily fits into a cache line, we consider a naïve linear search.

We extend our implementation to support also $k = 16$ by simulating a 256-bit word using four 64-bit words. The special CPU instructions can be extended by executing them on each of the four 64-bit words and then combining the results. The processors to our disposal actually support a variant of PCMPGTB for the parallel comparison of sixteen 16-bit words contained in a 256-bit word (namely the Intel intrinsic `_mm256_cmpgt_epi16`).

We implement B-trees largely following the description in [7, chapter 18]. Nodes have at most $B$ children and thus contain up to $B - 1$ keys used as splitters. When plugging fusion nodes into B-trees of degree $k$, we have *fusion trees*.

**Experimental evaluation.** We use $B := 8$ and $B := 16$ for fusion trees with $k = 8$ and $k = 16$, respectively, comparing fusion nodes using the PCMPGTB instruction for rank queries against those using simple linear scans. Further, we compare fusion trees to straight B-trees, finding predecessors in a node using $\mathcal{O}(\lg B)$-time binary or $\mathcal{O}(B)$-time linear search. There, we also consider much larger $B$, as preliminary experiments suggested that the performance of all operations peaks at $B := 64$. Our results for 64-bit keys are presented in Figure 6. (More results for smaller universes are given in Figure 9 in Appendix A.)

To our surprise, fusion trees achieve the lowest throughputs for all operations: B-trees with large $B$ are up to twice as fast, and even the B-trees with low degrees are visibly faster overall. Fusion trees also require more memory per key, which was expected, as each node needs to store three words (the compression mask and two matrices) in addition to the keys themselves. Interestingly, the fusion nodes using linear scans for ranking outperform those

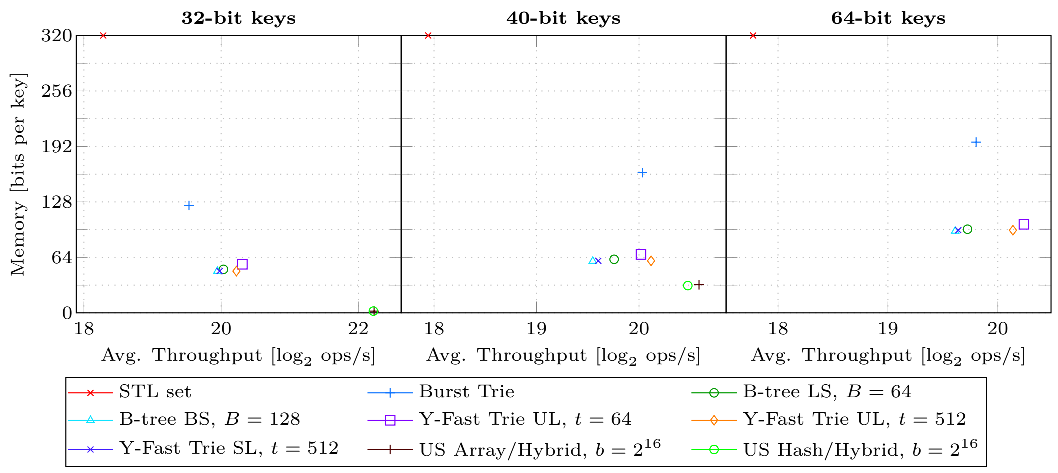


**Figure 7** Comparing the average throughput of operations versus memory use of dynamic predecessor data structures for different universes and $n = 2^{30}$.

that use the SIMD instructions in nearly all instances. The reason is presumably that the corresponding MMX/AVX registers have to be filled prior to executing these instructions: in a direct comparison answering immediately consecutive random rank queries, the SIMD variant is about 28% faster than scanning. Fusion trees with $B = 16$ perform slower overall than those with $B = 8$ despite their lower height, which is due to overheads in our simulation of 256-bit words. It shall be interesting to redo these experiments with natively supported wide registers (e.g., AVX-512) and necessary instructions in the future.

We have a brief closer look at B-trees. Our preliminary experiments are largely confirmed in that B-trees with $B = 64$ perform best overall. For $B \leq 64$, nodes backed by linear search perform faster than those backed by binary search. The exact opposite is the case for $B > 64$, where binary search becomes faster. Concerning memory, unsurprisingly, the higher $B$ is chosen, the less memory is required as the tree structure shrinks in height.

## 7 Comparison

In Figure 7, we plot the average throughput of insertions, predecessor queries and deletions against the memory usage of a subset of our data structures from Sections 4–6 for a fixed workload size of $2^{30}$. For comparison, we also show the performance of the STL set (`std::set`, an implementation of red-black trees), and the burst trie of Nash and Gregg [19] – to the best of our knowledge the best practical dynamic predecessor data structure thus far. Note that burst tries are *associative* and store a value along with each key, so for a fair comparison, one should subtract $w$ bits per key for each data point.

For all universes, at least one of our data structures is over four times faster than the STL set, the extreme being for 32-bit keys, where our sampling structures achieve an average throughput of approximately $2^{22.2}$ operations per second, whereas the set does about $2^{18.2}$. Furthermore, our data structures consume less than a third of the set's 320 bits per key. For 32-bit keys, we also outperform burst tries completely, where even our slowest data structure (B-trees with degree 128 and binary searched nodes) is about 33% faster. Our two sampling data structures with hybrid buckets of size $2^{16}$ are clearly the fastest. Their low space consumption of just about 2 bits per key should, however, be interpreted with care, as for $n = 2^{30}$, one quarter of all possible keys is contained in $S$, and hence they essentially store $S$ in a bit vector. This also shows up for $w = 40$ (but less pronounced), where they become the only data structure clearly faster than the burst tries. Our y-fast tries with unsorted buckets of size $2^9$ are about 6% faster than burst tries, but still require considerably less memory even respecting that 40 bits per key in burst tries are for associated values. It appears that y-fast tries with unsorted buckets scale best with the size of the universe: for 64-bit keys, it is the fastest data structure with buckets of size either $2^6$ or $2^9$ and is approximately 30% faster than burst tries, again consuming significantly less memory. The y-fast tries with sorted buckets are about on par with our B-trees, which are overall between 14% and 56% slower than y-fast tries with unsorted buckets.

For 32-bit keys, we intended to include the stratified tree [9], but it failed to stay within the memory limits (256 GB) starting at $2^{30}$ keys. For $2^{29}$ keys, it consumed about 1,480 bits per key, ranking lowest with an average throughput of circa $2^{17.7}$ operations per second.

**Conclusions.** Our dynamic predecessor data structures are the most memory efficient of all tested. They clearly outperform the STL set and for all universes in question, at least one of our data structures is faster than burst tries, the previously fastest known to us. We confirm once more [4, 10, 11, 15] that naïve solutions can be more practical than sophisticated data structures on modern hardware and sufficiently small inputs. We also observed that SIMD instructions, while faster than sequences of *classic* (SISD) instructions when used in batches, may turn out less useful in more complex scenarios.

## References

1 Advanced Micro Devices Inc. *AMD64 Architecture – Programmer's Manual Volume 3: General-Purpose and System Instructions*, September 2020. URL: `https://www.amd.com/system/files/TechDocs/24594.pdf`.

2 Miklós Ajtai, Michael L. Fredman, and János Komlós. Hash functions for priority queues. *Inf. Control.*, 63(3):217–225, 1984. `doi:10.1016/S0019-9958(84)80015-7`.

3 Arm Limited. *A64 Instruction Set Reference*, 2018. URL: `https://developer.arm.com/documentation/100076/0100/a64-instruction-set-reference`.

4 Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 12:1–12:16. Dagstuhl, 2017. `doi:10.4230/LIPIcs.SEA.2017.12`.

5 Djamal Belazzougui. Predecessor search, string algorithms and data structures. In *Encyclopedia of Algorithms*, pages 1605–1611. Springer, 2016. `doi:10.1007/978-1-4939-2864-4_632`.

6 Pedro Celis, Per-Åke Larson, and J. Ian Munro. Robin hood hashing (preliminary report). In *26th Symposium on Foundations of Computer Science (FOCS)*, pages 281–288. IEEE, 1985. `doi:10.1109/SFCS.1985.48`.

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

8 Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–14. ACM, 1997. `doi:10.1145/263105.263133`.

9 Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit key. In *6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 142–151. SIAM, 2004. URL: `https://web.archive.org/web/20201111145353/http://algo2.iti.kit.edu/dementiev/files/veb.pdf`.

10 Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical performance of space efficient data structures for longest common extensions.

In *28th European Symposium on Algorithms (ESA)*, pages 39:1–39:20. Dagstuhl, 2020. doi:10.4230/LIPIcs.ESA.2020.39.

11 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017. doi:10.1016/j.jda.2016.09.002.

12 E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, 1960. doi:10.1145/367390.367400.

13 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

14 Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002. doi:10.1145/506309.506312.

15 Lucian Ilie and Liviu Tinta. Practical algorithms for the longest common extension problem. In *16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 302–309. Springer, 2009. doi:10.1007/978-3-642-03784-9_30.

16 Intel Corporation. *Intel (R) 64 and IA-32 Architectures – Software Developer's Manual – Volume 2: Instruction Set Reference, A-Z*, September 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

17 Maureen Korda and Rajeev Raman. An experimental evaluation of hybrid data structures for searching. In *3rd International Workshop on Algorithm Engineering (WAE)*, pages 214–228. Springer, 1999. doi:10.1007/3-540-48318-7_18.

18 Kurt Maly. Compressed tries. *Commun. ACM*, 19(7):409–415, 1976. doi:10.1145/360248.360258.

19 Nicholas Nash and David Gregg. Comparing integer data structures for 32- and 64-bit keys. *ACM J. Exp. Algorithmics*, 15, 2010. doi:10.1145/1671970.1671977.

20 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5), 2020. doi:10.1145/3409371.

21 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *31st Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

22 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th Symposium on Foundations of Computer Science (FOCS)*, pages 166–175. IEEE, 2014. doi:10.1109/FOCS.2014.26.

23 Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979. doi:10.1145/359168.359175.

24 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Symposium on Foundations of Computer Science (FOCS)*, pages 75–84. IEEE, 1975. doi:10.1109/SFCS.1975.26.

25 M. Wenzel. Wörterbücher für ein beschränktes Universum (dictionaries for a bounded universe). *Master's thesis, Saarland University, Germany*, 1992.

26 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inform. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

## A Additional Results

We give experimental results in addition to those presented in Sections 4 through 6, which have been omitted there for the sake of clarity as they lead to largely the same conclusions. Figure 8 shows results for our y-fast tries (Section 5) for 32-bit and 40-bit universes, respectively, Figure 9 for our fusion and B-trees (Section 6).

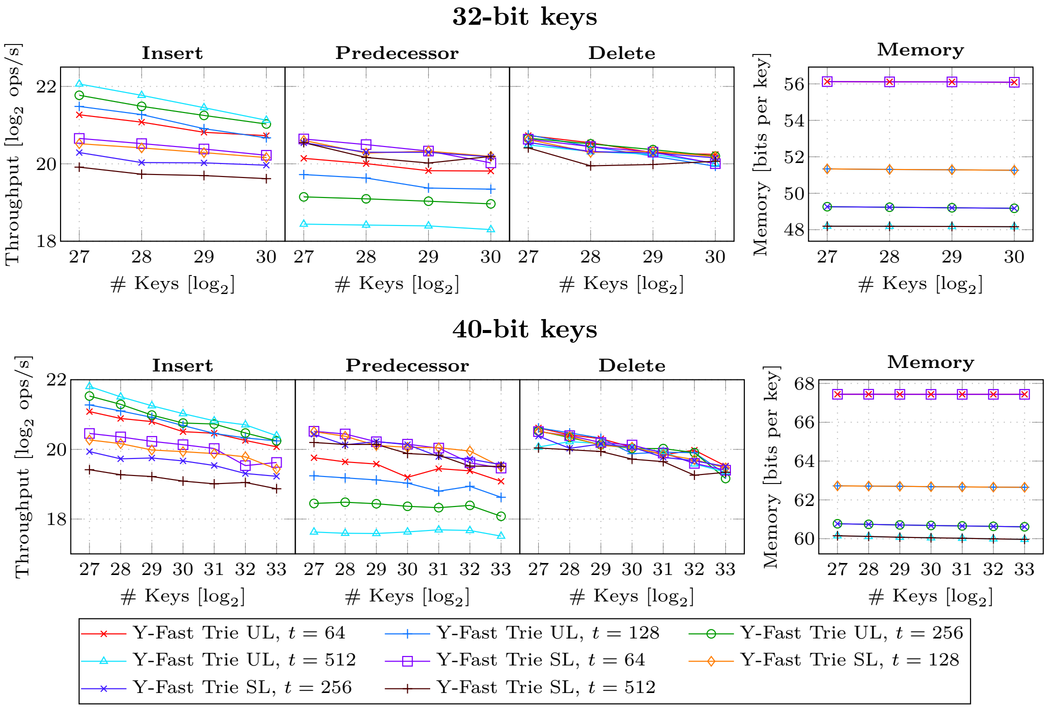


**Figure 8** Throughputs for the insert, predecessor and delete operations, as well as memory usage of the y-fast trie for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour.

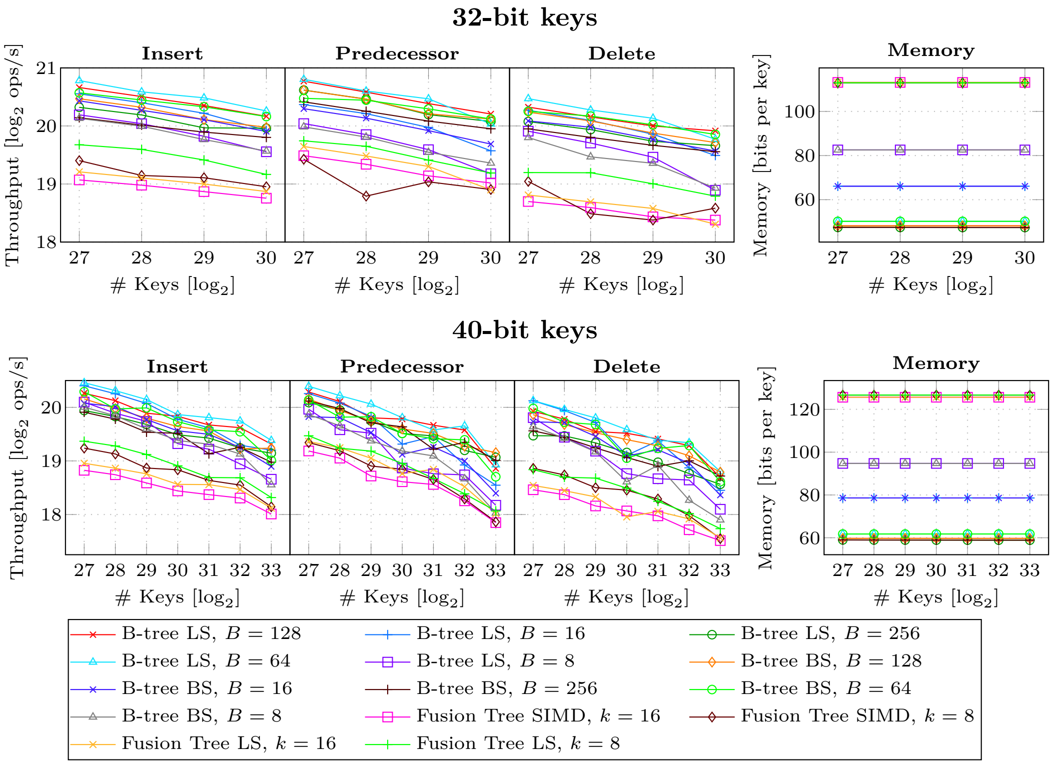


**Figure 9** Throughputs for the insert, predecessor and delete operations, as well as memory usage of the fusion and B-trees for 32-bit (top) and 40-bit keys (bottom). Best viewed in colour.

## B Choosing Parameters For Universe Sampling

The key question for preparing the experiments for our sampling data structure (Section 4) is how to configure the bucket size $b$, which is a direct trade-off parameter for the performance of the top level versus that of the bucket level: at the top level, we have worst-case costs of $\mathcal{O}(u/b)$ for updates or queries depending on the chosen data structure. At the bucket level, we have costs of $\mathcal{O}(b)$ for queries. To that end, we want to pick $b$ large enough so that the top level does not end up with too many entries, and pick it small enough so that operations on the bucket level do not take too much time.

We first do some considerations on the top level. When we assume a uniform distribution of keys inserted into the data structure, we observe that the number of insertions required until every bucket is active is distributed geometrically. What follows is that long scans for active buckets on the top level occur more and more rarely as more keys are inserted into the data structure. The same conclusion can be drawn when inserted keys are skewed towards a range within $U$, because then it occurs rarely that active buckets far away from others need to be accessed. Therefore, assuming that a large enough number of keys is going to be inserted so that long top-level scans occur rarely, we focus on bucket-level performance.

On the bucket level, we have to consider our three strategies for maintaining the keys. First, when using an unsorted list, smaller buckets clearly result in faster query times. In preliminary experiments, the performance declined only marginally up to a bucket size of $b_{\text{list}} := 2^{10}$ keys, whereas buckets any larger caused a significant drop. When using a bit vector, we benefit from scanning through bits packed into words, resulting in large buckets of $b_{\text{bv}} := 2^{24}$ keys still performing very well. Here, choosing larger buckets caused insertions and deletions to become slower. Because these operations simply mean setting or clearing bits, this effect can be explained by a higher number of cache misses. In the hybrid case, the initial notion is that we want the unsorted list to never consume more memory than the bit vector and thus switch to when a bucket exceeds $b/\lg b$ keys. As we desire to maintain the sweet spot threshold of $\theta_{\max} := b_{\text{list}} = 2^{10}$ for unsorted lists, we seek $b_{\text{hybrid}}$ such that $b_{\text{hybrid}}/\lg b_{\text{hybrid}} > 2^{10}$. This is the case for $b_{\text{hybrid}} \geq 2^{14}$, such that the bucket size of $b_{\text{bv}} = 2^{24}$ is again an option. However, consider the case where the bucket is switched from an unsorted list representation to a bit vector: we want to avoid a sudden explosion of memory occupied – and potentially wasted – by a bucket. Because a *perfect* choice of $b_{\text{hybrid}}$ cannot be done without any prior knowledge about the input, we explore different configurations.

We add here that we also tried *sorted* lists for maintaining the keys in the buckets, enabling binary search to speed up queries. However, the performance of updates greatly suffered and for smaller buckets, the query speedup compared to linear scans became marginal. Sorted lists are therefore not considered in our experiments.

## C Elaboration On Dynamic Fusion Nodes

We expand on the description of dynamic fusion nodes from Section 6. Let $\hat{x}^?$ indicate a compressed key that may contain don't cares. The $k \times k$ matrix $\hat{S}^?$ over the alphabet $\{\texttt{0}, \texttt{1}, \texttt{?}\}$ is represented by two $k \times k$ binary matrices Branch and Free defined as follows:

$$\textsc{Free}_{ij} = \begin{cases} 0 & \text{, if } \hat{S}^?_{ij} \neq \texttt{?} \\ 1 & \text{, if } \hat{S}^?_{ij} = \texttt{?} \end{cases} \qquad \textsc{Branch}_{ij} = \begin{cases} \hat{S}^?_{ij} & \text{, if } \textsc{Free}_{ij} = 0 \\ 0 & \text{, if } \textsc{Free}_{ij} = 1 \end{cases}$$

Intuitively, Free identifies the don't care bits in $\hat{S}^?$, and $\textsc{Branch}_{ij}$ is either equal to $\hat{S}^?_{ij}$ if a bit is used for branching, or zero if it is a don't care bit. The concatenation of the bits on the $i$-th *row* of $\hat{S}^?$ represent the compressed (with don't cares) $i$-th key contained in $S$.

We can find the predecessor of some key $x \in U$ by determining its rank $i < k$ among the keys in $S$. In the following, we reduce this rank query to compressed keys. To that end, we assume that $\hat{S}^?$ is maintained such that the rows are in ascending order. If this is not the case, we can afford to maintain an index as described in [2] without worsening the asymptotically constant query and update times. We now seek the number $i'$ of the row in $\hat{S}^?$ that corresponds to the rank of $\hat{x}$ among the compressed keys. We say that $\hat{x} \in \{0,1\}^k$ *matches* a compressed key $\hat{y}^? \in \{0,1,\texttt{?}\}^k$ with don't cares if all non-don't care bits in $\hat{y}^?$ are equal to the corresponding bits in $\hat{x}$. Formally, this is the case if $\forall j < k : \hat{y}^?\langle j\rangle = \texttt{?} \vee \hat{x}\langle j\rangle = \hat{y}^?\langle j\rangle$. We define the operation $\mathrm{match}(x)$ that, simultaneously for all $j < k$, tests whether $\hat{x}$ matches the compressed key encoded in the $j$-th row of $\hat{S}^?$ and reports the smallest $j$ where this is not the case. Pătraşcu and Thorup [22] show how to perform this operation in constant time by

**(1)** computing $\hat{S}^{\hat{x}}$ by replacing the don't care bits in $\hat{S}^?$ by the corresponding bits of the $k \times k$ bit matrix $\hat{x}^k$ that contains $k$ copies of $\hat{x}$ and

**(2)** performing a parallel row-wise greater-than comparison of $\hat{S}^{\hat{x}}$ against $\hat{x}^k$.

We then have $i' = \mathrm{match}(x)$. If $x \in S$, then $i'$ is also the rank of $x$ within $S$ as shown in [13]. Therefore, if $x = S[i']$, we already found the predecessor of $x$ after one match operation. However, if $x \neq S[i']$, it is $x \notin S$. In the trie, consider the ancestor of the leaf of $x$ – if $x$ were contained in $S$ – at level $j = \mathrm{msb}(x \oplus S[i'])$. At this node, we branched off in a direction that does not necessarily lead us to the predecessor of $x$. To see examples of this, refer to Examples 3 and 4 below. To find the actual rank $i$ of $x$ within $S$ and thus the predecessor $S[i]$ of $x$, we simulate the necessary trie navigation by performing another match operation. Consider the case where $x < S[i']$. In the trie, we navigate up to the lowest ancestor $v$ that has two children and take the path to the rightmost leaf in the left subtrie of $v$. An equivalent approach is to take the path to the leftmost leaf in the right subtrie of $v$, and subtract one from that leaf's rank. The latter can be simulated by computing $i = \mathrm{match}(x \wedge 1^{w-j}0^j) - 1$. In the case that $x > S[i']$, symmetrically, we simulate navigation to the rightmost leaf in the left subtrie of $v$ by computing $i = \mathrm{match}(x \vee 0^{w-j}1^j)$.

Pătraşcu and Thorup further show how to perform all the necessary manipulations of $\hat{S}^?$ in constant time in order to insert keys into the data structure. The intuition is always that $\hat{S}^?$ is stored in two words and all required word operations can be done in constant time.

▶ **Example 3.** We consider a predecessor search for $x = 25$ in the set $S$ from Figure 5 with $w = 5$. The binary representation of $x$ is `11001`, which we compress to $\hat{x} = \texttt{111}$. We compute $i' = \mathrm{match}(x) = 4$, corresponding to $S[i'] = 27$. This cannot be the predecessor of $x$ because $x < S[i']$. The position at which we branched off in the wrong direction is $j = \mathrm{msb}(x \oplus y) = 2$, at the node two levels above the leaf labeled by 27 in Figure 5a: a path leading to $x = 25$ would branch off to the left, whereas we branched off to the right. We simulate the necessary trie navigation by computing $i = \mathrm{match}(x \wedge 1^{w-j}0^j) - 1 = \mathrm{match}(\texttt{11000}) - 1 = 3$. Now, $S[i] = 12$ is the correct predecessor of $x$.

▶ **Example 4.** We consider a predecessor search for $x = 4$ in the set $S$ from Figure 5 with $w = 5$. The binary representation of $x$ is `00100`, which we compress to $\hat{x} = \texttt{000}$. We compute $i' = \mathrm{match}(x) = 1$, corresponding to $S[i'] = 2$. Since $x > S[i']$, we have the opposite case as in Example 3. The position at which we branched off in the wrong direction is $j = \mathrm{msb}(x \oplus y) = 2$, at the node two levels above the leaf labeled by 2 in Figure 5a. We compute $i = \mathrm{match}(x \vee 0^{w-j}1^j) = \mathrm{match}(\texttt{00111}) = 2$, and $S[i] = 3$ is the predecessor of $x$.

## Deleting keys

Pătraşcu and Thorup thoroughly describe the process of inserting a key into a dynamic fusion node in constant time [22]. They further claim that to delete a key, "we just have to invert the [...] process". However, some details require special attention, which is why we sketch the constant-time deletion of a key here. To that end, we use the same bag of tricks to perform the necessary $k \times k$ matrix manipulations in constant time using word operations and refer to [22] for the ideas.

Consider deleting a key $x \in U$ from our set $S$ containing at most $k$ keys from $U$. Recall that $S$ is stored in a $k \times k$ matrix $\hat{S}^{?}$ of bits and don't cares represented by two words Branch and Free, and that we maintain the $k$ distinguishing positions by setting the corresponding bits in a mask $M$ of $w$ bits. We first compute the rank $i = \mathrm{match}(x)$ of $x$ within $S$ in constant time. To verify that $x \in S$, we compare $x$ against $S[i]$. If $x \notin S$, we abort the deletion. Otherwise, we require the position $j$ of the least significant *distinguishing* bit at which $x$ branches off in the trie. This position may no longer be distinguishing after the deletion of $x$ and the, the corresponding bit must be removed from all remaining compressed keys in $\hat{S}^{?}$ to retain optimal compression. If $j$ remains a distinguishing position, we need to replace the corresponding bits in all keys in the subtrie beneath node $v$ by don't cares, where $v$ is the ancestor of the leaf corresponding to $x$ on level $j$. This is because due to the deletion of $x$, $v$ loses a child and is no longer a branching node. We will not consider any distinguishing positions of significance higher than $j$, because for those, by construction, there must be at least one other key in $S$ that branches off the corresponding trie node. The deletion of $x$ works as follows:

1. Find the position $j$ of the least significant distinguishing bit at which $x$ branches off in the trie. This corresponds to the position of the least significant non-don't care bit in $\hat{x}^{?}$. Compute $h$ by adding one to the number of trailing don't cares in $\hat{x}^{?}$, which equals the number of trailing ones in the $i$-th row of Free. Then, $j = \mathrm{select}_1(M, h)$.
2. Remove the $i$-th row, which contains $\hat{x}^{?}$, from $\hat{S}^{?}$.
3. Test if the deletion of $x$ results in $j$ no longer being a distinguishing position. This is the case if all non-don't care bits in the $h$-th column have the same value, indicating that there are no more branches at any node in the trie on level $j$. This can be done in constant time using a sequence of word operations; we refer to our code for details.
4. If that is the case, remove the $h$-th column from $\hat{S}^{?}$ and clear the $j$-th bit in $M$.
5. Otherwise, if $j$ remains distinguishing, find the range $i_0$ to $i_1$ of keys in the subtree beneath $j$. This range must contain at least one key, because otherwise column $h$ in row $i$ would have been a don't care. For all compressed keys in the range, column $h$ must be updated to a a don't care.

Compared to [22], we introduced two additional operations on words: counting trailing ones and a binary select operation. In Section 2, we already mentioned briefly how these can be performed in constant time both in theory and practice.

▶ **Example 5.** We consider the deletion of key $x = 12$ in *Figure* 5. It has rank $i = 3$ and occupies the third row in the matrix. We follow the steps of our sketched algorithm:

1. Observe that $\hat{x}^{?} = \texttt{01?}$ has one trailing don't care, so we have $h = 2$. The position of the least significant distinguishing bit at which $x$ branches off is thus $j = \mathrm{select}_1(M, h) = 2$. This corresponds to the second level in the trie shown in Figure 5a.
2. We remove the third row from $\hat{S}^{?}$, conceptually removing the leaf for $x = 12$ in the trie.

3. Observe how all non-don't care bits in column $h = 2$ of the matrix now have the same value 0. This corresponds to the fact that in the trie, on level $h = 2$, there are no longer any branches, which means that position $j$ is no longer distinguishing.
4. Because $j$ is no longer distinguishing, we remove the second column from the matrix completely and clear the corresponding bit in $M$.

The mask indicating distinguishing bits is now `10001`, and the matrix now consists of the compressed keys `00` (for key 2), `01` (for key 3) and `1?` (for key 27).

# Multilevel Hypergraph Partitioning with Vertex Weights Revisited

Tobias Heuer
Karlsruhe Institute of Technology, Germany

Nikolai Maas
Karlsruhe Institute of Technology, Germany

Sebastian Schlag
Karlsruhe Institute of Technology, Germany

## Abstract

The balanced hypergraph partitioning problem (HGP) is to partition the vertex set of a hypergraph into $k$ disjoint blocks of bounded weight, while minimizing an objective function defined on the hyperedges. Whereas real-world applications often use vertex and edge weights to accurately model the underlying problem, the HGP research community commonly works with unweighted instances.

In this paper, we argue that, in the presence of vertex weights, current balance constraint definitions either yield infeasible partitioning problems or allow unnecessarily large imbalances and propose a new definition that overcomes these problems. We show that state-of-the-art hypergraph partitioners often struggle considerably with weighted instances and tight balance constraints (even with our new balance definition). Thus, we present a recursive-bipartitioning technique that is able to reliably compute balanced (and hence feasible) solutions. The proposed method balances the partition by pre-assigning a small subset of the heaviest vertices to the two blocks of each bipartition (using an algorithm originally developed for the job scheduling problem) and optimizes the actual partitioning objective on the remaining vertices. We integrate our algorithm into the multilevel hypergraph partitioner `KaHyPar` and show that our approach is able to compute balanced partitions of high quality on a diverse set of benchmark instances.

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

**Keywords and phrases** multilevel hypergraph partitioning, balanced partitioning, vertex weights

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.8

**Supplementary Material** *Software (Source Code)*: `https://github.com/kahypar/kahypar`
archived at `swh:1:dir:c66f21225656426f1f902c5dc86404a622964a04`
*Dataset (Benchmark Set & Experimental Results)*: `http://algo2.iti.kit.edu/heuer/sea21/`

## 1 Introduction

Hypergraphs are a generalization of graphs where each hyperedge can connect more than two vertices. The $k$-way hypergraph partitioning problem (HGP) asks for a partition of the vertex set into $k$ disjoint blocks, while minimizing an objective function defined on the hyperedges. Additionally, a balance constraint requires that the weight of each block is smaller than or equal to a predefined upper bound (most often $L_k := (1+\varepsilon)\lceil\frac{c(V)}{k}\rceil$ for some parameter $\varepsilon$, where $c(V)$ is the sum of all vertex weights). The hypergraph partitioning problem is NP-hard [32] and it is even NP-hard to find good approximations [8]. The most commonly used heuristic to solve HGP in practice is the multilevel paradigm [1, 11, 29] which consists of three phases: First, the hypergraph is *coarsened* to obtain a hierarchy of smaller hypergraphs. After an *initial partitioning* algorithm is applied to the smallest hypergraph, *coarsening* is undone, and, at each level, *refinement* algorithms are used to improve the quality of the solution.

The two most prominent application areas of HGP are very large scale integration (VLSI) design [3, 29] and parallel computation of the sparse matrix-vector product [11]. In the former, HGP is used to divide a circuit into two or more blocks such that the number of external wires interconnecting circuit elements in different blocks is minimized. In this setting, each vertex is associated with a weight equal to the area of the respective circuit element [2] and tightly-balanced partitions minimize the total area required by the physical circuit [18]. In the latter, HGP is used to optimize the communication volume for parallel computations of sparse matrix-vector products [11]. In the simplest hypergraph model, vertices correspond to rows and hyperedges to columns of the matrix (or vice versa) and a partition of the hypergraphs yields an assignment of matrix entries to processors [11]. The work of a processor (which can be measured in terms of the number of non-zero entries [7]) is integrated into the model by assigning each vertex a weight equal to its degree [11]. Tightly-balanced partitions hence ensure that the work is distributed evenly among the processors.

Despite the importance of weighted instances for real-world applications, the HGP research community mainly uses unweighted hypergraphs in experimental evaluations [38]. The main rationale hereby being that even unweighted instances become weighted implicitly due to vertex contractions during the coarsening phase. Many partitioners therefore incorporate techniques that prevent the formation of heavy vertices [13, 24, 27] during coarsening to facilitate finding a feasible solution during the initial partitioning phase [38]. However, in practice, many weighted hypergraphs derived from real-world applications already contain heavy vertices – rendering the mitigation strategies of today's multilevel hypergraph partitioners ineffective. The popular ISPD98 VLSI benchmark set [2], for example, includes instances in which vertices can weigh up to 10% of the total weight of the hypergraph.

**Contributions and Outline.** After introducing basic notation in Section 2 and presenting related work in Section 3, we first formulate an alternative balance constraint definition in Section 4 that overcomes some drawbacks of existing definitions in presence of vertex weights. In Section 5, we then present an algorithm that enables partitioners based on the recursive bipartitioning (RB) paradigm to reliably compute balanced partitions for weighted hypergraphs. Our approach is based on the observation that usually only a small subset of the heaviest vertices is *critical* to satisfy the balance constraint. We show that pre-assigning these vertices to the two blocks of each bipartition (i.e., treating them as *fixed vertices*) and optimizing the actual objective function on the remaining vertices yields provable balance guarantees for the resulting $k$-way partition. We implemented our algorithms in the open source HGP framework `KaHyPar` [38]. The experimental evaluation presented in Section 6 shows that our new approach (called `KaHyPar-BP`) is able to compute balanced partitions for all instances of a large real-world benchmark set (without increasing the running time or decreasing the solution quality), while other partitioners such as the latest versions of `KaHyPar`, `hMetis`, and `PaToH` produced imbalanced partitions on 4.9% up to 42% of the instances for $\varepsilon = 0.01$ (4.3% up to 23.1% for $\varepsilon = 0.03$). Section 7 concludes the paper.

## 2 Preliminaries

A *weighted hypergraph* $H = (V, E, c, \omega)$ is defined as a set of vertices $V$ and a set of hyperedges/nets $E$ with vertex weights $c : V \to \mathbb{R}_{>0}$ and net weights $\omega : E \to \mathbb{R}_{>0}$, where each net $e$ is a subset of the vertex set $V$ (i.e., $e \subseteq V$). We extend $c$ and $\omega$ to sets in the natural way, i.e., $c(U) := \sum_{v \in U} c(v)$ and $\omega(F) := \sum_{e \in F} \omega(e)$. Given a subset $V' \subseteq V$, the *subhypergraph* $H_{V'}$ is defined as $H_{V'} := (V', \{e \cap V' \mid e \in E : e \cap V' \neq \emptyset\}, c, \omega)$.

A *$k$-way partition* of a hypergraph $H$ is a partition of the vertex set $V$ into $k$ non-empty disjoint subsets $\Pi_k = \{V_1, \dots, V_k\}$. We refer to a $k$-way partition $\Psi_k = \{P_1, \dots, P_k\}$ of a subset $P \subseteq V$ as a *$k$-way prepacking*. We call a vertex $v \in P$ a *fixed* vertex and a vertex $v \in V \setminus P$ an *ordinary* vertex. During partitioning, fixed vertices are not allowed to be moved to a different block of the partition. A $k$-way partition $\Pi_k$ is *$\varepsilon$-balanced* if each block $V_i$ satisfies the *balance constraint*: $c(V_i) \leq L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ for some parameter $\varepsilon$. The *$k$-way hypergraph partitioning problem* initialized with a $k$-way prepacking $\Psi_k = \{P_1, \dots, P_k\}$ is to find an $\varepsilon$-balanced $k$-way partition $\Pi_k = \{V_1, \dots, V_k\}$ of a hypergraph $H$ that minimizes an objective function and satisfies that $\forall i \in \{1, \dots, k\} : P_i \subseteq V_i$. In this paper, we optimize the *connectivity* metric $(\lambda - 1)(\Pi) := \sum_{e \in E}(\lambda(e) - 1)\,\omega(e)$, where $\lambda(e) := |\{V_i \in \Pi \mid V_i \cap e \neq \emptyset\}|$.

The *most balanced partition problem* is to find a $k$-way partition $\Pi_k$ of a weighted hypergraph $H = (V, E, c, \omega)$ such that $\max(\Pi_k) := \max_{V' \in \Pi_k} c(V')$ is minimized. For an optimal solution $\Pi_{\mathrm{OPT}}$ it holds that there exists no other $k$-way partition $\Pi'_k$ with $\max(\Pi'_k) < \max(\Pi_{\mathrm{OPT}})$. We use $\mathrm{OPT}(H, k) := \max(\Pi_{\mathrm{OPT}})$ to denote the weight of the heaviest block of an optimal solution. Note that the problem is equivalent to the most common version of the *job scheduling* problem: Given a sequence $J = \langle j_1, \dots, j_n \rangle$ of $n$ computing jobs each associated with a *processing time* $p_i$ for $i \in [1, n]$, the task is to find an assignment of the $n$ jobs to $k$ identical machines (each job $j_i$ runs exclusively on a machine for exactly $p_i$ time units) such that the latest completion time of a job is minimized.

## 3 Related Work

In the following, we will focus on work closely related to our main contributions. For an extensive overview on hypergraph partitioning we refer the reader to existing literature [3, 5, 35, 38]. Well-known multilevel HGP software packages with certain distinguishing characteristics include `PaToH` [4, 11] (originating from scientific computing), `hMetis` [29, 30] (originating from VLSI design), `KaHyPar` [26, 27] (general purpose, $n$-level), `Moondrian` [41] (sparse matrix partitioning), `UMPa` [14] (multi-objective) and `Zoltan` [16] (distributed partitioner).

**Partitioning with Vertex Weights.** The most widely used techniques to improve the quality of a $k$-way partition are move-based local search heuristics [19, 31] that greedily move vertices according to a *gain* value (i.e., the improvement in the objective function). Vertex moves violating the balance constraint are usually rejected, which can significantly deteriorate solution quality in presence of varying vertex weights [10]. This issue is addressed using techniques that allow intermediate balance violations [18] or use temporary relaxations of the balance constraint [9, 10]. Caldwell et al. [10] proposed to preassign each vertex with a weight greater than the average block weight $L_k$ to a seperate block before partitioning (treated as fixed vertices) and build the actual $k$-way partition around them. All of these techniques were developed and evaluated for flat (i.e., non-multilevel) partitioning algorithms. In the multilevel setting, even unweighted instances become implicitly weighted due to vertex contractions in the coarsening phase, which is why the formation of heavy vertices is prevented by penalizing the contraction of vertices with large weights [13, 24, 40] or enforcing a strict upper bound for vertex weights throughout the coarsening process [1, 27]. If the input hypergraph is unweighted, the aforementioned techniques often suffice to find a feasible solution [38]. `PaToH` [12] additionally uses bin packing techniques during initial partitioning.

**Job Scheduling Problem.** The job scheduling problem is NP-hard [20] and we refer the reader to existing literature [23, 36] for a comprehensive overview of the research topic. In this work, we make use of the *longest processing time* (LPT) algorithm proposed by Graham [22].

We will explain the algorithm in the context of the most balanced partition problem defined in Section 2: For a weighted hypergraph $H = (V, E, c, \omega)$, the algorithm iterates over the vertices of $V$ sorted in decreasing vertex-weight order and assigns each vertex to the block of the $k$-way partition with the lowest weight. The algorithm can be implemented to run in $\mathcal{O}(|V| \log |V|)$ time, and for a $k$-way partition $\Pi_k$ produced by the algorithm it holds that $\max(\Pi_k) \leq (\frac{4}{3} - \frac{1}{3k})\text{OPT}(H, k)$.

**KaHyPar.** The **Ka**rlsruhe **Hy**pergraph **Par**titioning framework takes the multilevel paradigm to its extreme by only contracting a single vertex in every level of the hierarchy. `KaHyPar` provides recursive bipartitioning [37] as well as direct $k$-way partitioning algorithms [1] (direct $k$-way uses RB in the initial partitioning phase). It uses a community detection algorithm as preprocessing step to restrict contractions to densely connected regions of the hypergraph during coarsening [27]. Furthermore, it employs a portfolio of bipartitioning algorithms for initial partitioning of the coarsest hypergraph [25, 37], and, during the refinement phase, improves the partition with a highly engineered variant of the classical FM local search [1] and a refinement technique based on network flows [21, 26].

During RB-based partitioning, `KaHyPar` ensures that the solution is balanced by adapting the imbalance ratio for each bipartition individually. Let $H_{V'}$ be the subhypergraph of the current bipartition that should be partitioned recursively into $k' \leq k$ blocks. Then,

$$\varepsilon' := \left( (1+\varepsilon)\frac{c(V)}{k} \cdot \frac{k'}{c(V')} \right)^{\frac{1}{\lceil \log_2(k') \rceil}} - 1 \tag{1}$$

is the imbalance ratio used for the bipartition of $H_{V'}$. The equation is based on the observation that the worst-case block weight of the resulting $k'$-way partition of $H_{V'}$ obtained via RB is smaller than $(1+\varepsilon')^{\lceil \log_2(k') \rceil} \frac{c(V')}{k'}$, if $\varepsilon'$ is used for all further bipartitions. Requiring that this weight must be smaller or equal to $L_k = (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ leads to the formula defined in Equation 1.

## 4 A New Balance Constraint For Weighted Hypergraphs

A $k$-way partition of a weighted hypergraph $H = (V, E, c, \omega)$ is balanced, if the weight of each block is below some predefined upper bound. In the literature, the most commonly used bounds are $L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ (standard definition) and $L_k^{\max} := L_k + \max_{v \in V} c(v)$ [19, 38, 39]. The latter was initially proposed by Fiduccia and Mattheyses [19] for bipartitioning to ensure that the highest-gain vertex can always be moved to the opposite block.

Both definitions exhibit shortcomings in the presence of heavy vertices: As soon as the hypergraph contains even a single vertex with $c(v) > L_k$, no feasible solution exists when the block weights are constrained by $L_k$, while for $L_k^{\max}$ it follows that $L_k^{\max} > 2L_k$ – allowing large variations in block weights even if $\varepsilon$ is small. In the following, we therefore propose a new balance constraint that (i) guarantees the existence of an $\varepsilon$-balanced $k$-way partition and (ii) avoids unnecessarily large imbalances.

While the optimal solution of the most balanced partition problem would yield a partition with the best possible balance, it is not feasible in practice to use $L_k^{\text{OPT}} := (1+\varepsilon)\text{OPT}(H, k)$ as balance constraint, because finding such a $k$-way partition is NP-hard [20]. Hence, we propose to use the bound provided by the LPT algorithm instead:

$$L_k^{\text{LPT}} := (1+\varepsilon)\ \text{LPT}(H, k) \leq \left( \frac{4}{3} - \frac{1}{3k} \right) L_k^{\text{OPT}}. \tag{2}$$

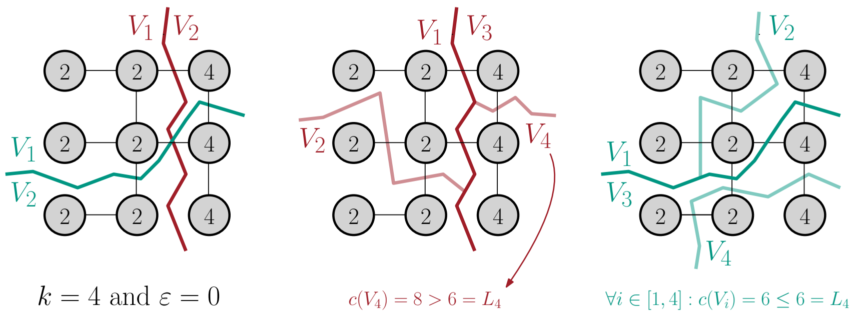

$V_1$
$V_2$
$k = 4$ and $\varepsilon = 0$
$V_3$
$V_4$
$c(V_4) = 8 > 6 = L_4$
$\forall i \in [1,4] : c(V_i) = 6 \leq 6 = L_4$

**Figure 1** Illustration of a deeply (left, green line) and a non-deeply balanced bipartition (left, red line). The numbers in each circle denotes the vertex weights. In both cases, the hypergraph is partitioned into $k = 4$ blocks with $\varepsilon = 0$ via recursive bipartitioning. Thus, the weight of heaviest block must be smaller or equal to $L_4 = 6$ and for the first bipartition, we use $L_2 = 12$ as an upper bound.

Note that if the hypergraph is unweighted, the LPT algorithm will always find an optimal solution with $\mathrm{OPT}(H, k) = \lceil \frac{|V|}{k} \rceil$ and thus, $L_k^{\mathrm{LPT}}$ is equal to $L_k$. Since all of today's partitioning algorithms bound the maximum block weight by $L_k$, Section 6 gives more details on how we employ this new balance constraint definition in our experimental evaluation.

## 5 Multilevel Recursive Bipartitioning with Vertex Weights Revisited

Most multilevel hypergraph partitioners either employ recursive bipartitioning directly [11, 16, 29, 37, 41] or use RB-based algorithms in the initial partitioning phase to compute an initial $k$-way partition of the coarsest hypergraph [1, 4, 14, 30]. In both settings, a $k$-way partition is derived by first computing a bipartition $\Pi_2 = \{V_1, V_2\}$ of the (input/coarse) hypergraph $H$ and then recursing on the subhypergraphs $H_{V_1}$ and $H_{V_2}$ by partitioning $V_1$ into $\lceil \frac{k}{2} \rceil$ and $V_2$ into $\lfloor \frac{k}{2} \rfloor$ blocks. Although `KaHyPar` adaptively adjusts the allowed imbalance at each bipartitioning step (using the imbalance factor $\varepsilon'$ as defined in Equation 1), an *unfortunate* distribution of the vertices in some bipartitions $\Pi_2$ can easily lead to instances for which it is impossible to find a balanced solution during the recursive calls – even though the current bipartition $\Pi_2$ satisfies the adjusted balance constraint. An example is shown in Figure 1 (left): Although the current bipartition (indicated by the red line) is perfectly balanced, it will not be possible to recursively partition the subhypergraph induced by the vertices of $V_2$ into two blocks of equal weight, because each of the three vertices has a weight of four.

To capture this problem, we introduce the notion of *deep balance*:

▶ **Definition 1.** (Deep Balance)**.** *Let $H = (V, E, c, \omega)$ be a weighted hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition, and let $H_{V'}$ be a subhypergraph of $H$ which should be partitioned into $k' \leq k$ blocks via recursive bipartitioning. A subhypergraph $H_{V'}$ is deeply balanced w.r.t. $k'$, if there exists a $k'$-way partition $\Pi_{k'}$ of $H_{V'}$ such that* $\max(\Pi_{k'}) \leq L_k := (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$. *A bipartition $\Pi_2 = \{V_1, V_2\}$ of $H_{V'}$ is deeply balanced w.r.t. $k'$, if the subhypergraphs $H_{V_1}$ and $H_{V_2}$ are deeply balanced with respect to $\lceil \frac{k'}{2} \rceil$ resp. $\lfloor \frac{k'}{2} \rfloor$.*

If a subhypergraph $H_{V'}$ is deeply balanced with respect to $k'$, there always exists a $k'$-way partition $\Pi_{k'}$ of $H_{V'}$ such that weight of the heaviest block satisfies the original balance constraint $L_k$ imposed on the partition of the input hypergraph $H$. Moreover, there also always exists a deeply balanced bipartition $\Pi_2 := \{V_1, V_2\}$ ($V_1$ is the union of the first $\lceil \frac{k'}{2} \rceil$

and $V_2$ of the last $\lfloor \frac{k'}{2} \rfloor$ blocks of $\Pi_{k'}$). Hence, a RB-based partitioning algorithm that is able to compute deeply balanced bipartitions on deeply balanced subhypergraphs will always compute $\varepsilon$-balanced $k$-way partitions (assuming the input hypergraph is deeply balanced).

**Deep Balance and Adaptive Imbalance Adjustments.** Computing deeply balanced bipartitions in the RB setting guarantees that the resulting $k$-way partition is $\varepsilon$-balanced. Thus, the concept of deep balance could replace the adaptive imbalance factor $\varepsilon'$ employed in `KaHyPar` [37] (see Equation 1). However, as we will see in the following example, combining both approaches gives the partitioner more flexibility (in terms of feasible vertex moves during refinement). Assume that we want to compute a 4-way partition via recursive bipartitioning and that the first bipartition $\Pi_2 := \{V_1, V_2\}$ is deeply balanced with $c(V_1) = (1+\varepsilon)\lceil \frac{c(V)}{2} \rceil$. The deep-balance property ensures that we can further partition $V_1$ into two blocks such that the weight of the heavier block is smaller than $L_4$. However, this bipartition has to be perfectly balanced:

$$L_2 = (1+\overline{\varepsilon})\left\lceil \frac{c(V_1)}{2} \right\rceil = (1+\overline{\varepsilon})\left\lceil \frac{(1+\varepsilon)\lceil \frac{c(V)}{2} \rceil}{2} \right\rceil \leq (1+\varepsilon)\left\lceil \frac{c(V)}{4} \right\rceil = L_4 \Rightarrow \overline{\varepsilon} \approx 0. \tag{3}$$

If we would have computed the first bipartition with an adjusted imbalance factor $\varepsilon'$, then $\max(\Pi_2) \leq (1+\varepsilon')\lceil \frac{c(V)}{2} \rceil = \sqrt{1+\varepsilon}\lceil \frac{c(V)}{2} \rceil$ – providing more flexibility for subsequent bipartitions. In the following, we therefore focus on computing deeply $\varepsilon'$-balanced bipartitions.

**Deep Balance and Multilevel Recursive Bipartitioning.** In general, computing a deeply balanced bipartition $\Pi_2 := \{V_1, V_2\}$ w.r.t. $k$ is NP-hard, as we must show that there exists a $k$-way partition $\Pi_k$ of $H$ with $\max(\Pi_k) \leq L_k$, which can be reduced to the most balanced partition problem presented in Section 2. However, we can first compute a $k$-way partition $\Pi_k := \{V'_1, \ldots, V'_k\}$ using the LPT algorithm, thereby approximating an optimal solution. If $\max(\Pi_k) \leq L_k$, we can then construct a deeply balanced bipartition $\Pi_2 = \{V_1, V_2\}$ by choosing $V_1 := V'_1 \cup \ldots \cup V'_{\lceil \frac{k}{2} \rceil}$ and $V_2 := V'_{\lceil \frac{k}{2} \rceil + 1} \cup \ldots \cup V'_k$. Unfortunately, this approach completely ignores the optimization of the objective function – yielding balanced partitions of low quality. If such a bipartition were to be used as initial solution in the multilevel setting, the objective could still be optimized during the refinement phase. However, this would necessitate that refinement algorithms are aware of the concept of deep balance and that they only perform vertex moves that don't destroy the deep-balance property of the starting solution. Since this is infeasible in practice, we propose a different approach that involves fixed vertices.

The key idea of our approach is to compute a prepacking $\Psi = \{P_1, P_2\}$ of the $m = |P_1| + |P_2|$ heaviest vertices of the hypergraph and to show that this prepacking suffices to ensure that each $\varepsilon'$-balanced bipartition $\Pi_2 = \{V_1, V_2\}$ with $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$ is deeply balanced. Note that the upcoming definitions and theorems are formulated from the perspective of the first bipartition of the input hypergraph $H$ to simplify notation. They can be generalized to subhypergraphs $H_{V'}$ in a similar fashion as was done in Definition 1. Furthermore, we say that the bipartition $\Pi_2 = \{V_1, V_2\}$ respects a prepacking $\Psi = \{P_1, P_2\}$, if $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$, and that the bipartition is balanced, if $\max(\Pi_2) \leq L_2 := (1+\varepsilon')\lceil \frac{c(V_1 \cup V_2)}{2} \rceil$ (with $\varepsilon'$ as defined in Equation 1). The following definition formalizes our idea.

▶ **Definition 2.** (Sufficiently Balanced Prepacking). *Let $H = (V, E, c, \omega)$ be a hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition via recursive bipartitioning. We call a prepacking $\Psi$ of $H$ sufficiently balanced if every balanced bipartition $\Pi_2$ respecting $\Psi$ is deeply balanced with respect to $k$.*

Our approach to compute $\varepsilon$-balanced $k$-way partitions is outlined in Algorithm 1. We first compute a bipartition $\Pi_2$. Before recursing on each of the two induced subhypergraphs, we check if $\Pi_2$ is deeply balanced using the LPT algorithm in a similar fashion as described in the beginning of this paragraph. If it is not deeply balanced, we compute a sufficiently balanced prepacking $\Psi$ and re-compute $\Pi_2$ – treating the vertices of the prepacking as fixed vertices. If this second bipartitioning call was able to compute a balanced bipartition, we found a deeply balanced partition and proceed to partition the subhypergraphs recursively.

Note that, in general, we may not detect that $\Pi_2$ is deeply balanced or fail to find a sufficiently balanced prepacking $\Psi$ or a balanced bipartition $\Pi_2$, since all involved problems are NP-hard. However, as we will see in Section 6, Algorithm 1 computes balanced partitions for all instances of our large real-world benchmark set. This seems to indicate that the above-mentioned problems only happen rarely in practice.

**Algorithm 1** Recursive Bipartitioning Algorithm.

```
Data: Hypergraph $H$ for which we seek an $\varepsilon$-balanced $k$-way partition and subhypergraph
      $H_{V'}$ of $H$ which is to be to bipartitioned recursively into $k' \leq k$ blocks.
Function recursiveBipartitioning($H$, $k$, $\varepsilon$ $H_{V'}$, $k'$):
    $L_2 \leftarrow (1+\varepsilon')\lceil \frac{c(V')}{2} \rceil$                        // with $\varepsilon'$ as defined in Equation 1
    $\Pi_2 := \{V_1, V_2\} \leftarrow$ multilevelBipartitioning($H_{V'}$, $L_2$, $\emptyset$)  // $\emptyset$ = empty prepacking
    if $k' = 2$ then return $\Pi_2$
    else if $\Pi_2$ is not deeply balanced w.r.t. $k'$ then
        $\Psi \leftarrow$ sufficientlyBalancedPrepacking($H$, $k$, $\varepsilon$, $H_{V'}$, $k'$)     // see Algorithm 2
        $\Pi_2 \leftarrow$ multilevelBipartitioning($H_{V'}$, $L_2$, $\Psi$)       // treating $\Psi$ as fixed vertices
    $\Pi_{k_1} \leftarrow$ recursiveBipartitioning($H$, $k$, $\varepsilon$, $H_{V_1}$, $k_1$) with $k_1 := \lceil \frac{k'}{2} \rceil$
    $\Pi_{k_2} \leftarrow$ recursiveBipartitioning($H$, $k$, $\varepsilon$, $H_{V_2}$, $k_2$) with $k_2 := \lfloor \frac{k'}{2} \rfloor$
    return $\Pi_{k_1} \cup \Pi_{k_2}$
```

**Computing a Sufficiently Balanced Prepacking.** The prepacking $\Psi$ is constructed by incrementally assigning vertices to $\Psi$ in decreasing order of weight and checking a property $\mathcal{P}$ after each assignment that, if satisfied, implies that the current prepacking is sufficiently balanced. In the proof of property $\mathcal{P}$ , we will extend a $k$-way prepacking $\Psi_k$ to an $\varepsilon$-balanced $k$-way partition $\Pi_k$ using the LPT algorithm and use the following upper bound on the weight of the heaviest block of $\Pi_k$.

▶ **Lemma 3.** (LPT Bound). *Let $H = (V, E, c, \omega)$ be a weighted hypergraph, $\Psi_k$ be a $k$-way prepacking for a set of fixed vertices $P \subseteq V$, and let $O := \langle v_1, \dots, v_m \mid v_i \in V \setminus P \rangle$ be the sequence of all ordinary vertices of $V \setminus P$ sorted in decreasing order of weight. If we assign the remaining vertices $O$ to the blocks of $\Psi_k$ by using the* LPT *algorithm, we can extend $\Psi_k$ to a $k$-way partition $\Pi_k$ of $H$ such that the weight of the heaviest block is bound by:*

$$\max(\Pi_k) \leq \max\{\frac{1}{k}c(P) + h_k(O), \max(\Psi_k)\}, \text{ with } h_k(O) := \max_{i \in \{1,\dots,m\}} c(v_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(v_j).$$

The proof of Lemma 3 can be found in Appendix A. $O$ is sorted in decreasing order of weight because for any permutation $O'$ of $O$, it holds that $h_k(O) \leq h_k(O')$ – resulting in the tightest bound for $\max(\Pi_k)$.

Assuming that the number $k$ of blocks is even (i.e., $k_1 = k_2 = k/2$) to simplify notation, the balance property $\mathcal{P}$ is defined as follows (the generalized version can be found in Appendix B):

▶ **Definition 4.** (Balance Property $\mathcal{P}$). *Let $H = (V, E, c, \omega)$ be a hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition and let $\Psi$ be a prepacking of $H$ for a set of fixed vertices $P \subseteq V$. Furthermore, let $O_t := \langle v_1, \dots, v_t \rangle$ be the sequence of the $t$ heaviest ordinary vertices of $V \setminus P$ sorted in decreasing order of weight such that $t$ is the smallest number that satisfies $\max(\Psi) + c(O_t) \geq L_2$ (see Line 2, Algorithm 1). We say that a prepacking $\Psi$ satisfies the balance property $\mathcal{P}$ if the following two conditions hold:*

**(i)** *the prepacking $\Psi$ is deeply balanced*

**(ii)** $\frac{1}{k/2} \max(\Psi) + h_{k/2}(O_t) \leq L_k$.

In the following, we will show that the LPT algorithm can be used to construct a $k/2$-way partition $\Pi_{k/2}$ for both blocks of any balanced bipartition $\Pi_2 = \{V_1, V_2\}$ that respects $\Psi$, such that the weight of the heaviest block can be bound by the left term of Condition (ii). This implies that $\max(\Pi_{k/2}) \leq L_k$ (right term of Condition (ii)) and thus proofs that any balanced bipartition $\Pi_2$ respecting $\Psi$ is deeply balanced. Note that choosing $t$ as the smallest number that satisfies $\max(\Psi) + c(O_t) \geq L_2$ minimizes the left term of Condition (ii) (since $h_k(O_t) \leq h_k(O_{t+1})$).

▶ **Theorem 5.** *A prepacking $\Psi$ of a hypergraph $H = (V, E, c, \omega)$ that satisfies the balance property $\mathcal{P}$ is sufficiently balanced with respect to $k$.*

**Proof.** For convenience, we use $k' := k/2$. Let $\Pi_2 = \{V_1, V_2\}$ be an abitrary balanced bipartition that respects the prepacking $\Psi = \{P_1, P_2\}$ with $\max(\Pi_2) \leq L_2$. Since $\Psi$ is deeply balanced (see Definition 4(i)), there exists a $k'$-way prepacking $\Psi_{k'}$ of $P_1$ such that $\max(\Psi_{k'}) \leq L_k$. We define the sequence of the ordinary vertices of block $V_1$ sorted in decreasing weight order with $O_1 := \langle v_1, \dots, v_m \mid v_i \in V_1 \setminus P_1 \rangle$. We can extend $\Psi_{k'}$ to a $k'$-way partition $\Pi_{k'}$ of $V_1$ by assigning the vertices of $O_1$ to the blocks in $\Psi_{k'}$ using the LPT algorithm. Lemma 3 then establishes an upper bound on the weight of the heaviest block.

$$\begin{aligned}\max(\Pi_{k'}) &\overset{\text{Lemma 3}}{\leq} \max\{\frac{1}{k'}c(P_1) + h_{k'}(O_1), \max(\Psi_{k'})\} \\ &\overset{\max(\Psi_{k'}) \leq L_k}{\leq} \max\{\frac{1}{k'}c(P_1) + h_{k'}(O_1), L_k\}\end{aligned}$$

Let $O_t$ be the sequence of the $t$ heaviest ordinary vertices of $V \setminus P$ with $P := P_1 \cup P_2$ as defined in Definition 4.

▷ Claim 6. It holds that: $\frac{1}{k'}c(P_1) + h_{k'}(O_1) \leq \frac{1}{k'}\max(\Psi) + h_{k'}(O_t)$.

For a proof of Claim 6 see Appendix C. We can conclude that

$$\frac{1}{k'}c(P_1) + h_{k'}(O_1) \overset{\text{Claim 6}}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t) \overset{\text{Definition 4(ii)}}{\leq} L_k.$$

This proves that the subhypergraph $H_{V_1}$ is deeply balanced. The proof for block $V_2$ can be done analogously, which then implies that $\Pi_2$ is deeply balanced. Since $\Pi_2$ is an abitrary balanced bipartition respecting $\Psi$, it follows that $\Psi$ is sufficiently balanced. ◀

Algorithm 2 outlines our approach to efficiently compute a sufficiently balanced prepacking $\Psi$. In Line 6, we compute a $k'$-way prepacking $\Psi_{k'}$ of the $i$ heaviest vertices with the LPT algorithm and if $\Psi_{k'}$ satisfies $\max(\Psi_{k'}) \leq L_k$, then Line 7 constructs a deeply balanced prepacking $\Psi$ (which fullfils Condition (i) of Definition 4). We store the blocks $P'_j$ of $\Psi_{k'}$ together with their weights $c(P'_j)$ as key in an addressable priority queue such that we can determine and update the block with the smallest weight in time $\mathcal{O}(\log k')$ (Line 6). In Line 9, we compute the smallest $t$ that satisfies $\max(\Psi) + c(O_t) \geq L_2$ via a binary search in logarithmic time over an array containing the vertex weight prefix sums of the sequence $O$, which can be precomputed in linear time. Furthermore, we construct a range maximum query data structure over the array $H_{k'/2} = \langle c(v_1), c(v_2) + \frac{1}{k'/2}c(v_1), \ldots, c(v_n) + \frac{1}{k'/2}\sum_{j=1}^{n-1} c(v_j)\rangle$. Caculating $h_{k'/2}(O_t)$ (Line 10) then corresponds to a range maximum query in the interval $[i+1, i+t]$ in $H_{k'/2}$, which can be answered in constant time after $H_{k'/2}$ has been precomputed in time $\mathcal{O}(n)$ [6]. In total, the running time of the algorithm is $\mathcal{O}(n(\log k' + \log n))$. Note that if the algorithm reaches Line 12, we could not proof that any of the intermediate constructed prepackings were sufficiently balanced, in which case $\Psi$ represents a bipartition of $H_{V'}$ computed by the LPT algorithm.

**Algorithm 2** Prepacking Algorithm.

```
Data: Hypergraph $H = (V, E, c, \omega)$ for which we seek an $\varepsilon$-balanced $k$-way partition and
      subhypergraph $H_{V'} = (V', E', c, \omega)$ of $H$ which is to be to bipartitioned recursively
      into $k' \leq k$ blocks.
1  Function sufficientlyBalancedPrepacking($H$, $k$, $\varepsilon$ $H_{V'}$, $k'$):
2      $\Psi = \langle P_1, P_2\rangle \leftarrow \langle\emptyset, \emptyset\rangle$ and $\Psi_{k'} = \langle P'_1, \ldots, P'_{k'}\rangle \leftarrow \langle\emptyset, \ldots, \emptyset\rangle$    // Initialization
3      $L_2 \leftarrow (1+\varepsilon')\lceil\frac{c(V')}{2}\rceil$ and $L_k \leftarrow (1+\varepsilon)\lceil\frac{c(V)}{k}\rceil$    // with $\varepsilon'$ as defined in Equation 1
4      $O \leftarrow \langle v_1, \ldots, v_n \mid v_i \in V'\rangle$    // $V'$ sorted in decreasing order of weight $\Rightarrow \mathcal{O}(n \log n)$
5      for $i = 1$ to $n$ do
6          Add $v_i \in O$ to bin $P'_j \in \Psi_{k'}$ with smallest weight    // LPT algorithm
7          $\Psi \leftarrow \{P'_1 \cup \ldots \cup P'_x, P'_{x+1} \cup \ldots \cup P'_{k'}\}$ with $x := \lceil\frac{k'}{2}\rceil$
8          if $\max(\Psi) \leq L_2$ and $\max(\Psi_{k'}) \leq L_k$ then    // $\Rightarrow \Psi$ is deeply ($\varepsilon'$-)balanced
9              $t \leftarrow \min(\{t \mid \max(\Psi) + c(O_t) \geq L_2\})$    // $O_t := \langle v_{i+1}, \ldots, v_{i+t}\rangle$
10             if $\frac{2}{k'}\max(\Psi) + h_{k'/2}(O_t) \leq L_k$ then    // Condition (ii) of Definition 4
11                 return $\Psi$    // $\Rightarrow \Psi$ is sufficiently balanced (Theorem 5)
12     return $\Psi$  // No sufficiently balanced prepacking found $\Rightarrow$ treat all vertices as fixed vertices
```

## 6 Experimental Evaluation

We integrated the prepacking technique (see Algorithms 1 and 2) into the recursive bipartitioning algorithm of `KaHyPar`. Our implementation is available from `http://www.kahypar.org`. The code is written in C++17 and compiled using g++9.2 with the flags `-mtune=native -O3 -march=native`. Since `KaHyPar` offers both a recursive bipartitioning and direct $k$-way partitioning algorithm (which uses the RB algorithm in the initial partitioning phase), we refer to the RB-version using our improvements as `KaHyPar-BP-R` and to the direct $k$-way version as `KaHyPar-BP-K` (`BP` = **B**alanced **P**artitioning).

**Instances.** The following experimental evaluation is based on two benchmark sets. The RealWorld benchmark set consists of 50 hypergraphs originating from the VLSI design and scientific computing domain. It contains instances from the ISPD98 VLSI Circuit Benchmark Suite [2] (18 instances), the DAC 2012 Routability-Driven Placement Benchmark Suite [42] (9 instances), 16 instances from the Stanford Network Analysis (SNAP) Platform [33], and 7 highly asymmetric matrices of Davis et al. [15] (referred to as ASM). For VLSI instances

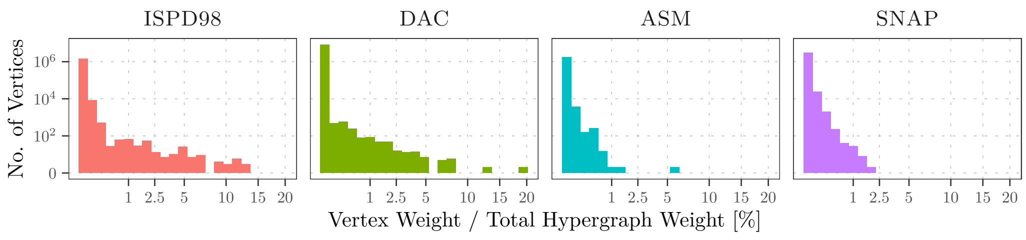


**Figure 2** Vertex weight distributions for each instance type. Each bucket of a histogram shows the number of vertices (y-axis) that contribute $x\%$ to the total weight of the corresponding hypergraph.

(ISPD98 and DAC), we use the area of a circuit element as the weight of its corresponding vertex. We translate sparse matrices (SNAP and ASM instances) to hypergraphs using the row-net model [13] and use the degree of a vertex as its weight. The vertex weight distributions of the individual instance types are depicted in Figure 2.[1]

Additionally, we generate ten ARTIFICIAL instances that use the net structure of the ten largest ISPD98 instances. Instead of using the area as weight, we assign new vertex weights that yield instances for which it is difficult to satisfy the balance constraint: Each vertex is assigned either unit weight or a weight chosen randomly from an uniform distribution in $[1, W] \subseteq \mathbb{N}_+$. Both the probability that a vertex has non-unit weight and the parameter $W$ are determined (depending on the total number of vertices) such that the expected number of vertices with non-unit weight is 120 and the expected total weight of these vertices is half the expected total weight of the resulting hypergraph.

**System and Methodology.** All experiments are performed on a single core of a cluster with Intel Xeon Gold 6230 processors running at 2.1 GHz with 96GB RAM. We compare `KaHyPar-BP-R` and `KaHyPar-BP-K` with the latest recursive bipartitioning (`KaHyPar-R`) and direct $k$-way version (`KaHyPar-K`) of `KaHyPar` [21], the default (`PaToH-D`) and quality preset (`PaToH-Q`) of `PaToH` 3.3 [11], as well as with the recursive bipartitioning (`hMetis-R`) and direct $k$-way version (`hMetis-K`) of `hMetis` 2.0 [29, 30]. Details about the choices of config parameters that influence partitioning quality or imbalance can be found in Appendix D.

We perform experiments using $k \in \{2, 4, 8, 16, 32, 64, 128\}$, $\varepsilon \in \{0.01, 0.03, 0.1\}$, ten repetitions using different seeds for each combination of $k$ and $\varepsilon$, and a time limit of eight hours. We call a combination of a hypergraph $H = (V, E, c, \omega)$, $k$, and $\varepsilon$ an *instance*. Before partitioning an instance, we remove all vertices $v \in V$ from $H$ with a weight greater than $L_k = (1+\varepsilon)\lceil \frac{c(V)}{k} \rceil$ as proposed by Caldwell et al. [10] and adapt $k$ to $k' := k - |V_R|$, where $V_R$ represents the set of removed vertices. We repeat that step recursively until there is no vertex with a weight greater than $L_{k'} := (1+\varepsilon)\lceil \frac{c(V \setminus V_R)}{k'} \rceil$. The input for each partitioner is the subhypergraph $H_{V \setminus V_R}$ of $H$ for which we compute a $k'$-way partition with $L_{k'}^{\mathrm{LPT}}$ as maximum allowed block weight. Note that since all evaluated partitioners internally employ $L_{k'}$ as balance constraint, we initialize each partitioner with a modified imbalance factor $\hat{\varepsilon}$

[1] The benchmark sets and detailed statistics of their properties are publicly available from `http://algo2.iti.kit.edu/heuer/sea21/`.

**Table 1** Percentage of imbalanced instances produced by each partitioner for each combination of instance type and $\varepsilon$.

| | ISPD98 | | | DAC | | | ASM | | | SNAP | | | ARTIFICIAL | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 |
| KaHyPar-BP-K | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| KaHyPar-BP-R | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| KaHyPar-K | 6.3 | 5.6 | 0.8 | 9.5 | 7.9 | 6.3 | 4.1 | 4.1 | 2.0 | 0.9 | 0.9 | 0.0 | 27.1 | 22.9 | 11.4 |
| KaHyPar-R | 10.3 | 8.7 | 7.1 | 19.0 | 19.0 | 14.3 | 6.1 | 4.1 | 4.1 | 6.2 | 2.7 | 0.9 | 28.6 | 24.3 | 12.9 |
| hMetis-K | 43.7 | 22.2 | 9.5 | 33.3 | 22.2 | 11.1 | 67.3 | 32.7 | 4.1 | 33.9 | 20.5 | 3.6 | 51.4 | 38.6 | 24.3 |
| hMetis-R | 17.5 | 15.1 | 7.1 | 20.6 | 15.9 | 12.7 | 8.2 | 6.1 | 4.1 | 15.2 | 10.7 | 4.5 | 58.6 | 54.3 | 34.3 |
| PaToH-Q | 15.9 | 11.1 | 5.6 | 23.8 | 17.5 | 9.5 | 24.5 | 6.1 | 4.1 | 33.9 | 8.0 | 1.8 | 31.4 | 24.3 | 14.3 |
| PaToH-D | 9.5 | 7.9 | 3.2 | 20.6 | 17.5 | 9.5 | 28.6 | 6.1 | 4.1 | 22.3 | 11.6 | 2.7 | 20.0 | 15.7 | 8.6 |

instead of $\varepsilon$ which is calculated as follows:

$$L_{k'} = (1+\hat{\varepsilon})\left\lceil \frac{c(V \setminus V_R)}{k'} \right\rceil = (1+\varepsilon)\mathrm{LPT}(H_{V\setminus V_R}, k') = L_{k'}^{\mathrm{LPT}} \Rightarrow \hat{\varepsilon} = \frac{L_{k'}^{\mathrm{LPT}}}{\lceil \frac{c(V\setminus V_R)}{k'} \rceil} - 1.$$

We consider the resulting $k'$-way partition $\Pi_{k'}$ to be imbalanced, if it is not $\hat{\varepsilon}$-balanced. Each partitioner optimizes the connectivity metric, which we also refer to as the quality of a partition. Partition $\Pi_{k'}$ can be extended to a $k$-way partition $\Pi_k$ by adding each of the removed vertices $v \in V_R$ to $\Pi_k$ as a separate block. Note that adding the removed vertices increases the connectivity metric of a $k'$-way partition only by a constant value $\alpha \geq 0$. Thus, we report the quality of $\Pi_{k'}$, since $(\lambda - 1)(\Pi_k)$ will be always equal to $(\lambda - 1)(\Pi_{k'}) + \alpha$.

For each instance, we average quality and running times using the arithmetic mean (over all seeds). To further average over multiple instances, we use the geometric mean for absolute running times to give each instance a comparable influence. Runs with imbalanced partitions are not excluded from averaged running times. If *all ten runs* of a partitioner produced imbalanced partitions on an instance, we consider the instance as *imbalanced* and mark it with ✗ in the plots.

To compare the solution quality of different algorithms, we use *performance profiles* [17]. Let $\mathcal{A}$ be the set of all algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm $A$, we plot the fraction of instances ($y$-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where $\tau$ is on the $x$-axis. For $\tau = 1$, the $y$-value indicates the percentage of instances for which an algorithm $A \in \mathcal{A}$ performs best. Note that these plots relate the quality of an algorithm to the best solution and thus do not permit a full ranking of three or more algorithms.

**Balanced Partitioning.** In Table 1, we report the percentage of imbalanced instances produced by each partitioner for each instance type and $\varepsilon$. Both KaHyPar-BP-K and KaHyPar-BP-R compute balanced partitions for all tested benchmark sets and parameters. For the remaining partitioners, the number of imbalanced solutions increases as the balance constraint becomes tighter. For the previous KaHyPar versions, the number of imbalanced partitions is most pronounced on VLSI instances: For $\varepsilon = 0.01$, KaHyPar-K and KaHyPar-R compute infeasible solutions for 6.3% (10.3%) of the ISPD98 and for 9.5% (19.0%) of the DAC instances. Comparing the distribution of vertex weights reveals that these instances tend to have a larger proportion of *heavier* vertices compared to the ASM and SNAP instances (see Figure 2. The largest benefit of using our approach can be observed on the artificially generated instances, where KaHyPar-K and KaHyPar-R only computed balanced partitions for 72.9% (71.4%) of the instances (for $\varepsilon = 0.01$).

**Table 2** Percentage of imbalanced instances produced by each partitioner on our RealWorld benchmark set for each combination of $k$ and $\varepsilon$.

| | $k \in \{2,4,8\}$ | | | $k \in \{16,32\}$ | | | $k \in \{64,128\}$ | | | RealWorld | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon$ | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 | 0.01 | 0.03 | 0.1 |
| KaHyPar-BP-K | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| KaHyPar-BP-R | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| KaHyPar-K | 0.7 | 0.0 | 0.0 | 5.0 | 6.0 | 2.0 | 11.1 | 9.1 | 4.0 | 4.9 | 4.3 | 1.7 |
| KaHyPar-R | 2.0 | 0.7 | 0.7 | 11.0 | 9.0 | 7.0 | 21.0 | 18.0 | 13.0 | 10.0 | 8.0 | 6.0 |
| hMetis-K | 12.0 | 2.0 | 0.0 | 53.0 | 21.0 | 11.0 | 76.0 | 57.0 | 14.0 | 42.0 | 23.1 | 7.1 |
| hMetis-R | 2.7 | 2.0 | 0.0 | 18.0 | 14.0 | 7.0 | 34.0 | 27.0 | 17.0 | 16.0 | 12.6 | 6.9 |
| PaToH-Q | 15.3 | 2.7 | 0.7 | 28.0 | 11.0 | 5.0 | 34.0 | 22.0 | 11.0 | 24.3 | 10.6 | 4.9 |
| PaToH-D | 9.3 | 2.7 | 0.7 | 18.0 | 11.0 | 4.0 | 32.0 | 22.0 | 10.0 | 18.3 | 10.6 | 4.3 |

**Table 3** Occurrence of prepacked vertices (i.e., vertices that are fixed to a specific block during partitioning) for each combination of $k$ and $\varepsilon$ when using KaHyPar-BP-R on RealWorld instances: Minimum/average/maximum percentage of prepacked vertices (left), and percentage of instances for which the prepacking is executed at least once (right).

| | $\varepsilon = 0.01$ | | | $\varepsilon = 0.03$ | | | $\varepsilon = 0.1$ | | | Prepacking Triggered | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | Min | Avg | Max | Min | Avg | Max | Min | Avg | Max | $\varepsilon = 0.01$ | $\varepsilon = 0.03$ | $\varepsilon = 0.1$ |
| 2 | - | - | - | - | - | - | - | - | - | - | - | - |
| 4 | - | - | - | - | - | - | - | - | - | - | - | - |
| 8 | $\leq 0.1$ | $\leq 0.1$ | 0.2 | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | - | - | - | 5.0 | 1.7 | - |
| 16 | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | $\leq 0.1$ | 8.3 | 5.0 | 3.3 |
| 32 | $\leq 0.1$ | 8.1 | 59.0 | $\leq 0.1$ | 6.2 | 68.4 | $\leq 0.1$ | 1.9 | 14.7 | 20.0 | 18.3 | 10.0 |
| 64 | $\leq 0.1$ | 23.2 | 87.7 | $\leq 0.1$ | 17.3 | 90.9 | $\leq 0.1$ | 2.7 | 35.9 | 18.3 | 13.3 | 10.0 |
| 128 | $\leq 0.1$ | 67.9 | 100.0 | $\leq 0.1$ | 42.0 | 96.3 | $\leq 0.1$ | 15.4 | 97.0 | 26.7 | 20.0 | 15.0 |

With some notable exceptions, the number of imbalanced partitions of both variants of PaToH and hMetis-R is comparable to that of KaHyPar-R: PaToH computes significantly fewer feasible solutions on sparse matrix instances (ASM and SNAP) for $\varepsilon = 0.01$, while hMetis-R performs considerably worse on the Artificial benchmark set. Out of all partitioners, hMetis-K yields the most imbalanced instances across all benchmark sets. As can be seen in Table 2, the number of imbalanced partitions produced by each competing partitioner increases with deceasing $\varepsilon$ and increasing $k$.

Table 3 shows (i) how often our prepacking algorithm is triggered at least once in KaHyPar-BP-R (see Line 5 in Algorithm 1) and (ii) the percentage of vertices that are treated as fixed vertices (see Table 4 in Appendix E for the results of KaHyPar-BP-K). Except for $k = 128$, on average less than 25% of the vertices are treated as fixed vertices (even less than 10% for $k < 64$), which provides sufficient flexibility to optimize the connectivity objective on the remaining ordinary vertices. However, in a few cases there are also runs where almost all vertices are added to the prepacking. As expected, the triggering frequency and the percentage of fixed vertices increases for larger values of $k$ and smaller $\varepsilon$.

**Quality and Running Times.** Comparing the different KaHyPar configurations in Figure 3 (left), we can see that our new configurations provide the same solution quality as their non-prepacking counterparts. Furthermore, we see that, in general, the direct $k$-way algorithm still performs better than its RB counterpart [38]. Figure 3 (middle) therefore compares the

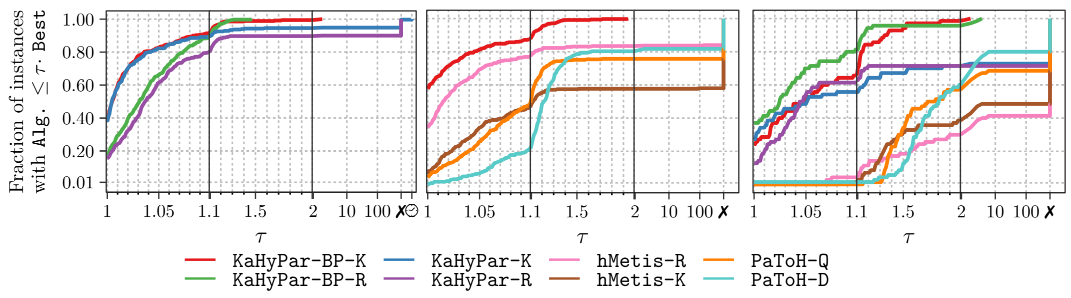


**Figure 3** Performance profiles comparing the solution quality of `KaHyPar-BP-K` and `KaHyPar-BP-R` with `KaHyPar-K` (left), `KaHyPar-R` (left), `PaToH` (middle), and `hMetis` (middle) on our RealWorld benchmark set, and with all systems on our Artificial benchmark set (right) ($\varepsilon = 0.01$).

strongest configuration `KaHyPar-BP-K` with `PaToH` and `hMetis`. We see that `KaHyPar-BP-K` performs considerably better than the competitors. If we compare `KaHyPar-BP-K` with each partitioner individually on the RealWorld benchmark set, `KaHyPar-BP-K` produces partitions with higher quality than those of `KaHyPar-K`, `KaHyPar-BP-R`, `KaHyPar-R`, `hMetis-R`, `hMetis-K`, `PaToH-Q` and `PaToH-D` on 48.9%, 70.2%, 73.2%, 76.4%, 84.3%, 92.9% and 97.9% of the instances, respectively. `KaHyPar-BP-K` outperforms `KaHyPar-BP-R` on the RealWorld benchmark set. On artificial instances, both algorithms produce partitions with comparable quality for $\varepsilon = \{0.01, 0.03\}$, while the results are less clear for $\varepsilon = 0.1$ (see Figure 3 (right), as well as Figures 4 in Appendix F).

The running time plots (see Figure 5 and 6 in Appendix G) show that our new approach does not impose any additional overheads in `KaHyPar`. On average, `KaHyPar-BP-K` is slightly faster than `KaHyPar-K` as our new algorithm has replaced the previous balancing strategy in `KaHyPar` (restarting the bipartition with an tighter bound on the weight of the heaviest block if the bipartition is imbalanced). The running time difference is less pronounced for `KaHyPar-BP-R` and `KaHyPar-R`. This can be explained by the fact that, in `KaHyPar-BP-R`, our prepacking algorithm is executed on the input hypergraph, whereas it is executed on the coarsest hypergraph in `KaHyPar-BP-K`.

## 7 Conclusion and Future Work

In this work, we revisited the problem of computing balanced partitions for weighted hypergraphs in the multilevel setting and showed that many state-of-the-art hypergraph partitioners struggle to find balanced solutions on hypergraphs with weighted vertices – especially for tight balance constraints. We therefore developed an algorithm that enables partitioners based on the recursive bipartitioning scheme to reliably compute balanced partitions. The method is based on the concept of *deeply balanced* bipartitions and is implemented by pre-assigning a small subset of the heaviest vertices to the two blocks of each bipartiton. For this pre-assignment, we established a property that can be verified in polynomial time and, if fulfilled, leads to provable balance guarantees for the resulting $k$-way partition. We integrated the approach into the recursive bipartitioning algorithm of `KaHyPar`. Our new algorithms `KaHyPar-BP-K` and `KaHyPar-BP-R` are capable of computing balanced solutions on all instances of a diverse benchmark set, without negatively affecting the solution quality or running time of `KaHyPar`.

Interesting opportunities for future research include replacing the LPT algorithm with an algorithm that additionally optimizes the partitioning objective to construct sufficiently balanced prepackings with improved solution quality [34], and integrating rebalancing strategies similar to the techniques proposed for non-multilevel partitioners [9, 10, 18] into multilevel refinement algorithms.

## References

1 Y. Akhremtsev, T. Heuer, P. Sanders, and S. Schlag. Engineering a Direct $k$-way Hypergraph Partitioning Algorithm. In *19th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 28–42. SIAM, January 2017.

2 C. J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*, pages 80–85, April 1998.

3 C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration: The VLSI Journal*, 19(1-2):1–81, 1995.

4 C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-Level Direct $k$-Way Hypergraph Partitioning with Multiple Constraints and Fixed Vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.

5 D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, February 2013.

6 M. A. Bender and M. Farach-Colton. The LCA Problem Revisited. In *Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer, 2000.

7 R. H. Bisseling, B. O. Auer Fagginger, A. N. Yzelman, T. van Leeuwen, and Ü. V. Çatalyürek. Two-Dimensional Approaches to Sparse Matrix Partitioning. *Combinatorial Scientific Computing*, pages 321–349, 2012.

8 T. N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, May 1992.

9 A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 661–666, 2000.

10 A. E. Caldwell, A. B. Kahng, and I. L. Markov. Iterative Partitioning with Varying Node Weights. *VLSI Design*, 11(3):249–258, 2000.

11 Ü. V Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 75–86. Springer, 1996.

12 Ü. V. Çatalyürek and C. Aykanat. PaToH: Partitioning Tool for Hypergraphs. `https://www.cc.gatech.edu/~umit/PaToH/manual.pdf`, 2011.

13 Ü. V. Çatalyürek 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.

14 Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar. UMPa: A Multi-Objective, Multi-Level Partitioner for Communication Minimization. In *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*, pages 53–66, February 2012.

15 T. Davis, I. S. Duff, and S. Nakov. Design and Implementation of a Parallel Markowitz Threshold Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 41(2):573–590, April 2020.

16 K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek. Parallel Hypergraph Partitioning for Scientific Computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

17 E. D. Dolan and J. J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002.

**18** S. Dutt and H. Theny. Partitioning Around Roadblocks: Tackling Constraints with Intermediate Relaxations. In *International Conference on Computer-Aided Design (ICCAD)*, pages 350–355, November 1997.

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

**20** M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, volume 174. W.H. Freeman, San Francisco, 1979.

**21** Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. In *18th International Symposium on Experimental Algorithms (SEA 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

**22** R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

**23** R. L. Graham, E. L. Lawler, J. K. Lenstra, and R. Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. Elsevier, 1979.

**24** S. A. Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, 1995.

**25** T. Heuer. Engineering Initial Partitioning Algorithms for direct $k$-way Hypergraph Partitioning. Bachelor thesis, Karlsruhe Institute of Technology, August 2015.

**26** T. Heuer, P. Sanders, and S. Schlag. Network Flow-Based Refinement for Multilevel Hypergraph Partitioning. *ACM Journal of Experimental Algorithmics (JEA)*, 24(1):2.3:1–2.3:36, September 2019.

**27** T. Heuer and S. Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017.

**28** G. Karypis. A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 5.1.0. Technical report, University of Minnesota, 2013.

**29** G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Application in VLSI Domain. In *34th Conference on Design Automation (DAC)*, pages 526–529, June 1997.

**30** G. Karypis and V. Kumar. Multilevel $k$-way Hypergraph Partitioning. *VLSI Design*, 11(3):285–300, 2000.

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

**32** T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, Inc., 1990.

**33** J. Leskovec and A. Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, 2014.

**34** N. Maas. Multilevel Hypergraph Partitioning with Vertex Weights Revisited. Bachelor thesis, Karlsruhe Institute of Technology, May 2020.

**35** D. A. Papa and I. L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. Citeseer, 2007.

**36** M. Pinedo. *Scheduling*, volume 29. Springer, 2012.

**37** S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. $k$-way Hypergraph Partitioning via $n$-Level Recursive Bisection. In *18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 53–67. SIAM, January 2016.

**38** Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2020.

**39** C. Schulz. *High Quality Graph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

40 H. Shin and C. Kim. A Simple Yet Effective Technique for Partitioning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(3):380–386, 1993.
41 B. Vastenhouw and R. H. Bisseling. A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication. *SIAM Review*, 47(1):67–95, 2005.
42 N. Viswanathan, C. J. Alpert, C. C. N. Sze, Z. Li, and Y. Wei. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*, pages 774–782. ACM, June 2012.

## A Proof of Lemma 3

▶ **Lemma 3.** (LPT Bound). *Let $H = (V, E, c, \omega)$ be a weighted hypergraph, $\Psi_k$ be a $k$-way prepacking for a set of fixed vertices $P \subseteq V$, and let $O := \langle v_1, \dots, v_m \mid v_i \in V \setminus P\rangle$ be the sequence of all ordinary vertices of $V \setminus P$ sorted in decreasing order of weight. If we assign the remaining vertices $O$ to the blocks of $\Psi_k$ by using the* LPT *algorithm, we can extend $\Psi_k$ to a $k$-way partition $\Pi_k$ of $H$ such that the weight of the heaviest block is bound by:*

$$\max(\Pi_k) \leq \max\{\frac{1}{k}c(P) + h_k(O), \max(\Psi_k)\}, \text{ with } h_k(O) := \max_{i\in\{1,\dots,m\}} c(v_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(v_j).$$

**Proof.** We define $\Psi_k := \{P_1, \dots, P_k\}$ and $\Pi_k := \{V_1, \dots, V_k\}$. Let assume that the LPT algorithm assigned the $i$-th vertex $v_i$ of $O$ to block $V_j \in \Pi_k$. We define $V_j^{(i)}$ as a subset of block $V_j$ that only contains vertices of $\langle v_1, \dots, v_i\rangle \subseteq O$ and $P$. Since the LPT algorithm always assigns an vertex to a block with the smallest weight (see Section 3), the weight of $V_j^{(i-1)}$ must be smaller or equal to $\frac{1}{k}(c(P) + \sum_{j=1}^{i-1} c(v_j))$ (average weight of all previously assigned vertices), otherwise $V_j^{(i-1)}$ would be not the block with the smallest weight.

$$\Rightarrow c(V_j^{(i)}) = c(V_j^{(i-1)}) + c(v_i) \leq \frac{1}{k}(c(P) + \sum_{j=1}^{i-1} c(v_j)) + c(v_i) \leq \frac{1}{k}c(P) + h_k(O)$$

We can establish an upper bound on the weight of all blocks to which the LPT algorithm assigns an vertex to with $\frac{1}{k}c(P) + h_k(O)$. If the LPT algorithm does not assign any vertex to a block $V_j \in \Pi_k$, its weight is equal to $c(P_j) \leq \max(\Psi_k)$.

$$\Rightarrow \max(\Pi_k) \leq \max\{\frac{1}{k}c(P) + h_k(O), \max(\Psi_k)\}$$ ◀

## B Generalized Balance Property

▶ **Definition 7.** (Generalized Balance Property). *Let $H = (V, E, c, \omega)$ be a hypergraph for which we want to compute an $\varepsilon$-balanced $k$-way partition and $\Psi := \{P_1, P_2\}$ be a prepacking of $H$ for a set of fixed vertices $P \subseteq V$. Furthermore, let $O_{t_1}$ resp. $O_{t_2}$ be the sequence of the $t_1$ resp. $t_2$ heaviest ordinary vertices of $V \setminus P$ sorted in decreasing vertex weight order such that $t_1$ resp. $t_2$ is the smallest number that satisfies $c(P_1) + c(O_{t_1}) \geq L_2$ resp. $c(P_2) + c(O_{t_2}) \geq L_2$ (see Line 2, Algorithm 1). We say that a prepacking $\Psi$ satisfies the balance property with respect to $k$ if the following conditions hold:*

**(i)** *$\Psi$ is deeply balanced*
**(ii)** $\frac{1}{k_1}c(P_1) + h_{k_1}(O_{t_1}) \leq L_k$ *with* $k_1 := \lceil \frac{k}{2} \rceil$
**(iii)** $\frac{1}{k_2}c(P_2) + h_{k_2}(O_{t_2}) \leq L_k$ *with* $k_2 := \lfloor \frac{k}{2} \rfloor$

The proof of Theorem 5 can be adapted such that we show that there exist a $k_1$- resp. $k_2$-way partition $\Pi_{k_1}$ resp. $\Pi_{k_2}$ for $V_1$ resp. $V_2$ of any balanced bipartition $\Pi_2 := \{V_1, V_2\}$ that respects the prepacking $\Psi$ with $\max(\Pi_{k_1}) \leq \frac{1}{k_1}c(P_1) + h_{k_1}(O_{t_1}) \leq L_k$ (Defintion (ii)) and $\max(\Pi_{k_2}) \leq \frac{1}{k_2}c(P_2) + h_{k_2}(O_{t_2}) \leq L_k$ (Defintion (iii)).

## C Proof of Claim 6

▶ **Lemma 8.** *Let $L = \langle a_1, \dots, a_n\rangle$ be a sequence of elements sorted in decreasing weight order with respect to a weight function $c : L \to \mathbb{R}_{\geq 0}$ (for a subsequence $A := \langle a_1, \dots, a_l\rangle$ of $L$, we define $c(A) := \sum_{i=1}^{l} c(a_i)$), $L'$ be an abitrary subsequence of $L$ sorted in decreasing weight order and $L_m = \langle a_1, \dots, a_m\rangle$ the subsequence of the $m \leq n$ heaviest elements in $L$. Then the following conditions hold:*

**(i)** *If $c(L') \leq c(L_m)$, then $h_k(L') \leq h_k(L_m)$*

**(ii)** *If $c(L') > c(L_m)$, then $h_k(L') - \frac{1}{k}c(L') \leq h_k(L_m) - \frac{1}{k}c(L_m)$*

**Proof.** For convenience, we define $L' := \langle b_1, \dots, b_l\rangle$. Note that $\forall i \in \{1, \dots, \min(m, l)\}$ : $c(a_i) \geq c(b_i)$, since $L_m$ contains the $m$ heaviest elements in decreasing order. We define $i := \arg\max_{i \in \{1,\dots,l\}} c(b_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(b_j)$ (index that maximizes $h_k(L')$).

(i) + (ii): If $i \leq m$, then

$$h_k(L') = c(b_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(b_j) \overset{\forall j \in [1,i]:\ c(b_j) \leq c(a_j)}{\leq} c(a_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(a_j) \leq h_k(L_m)$$

(i): If $m < i \leq l$, then

$$h_k(L') = c(b_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(b_j) = c(b_i) - \frac{1}{k}\sum_{j=i}^{n} c(b_j) + \frac{1}{k}c(L') \leq \left(1 - \frac{1}{k}\right)c(b_i) + \frac{1}{k}c(L')$$
$$\overset{\substack{c(b_i) \leq c(a_m)\\ c(L') \leq c(L_m)}}{\leq} \left(1 - \frac{1}{k}\right)c(a_m) + \frac{1}{k}c(L_m) = c(a_m) + \frac{1}{k}\sum_{j=1}^{m-1} c(a_j) \leq h_k(L_m)$$

(ii): If $m < i \leq l$, then

$$h_k(L') - \frac{1}{k}c(L') = c(b_i) + \frac{1}{k}\sum_{j=1}^{i-1} c(b_j) - \frac{1}{k}c(L') = c(b_i) - \frac{1}{k}\sum_{l=i}^{n} c(b_l) \leq \left(1 - \frac{1}{k}\right)c(b_i)$$
$$\overset{c(b_i) \leq c(a_m)}{\leq} \left(1 - \frac{1}{k}\right)c(a_m) = c(a_m) + \frac{1}{k}\sum_{j=1}^{m-1} c(a_j) - \frac{1}{k}c(L_m) \leq h_k(L_m) - \frac{1}{k}c(L_m)$$ ◀

▷ Claim 6. It holds that: $\frac{1}{k'}c(P_1) + h_{k'}(O_1) \leq \frac{1}{k'}\max(\Psi) + h_{k'}(O_t)$.

Proof. Remember, $\Psi = \{P_1, P_2\}$, $\Pi_2 = \{V_1, V_2\}$ with $P_1 \subseteq V_1$ and $P_2 \subseteq V_2$, $O_1$ is equal to $V_1 \backslash P_1$ and $O_t$ represents the $t$ heaviest vertices of $(V_1 \cup V_2) \backslash (P_1 \cup P_2)$ with $\max(\Psi) + c(O_t) \geq L_2$ as defined in Definition 4. The following proof distingush two cases based on Lemma 8.

If $c(O_1) \leq c(O_t)$, then

$$\frac{1}{k'}c(P_1) + h_{k'}(O_1) \overset{\text{Lemma 8(i)}}{\leq} \frac{1}{k'}c(P_1) + h_{k'}(O_t) \overset{c(P_1) \leq \max(\Psi)}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t)$$

If $c(O_1) > c(O_t)$, then

$$\begin{aligned}
\frac{1}{k'}c(P_1) + h_{k'}(O_1) &= \frac{1}{k'}c(P_1) + h_{k'}(O_1) - \frac{1}{k'}c(O_1) + \frac{1}{k'}c(O_1) \\
&\overset{\text{Lemma 8(ii)}}{\leq} \frac{1}{k'}(c(P_1) + c(O_1)) + h_{k'}(O_t) - \frac{1}{k'}c(O_t) \\
&\overset{c(P_1)+c(O_1)=c(V_1)}{=} \frac{1}{k'}(c(V_1) - c(O_t)) + h_{k'}(O_t) \\
\overset{c(V_1)\leq L_2}{\leq} \frac{1}{k'}(L_2 - c(O_t)) + h_{k'}(O_t) &\overset{\max(\Psi)+c(O_t)\geq L_2}{\leq} \frac{1}{k'}\max(\Psi) + h_{k'}(O_t) \qquad \triangleleft
\end{aligned}$$

## D Configuration of Evaluated Partitioners

hMetis does not directly optimize the $(\lambda - 1)$-metric. Instead it optimizes the *sum-of-external-degrees* (SOED), which is closely related to the connectivity metric: $(\lambda - 1)(\Pi) = \text{SOED}(\Pi) - \text{cut}(\Pi)$. We therefore configure hMetis to optimize SOED and calculate the $(\lambda - 1)$-metric accordingly. The same approach is also used by the authors of hMetis [30]. Additionally, hMetis-R defines the maximum allowed imbalance of a partition differently [28]. For example, an imbalance value of 5 means that a block weight between $0.45 \cdot c(V)$ and $0.55 \cdot c(V)$ is allowed at each bisection step. We therefore translate the imbalance parameter $\varepsilon$ to a modified parameter $\varepsilon'$ such that the correct allowed block weight is matched after $\log_2(k)$ bisections:

$$\varepsilon' := 100 \cdot \left( \left( (1+\varepsilon)\frac{\lceil \frac{c(V)}{k} \rceil}{c(V)} \right)^{\frac{1}{\log_2(k)}} - 0.5 \right)$$

PaToH is evaluated with both the default (PaToH-D) and the quality preset (PaToH-Q). However, there are also more fine-grained parameters available for PaToH as described in [12]. In our case, the balance parameter is of special interest as it might affect the ability of PaToH to find a balanced partition. Therefore, we evaluated the performance of PaToH on our benchmark set with each of the possible options Strict, Adaptive and Relaxed. The configuration using the Strict option (which is also the default) consistently produced fewest imbalanced partitions and had similar quality to the other configurations. Consequently, we only report the results of this configuration.

## E Prepacking Algorithm Statistics for `KaHyPar-BP-K`

**Table 4** Occurrence of prepacked vertices (i.e., vertices that are fixed to a specific block during partitioning) for each combination of $k$ and $\varepsilon$ when using `KaHyPar-BP-K` on RealWorld instances: Minimum/average/maximum percentage of prepacked vertices (left), and percentage of instances for which the prepacking is executed at least once (right).

| | $\varepsilon = 0.01$ | | | $\varepsilon = 0.03$ | | | $\varepsilon = 0.1$ | | | Prepacking Triggered [%] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $k$ | Min | Avg | Max | Min | Avg | Max | Min | Avg | Max | $\varepsilon = 0.01$ | $\varepsilon = 0.03$ | $\varepsilon = 0.1$ |
| 2 | - | - | - | - | - | - | - | - | - | - | - | - |
| 4 | - | - | - | - | - | - | - | - | - | - | - | - |
| 8 | 6.7 | 17.1 | 41.6 | 0.4 | 0.5 | 0.6 | 2.3 | 2.3 | 2.3 | 5.0 | 3.3 | 1.7 |
| 16 | 3.1 | 15.6 | 34.0 | 0.2 | 2.0 | 7.2 | 1.9 | 2.1 | 2.3 | 8.3 | 6.7 | 3.3 |
| 32 | 0.3 | 29.9 | 56.0 | 0.1 | 11.7 | 42.3 | 0.2 | 3.4 | 26.3 | 13.3 | 15.0 | 6.7 |
| 64 | 0.2 | 54.4 | 94.3 | 0.3 | 23.0 | 69.3 | 0.4 | 6.6 | 94.7 | 21.7 | 10.0 | 8.3 |
| 128 | 0.5 | 76.5 | 100.0 | 0.4 | 42.4 | 91.0 | 0.3 | 15.7 | 59.8 | 28.3 | 21.7 | 11.7 |

## F Quality Comparison for $\varepsilon = 0.03$ and $\varepsilon = 0.1$

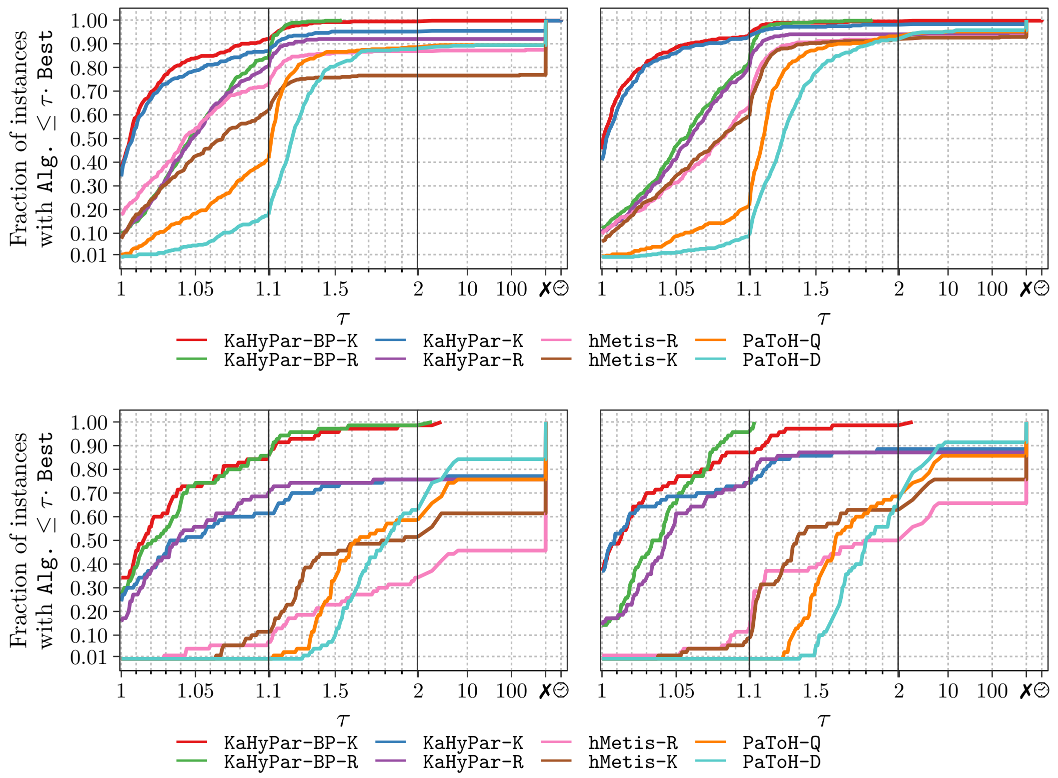


**Figure 4** Comparing the solution quality of each evaluated partitioner for $\varepsilon = 0.03$ (left) and $\varepsilon = 0.1$ (right) on our RealWorld (top) and Artificial (bottom) benchmark set. Note, ⏲ marks instances that exceeded the time limit.

## G Absolute Running Times

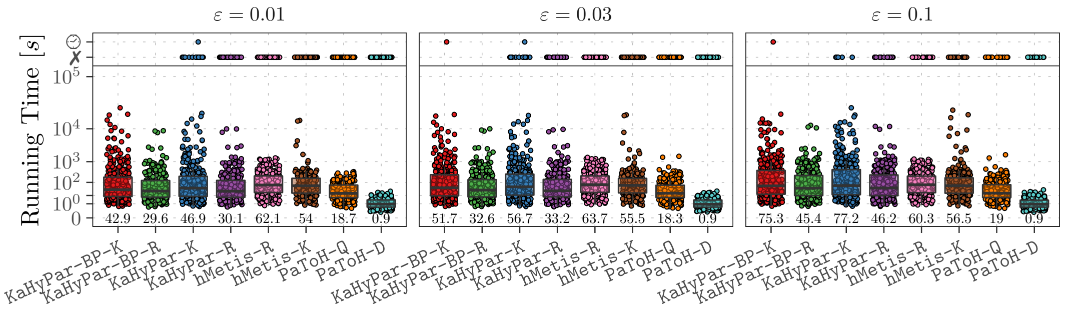


**Figure 5** Comparing the running time of each evaluated partitioner for different values of $\varepsilon$ on our RealWorld benchmark set. The number under each boxplot denotes the average running time of the corresponding partitioner. Note, ⏲ marks instances that exceeded the time limit.

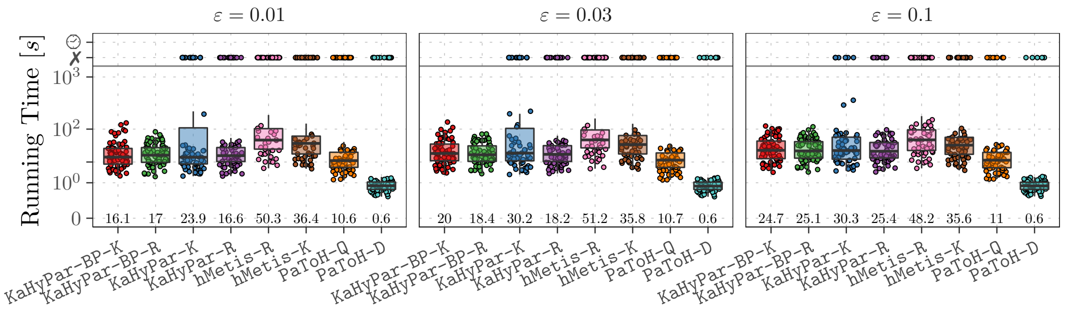


**Figure 6** Comparing the running time of each evaluated partitioner for different values of $\varepsilon$ on our Artificial benchmark set. The number under each boxplot denotes the average running time of the corresponding partitioner. Note, ⏲ marks instances that exceeded the time limit.

# On Tamaki's Algorithm to Compute Treewidths

**Ernst Althaus**
Johannes Gutenberg-Universität Mainz, Germany

**Daniela Schnurbusch**
Johannes Gutenberg-Universität Mainz, Germany

**Julian Wüschner**
Johannes Gutenberg-Universität Mainz, Germany

**Sarah Ziegler**
Johannes Gutenberg-Universität Mainz, Germany

**Abstract**

We revisit the exact algorithm to compute the treewidth of a graph of Tamaki and present it in a way that facilitates improvements. The so-called $I$-blocks and $O$-blocks enumerated by the algorithm are interpreted as subtrees of a tree-decomposition that is constructed. This simplifies the proof of correctness and allows to discard subtrees from the enumeration by some simple observations. In our experiments, we show that one of these modifications in particular reduces the number of enumerated objects considerably.

**2012 ACM Subject Classification** Theory of computation → Fixed parameter tractability

**Keywords and phrases** Tree Decomposition, Exact Algorithm, Algorithms Engineering

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.9

**Supplementary Material** *Software*: `https://gitlab.rlp.net/daschnur/compute_treewidths`
archived at `swh:1:dir:083d68c2152e2521bcd065be491c7a6b35267cc5`

## 1 Introduction

Tree decompositions are a major tool for obtaining parameterized algorithms for hard graph problems as many problems allow parameterization with respect to the treewidth of the graph (see, e.g., [11]). In order to use such algorithms, the first step is to compute a tree decomposition with minimal width or an approximation thereof.

It is well known that computing an optimal tree decomposition is $\mathcal{NP}$-hard [3]. Owing to the early results of Robertson and Seymour, we know that an optimal tree decomposition can be computed in $\mathcal{O}(n^2)$ if the treewidth is bounded, but the proof is non-constructive. In [5], Bodlaender presented a linear-time algorithm for this problem, effectively settling the matter from a theoretical point of view. Since the description of the algorithm is hardly accessible, we recently published a simpler description of it [1, 2].

The main problem of Bodlaender's algorithm is its running-time dependency on the treewidth. More precisely, for a graph with treewidth $tw$, the best running time estimate is $2^{\mathcal{O}(tw^3)} \cdot n$, which makes the dependence on the treewidth in the computation of the tree decomposition a major theoretical bottleneck of the computation. Notice that algorithms using tree decompositions do not necessarily require the latter to be optimal for being a parameterized algorithm, as long as their width is bounded in the optimal treewidth. For this reason, efforts to compute *approximately* optimal tree decompositions gained considerable attention in the recent past, leading, among other things, to the discovery of the first such algorithm with linear time dependency on the size of the graph and a single exponential dependency on the treewidth [6].

From an applied perspective, many heuristics were designed to compute tree decompositions, including the min-fill and min-degree heuristics, see, e.g., [15, 12]. Nevertheless, there remains a large interest in practically feasible algorithms for the construction of optimal tree decompositions, as evidenced, e.g., by recent iterations of the PACE competition (`https://pacechallenge.org`). Currently, the algorithm of Tamaki [15, 14] (`https://github.com/TCS-Meiji/PACE2017-TrackA`) seems to be one of the fastest algorithms.

One key ingredient for practically efficient algorithms is a suitable preprocessing of the (input) graph. Studies have shown that the notion of so-called safe separators is very beneficial for tree decompositions[7]. Safe separators are subsets of the nodes whose removal separates the graph into several parts. An optimal tree decomposition can then be found by considering the components (for each component, including the separator) individually. Bodlaender et al. [7] showed some sufficient conditions for separators to be safe. Furthermore, the authors showed that separators that are an almost clique are safe and presented an algorithm find them. Tamaki [14] gives a heuristic to compute safe separators. The author first computes separators by using a simple heuristic algorithm to compute a tree decomposition and uses the intersections of neighboring bags as candidate separators. For each candidate, he uses an additional heuristic to test another sufficient condition for safe separators.

Computing lower bounds is a subroutine in many exact algorithms for $\mathcal{NP}$-hard problems. An overview of lower bounds for the treewidth is given in [9]. A simple lower bound is obtained by heuristically computing a minor of the given graph and using the second smallest degree as lower bound (if the smallest degree appears at least twice, this degree is to be taken).

In this paper we describe the algorithm of Tamaki in a way that can be interpreted more intuitively. In order to construct an optimal tree decomposition, we assume the tree decomposition to be rooted at a defined vertex. Then we build the tree decomposition from the leaves and call the constructed structures *partial tree decompositions* (a formal definition is given in Section 3). The key insight of the algorithm of Tamaki is that there are only a linear number of possible bags of the root of a partial tree decomposition, knowing the partial tree decompositions for the children. The same holds for the leaves.

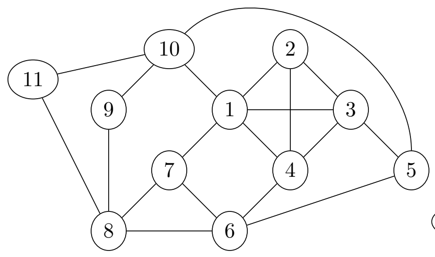
The interpretation of the enumerated partial tree decompositions allows to remove some of them by simple observations that exclude partial tree decompositions from being extended to an optimal tree decomposition or being replaced by partial tree decompositions already enumerated. Some of these observations are automatically guaranteed in the original description of the Tamaki's algorithm. Furthermore, we improve the usage of safe separators by testing the sufficient conditions on all separators constructed when enumerating partial tree decompositions.

The paper is organized as follows: In Section 2, we introduce the terminology, basic definitions, and known properties. The basic algorithm is stated in Section 3, followed by a section on the methods to eliminate partial tree decompositions. Before we show some experimental results in Section 6, we present some additional details on the algorithm in Section 5. Finally, we give a short conclusion.

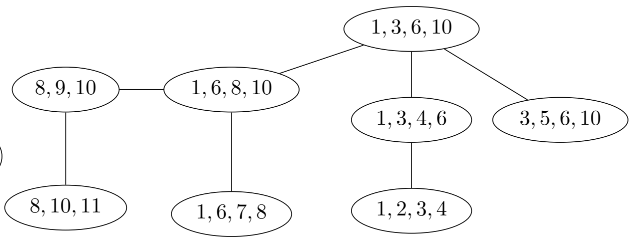
## 2 Definitions and Basic Properties

Unless stated otherwise, we assume all graphs to be undirected, finite, and without self-loops. Let $G$ be a graph. We denote its vertex and edge sets as $V(G)$ and $E(G)$, respectively. Additionally, we define

- $N_G(v) := \{u \in V(G) | (u,v) \in E(G)\}$, the open neighborhood of a vertex $v$ in $G$,
- $N_G[v] := N_G(v) \cup \{v\}$, the closed neighborhood of $v$ in $G$,
- $deg(v) := |N_G(v)|$, the degree of $v$ in $G$,
- $N_G(U) := \bigcup_{u \in U} N_G(u) \setminus U$, the open neighborhood of the vertex set $U \subseteq V(G)$,
- $N_G[U] := N_G(U) \cup U$, the closed neighborhood of $U \subseteq V(G)$.



**(a)** An example graph.



**(b)** Corresponding tree decomposition.

**Figure 1** Example for a tree decomposition of width 3 from a given graph. In this case, the tree decomposition is optimal and thus also has a treewidth of 3.

The subscript $G$ can be omitted in cases where there is no ambiguity about the graph.

For $U \subseteq V(G)$, $G[U]$ denotes the subgraph of $G$ induced by $U$, i.e., the graph with vertex set $V(G[U]) = U$ and edge set $E(G[U]) = \{(a,b) \in E(G) | a,b \in U\}$. $G\langle U\rangle$ denotes the graph obtained from $G$ by completing $U$ to a clique.

Let $C, S \subseteq V(G)$ be two vertex sets. We define:

- $C$ is *connected* in $G$ if there exists a path between all pairs of vertices $u, v \in C$,
- $C$ is a *connected component* of $G$ if $C$ is connected and inclusion-wise maximal with this property.
- $C$ is a *component associated with* $S$ if $C$ is a connected component in $G[V(G) \setminus S]$
- $S$ is a *separator* of $G$ if there exist at least two components associated with $S$.
- $S$ is a *$u,v$-separator* for vertices $u, v \in V(G)$ if $u$ and $v$ lie in different components associated with $S$.
- $C$ is a *full component* associated with $S$ if $N(C) = S$.
- $S$ is a *minimal separator* if there exist at least two full components associated with $S$.

One can show that a minimal separator is minimal in the sense that, if we remove a vertex from it, it no longer separates vertices that lie in different full components. It may, however, still separate vertices in a previously non-full component from the rest of the graph.

▶ **Definition 1** (Tree Decomposition, [13]). *For a graph $G$, let $T$ be a tree and $(X_t)_{t \in V(T)}$ a family of vertex sets indexed by the nodes of $T$ with $X_t \subseteq V(G)$ for all $t \in V(G)$. $\mathcal{T} := (T, (X_t)_{t \in V(T)})$ is called a tree decomposition of $G$ if*

1. $\bigcup_{t \in V(T)} X_t = V$,
2. *for each edge $e = (u,v) \in E(G)$, there exists a node $t \in V(T)$ with $u, v \in X_t$, and*
3. *for all $t_1, t_2, t_3 \in V(T)$, such that node $t_2$ lies on the path between nodes $t_1$ and $t_3$, $X_{t_1} \cap X_{t_3} \subseteq X_{t_2}$.*

We will refer to the third condition as the *consistency property* throughout this paper. The vertex sets associated with nodes of a tree decomposition are called *bags.* We say that two bags are adjacent if the corresponding nodes are adjacent. For an example, refer Figure 1.

▶ **Definition 2.** *The width of a tree decomposition $(T, (X_t)_{t \in V(T)})$ is $max_{t \in V(T)} |X_t| - 1$.*

▶ **Definition 3.** *The treewidth $tw(G)$ of a graph $G$ is the minimum width of all tree decompositions of $G$.*

The treewidth of a graph $G$ can alternatively be defined in terms of its triangulations. A *chord* of a cycle $C$ in $G$ is an edge whose endpoints lie in the vertex set of $C$ but that is not in the edge set of $C$ [16]. It is easy to see that all cycles of length 3 (i.e., triangles) have no chords. $G$ is *chordal* (or *triangulated*) if every cycle of length at least 4 has a chord. A graph $H = (V(G), E')$ is a *triangulation* of $G$ if $H$ is chordal and $G$ is a subgraph of $H$, i.e., $E' \supseteq E(G)$. A triangulation is *minimal* if the edge set is inclusion-wise minimal with respect to these conditions. In [10], Bouchitté and Todinca show that there is a minimal triangulation such that a tree decomposition of minimal width can be obtained by constructing a bag for each maximal clique of the triangulation. These bags can be connected to a tree satisfying consistency and the intersections between neighboring bags are minimal separators.

▶ **Definition 4.** *A potential maximal clique (pmc) of a graph $G$ is a vertex set that induces a maximal clique in some minimal triangulation of $G$.*

This definition, however, is difficult to work with algorithmically. In [10], Bouchitté and Todinca show that potential maximal cliques can also be characterized by local features:

▶ **Definition 5.** *A vertex set $K \subset V(G)$ is cliquish if for every pair of distinct vertices $u, v \in K$ there exists a path from $u$ to $v$ that does not lead through other vertices of $K$.*

▶ **Lemma 6** ([10], Theorem 3.15)**.** *A vertex set $K \subset V(G)$ is a potential maximal clique if and only if $K$ is cliquish and has no full components associated with it.*

▶ **Definition 7.** *A canonical tree decomposition of a graph $G$ is a tree decomposition of $G$, where each bag is a potential maximal clique of $G$ and for every two adjacent bags $X$ and $Y$, $X \cap Y$ is a minimal separator.*

The follwoing two lemmas follow directly from the theory of Bouchitté and Todinca [10] and are also used by Tamaki [14]. The first one summarizes the discussion above.

▶ **Lemma 8.** *For each graph $G$ there is a canonical tree decomposition of width $tw(G)$.*

▶ **Lemma 9.** *Let $\mathcal{T} = (T, (X_i)_{i \in V(T)})$ be a canonical tree decomposition and let $\mathcal{T}' = (T', (Y_j)_{j \in V(T')})$ be any tree decomposition of a graph $G$, such that for every $Y_j$ there exists $X_i$ with $Y_j \subseteq X_i$. Then for every $X_i$ there exists $Y_s$ with $X_i = Y_j$.*

In other words, if all bags of a tree decomposition are subsets of bags of a canonical tree decomposition, then all bags in the canonical tree decomposition are also part of the other tree decomposition.

We informally argue that Lemma 9 holds as follows: Transforming each bag of a tree decomposition into a clique will result in a triangulation. If the lemma would not be satisfied, the triangulation of the tree decomposition $\mathcal{T}'$ is a strict subset of the triangulation of $\mathcal{T}$.

## 3 The Algorithm

The algorithm presented by Tamaki [14] decides whether a graph $G$ has a treewidth of at most $k$, for $k$ ranging from a lower bound on $tw(G)$ to $tw(G)$ (which is, of course, unknown in advance). This is done by trying to construct a canonical tree decomposition of $G$ with width $k$. The construction is performed by determining all possible candidates for leaves of such a tree decomposition and then iteratively forming larger tree structures by combining two of them along with applying the canonization rules resulting from Lemma 15. This procedure succeeds only in the case of $k = tw(G)$, yielding a canonical tree decomposition of $G$ with optimal treewidth. In the following, it is easier to assume $G$ to be connected, which is guaranteed after the preprocessing techniques discussed in Section 5 (tree decompositions can be computed for each connected component separately).

## 3.1 Theoretical Foundations of the Algorithm

▶ **Definition 10.** *Let $G$ be a graph and $\mathcal{T} = (T, (X_t)_{t \in V(T)})$ be a rooted canonical tree decomposition of $G$. For a bag $Y_t$ with $t \in V(T)$, let $T_{Y_t}$ be the subtree of $T$ with root $t$ and all descendants. $\mathcal{T}_Y = (T_Y, (X_t)_{t \in V(T_Y)})$ is called the partial tree decomposition of $\mathcal{T}$ rooted at $Y$.*

We abbreviate the term *partial tree decomposition* to *ptd*. The algorithm will construct subtrees of tree decompositions, but we will not show that these are necessarily canonical.

▶ **Definition 11.** *For a ptd $\mathcal{T} = (T, (X_t)_{t \in V(T)})$ with root bag $X_r$ we define*

- *$bag(\mathcal{T}) = X_r$,*
- *$V(\mathcal{T}) = \bigcup_{t \in V(T)} X_t$,*
- *$outlet(\mathcal{T}) := N(V(G) \setminus V(\mathcal{T}))$,*
- *$inlet(\mathcal{T}) := V(\mathcal{T}) \setminus outlet(\mathcal{T})$,*
- *a child of $\mathcal{T}$ as the ptd induced by a child node of $\mathcal{T}$ and its descendants, and*
- *$children(\mathcal{T})$ as the set of all children of $\mathcal{T}$.*

Note that $outlet(\mathcal{T}) \subseteq bag(\mathcal{T})$. This is easy to see: the vertices in $outlet(\mathcal{T})$ are exactly those vertices having neighbors that do not lie in $V(\mathcal{T})$. Hence, the edge between such a vertex and its neighbor has to be covered by a bag outside of $\mathcal{T}$. By the consistency property of tree decompositions, however, all bags containing a given vertex have to be connected, implying that all vertices in $outlet(\mathcal{T})$ are also in $bag(\mathcal{T})$.

The second type of structure created by the algorithm is called a *partial tree decomposition with unfinished root* (or *ptdur*). Ptdurs generalize ptds and their role is to gather several ptds under one root. We will later see how they can be extended into ptds using the canonization rules obtained from Lemma 15.

▶ **Definition 12.** *Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be ptds. A partial tree decomposition with unfinished root is a rooted tree with its root labeled $\bigcup_{i \in I} outlet(\mathcal{T}_i)$ and the $\mathcal{T}_i$ as its children.*

We extend the notation from Definition 11 to ptdurs. Note that we do not require the root of a ptdur to be a potential maximal clique, so ptdurs are not necessarily pdts as well. So far, ptdurs also do not necessarily satisfy the consistency property. We fix the latter by immediately discarding any ptdur that is built by the algorithm if it is not possibly usable:

▶ **Definition 13.** *A ptdur $\mathcal{T}$ is possibly usable if for every pair of distinct children $\mathcal{T}_i, \mathcal{T}_j$*

- *$inlet(\mathcal{T}_i) \cap inlet(\mathcal{T}_j) = \emptyset$, and*
- *$V(\mathcal{T}_i) \cap V(\mathcal{T}_j) \subseteq outlet(\mathcal{T}_i) \cap outlet(\mathcal{T}_j)$.*

The following lemma gives all candidates for leaves in a canonical tree decomposition.

▶ **Lemma 14.** *Let $\mathcal{T} = (T, (X_t)_{t \in V(T)})$ be a canonical tree decomposition of a graph $G$. Let $X_l$ be a bag associated with a leaf in $\mathcal{T}$. Then $X_l = N[v]$ for a vertex $v \in V(G)$.*

**Proof.** If $\mathcal{T}$'s only bag is $X_l$, then $X_l$ is a clique in $G$. This is argued as follows: Suppose $X_l$ is not a clique, then there exist $u, v \in X_l$ such that $(u, v)$ is not an edge in $G$. In that case the tree decomposition consisting of the bags $X_l \setminus \{v\}$ and $X_l \setminus \{u\}$ is another tree decomposition of $G$ whose bags are proper subsets of $X_l$. This contradicts Lemma 9.

Otherwise, $\mathcal{T}$ has at least two bags and thus a parent of the leaf associated with $X_l$ exists. Denote its bag by $X_p$. Since $\mathcal{T}$ is canonical and no bag is a subset of another bag, there exists $v \in X_l \setminus X_p$. Because of the consistency property, $v$ does not lie in any other bag

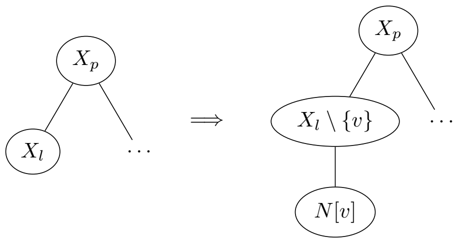


**Figure 2** We illustrate the proof of Lemma 14. Replacing the node with the bag $X_l$ by two nodes with the bags $X_l \setminus \{v\}$ and $N[v]$ in $\mathcal{T}$ yields an alternative tree decomposition of $G$, the existence of which leads to a contradiction.

of $\mathcal{T}$ and has therefore no neighbors outside of $X_l$. Hence, $N[v] \subseteq X_l$. Now suppose that $N[v] \subsetneq X_l$, i.e., there is $u \in X_l \setminus N[v]$. Then, because $v$ and $u$ are not adjacent in $G$, $G$ has an alternative tree decomposition where the node associated with $X_l$ is replaced by two nodes with the bags $N[v]$ and $X_l \setminus \{v\}$. Both of these bags, however, are subsets of $X_l$, so it follows with Lemma 9 that one of them is actually equal to $X_l$. This is impossible because $N[v]$ does not contain $u$, whereas $X_l \setminus \{v\}$ does not contain $v$. Thus, $X_l = N[v]$. ◀

A figure illustrating the proof is given in Figure 2. The following lemma generalizes this result to all bags of a canonical tree decomposition.

▶ **Lemma 15.** *Let $\mathcal{T}$ be a canonical tree decomposition of a graph $G$. Let $\mathcal{T}'$ be a ptd of $\mathcal{T}$ and let $O = \bigcup_{\mathcal{T}'_c \in children(\mathcal{T}')} outlet(\mathcal{T}'_c)$ and $W = \bigcup_{\mathcal{T}'_c \in children(\mathcal{T}')} V(\mathcal{T}'_c)$ denote the union of the outlets or vertices of its children, respectively. Then, for the root bag $X$ of $\mathcal{T}'$, one of the following conditions holds:*

1. *$X = O$,*
2. *$X = N[v]$ for a vertex $v \in V \setminus W$, such that $O \subseteq N[v]$, or*
3. *$X = O \cup (N(v) \setminus W)$ for a vertex $v \in O$.*

**Proof.** Throughout this proof, we refer to the example in Figure 6 in the Appendix.

We established earlier in this section that $outlet(\mathcal{T}_i^X) \subseteq bag(\mathcal{T}_i^X)$ for every $i \in I$ because all vertices in the outlet have to lie in at least one other bag of $\mathcal{T}$ outside of $\mathcal{T}_i^X$. Since by the consistency property all bags containing a given vertex have to be connected and the only other incident bag is $X$, that vertex must also be in $X$. By repeating this argument for all vertices in $O$, we get $O \subseteq X$.

If $O = X$, case 1 applies (see Figure 6b).

Otherwise, $O \subsetneq X$. We distinguish between two cases:

▶ **Case 1.** *There exists a vertex $v \in X \setminus (O \cup outlet(\mathcal{T}'))$.*

*The proof for this case is similar to the proof of Lemma 14 above. As $v$ is neither contained in the intersection of $X$ with the bag of the parent of $X$, nor in the intersection of $X$ with any bag of one of its children, it follows that $X$ is the only bag containing $v$ by the consistency property. Consequently, all edges incident to $v$ have to be covered by $X$, i.e., $N[v] \subseteq X$. Now suppose that $N[v] \subsetneq X$, i.e., there exists $u \in X \setminus N[v]$ (see Figure 6c). Then we can replace the node associated with $X$ by two nodes associated with the bags $X \setminus \{v\}$ and $N[v]$ to construct an alternative tree decomposition of $G$. The newly introduced bags are subsets of $X$ and by Lemma 9 one of them has to be equal to $X$. Because $X \setminus \{v\}$ does not contain $v$ and $N[v]$ does not contain $u$, however, we reach a contradiction. Hence, in this case, $X$ is of type 2 as claimed in the lemma.*

▶ **Case 2.** *The set $X \setminus (O \cup outlet(\mathcal{T}'))$ is empty.*

*In this case it follows that every vertex in $X \setminus O$ lies in $outlet(\mathcal{T}')$ and therefore also in the bag associated with the parent of $\mathcal{T}'$. At least one vertex $v \in O$ cannot lie in the parent's bag because otherwise $X$ would be a subset of it. Hence, all of $v$'s neighbors have to be covered somewhere within $\mathcal{T}'$. The vertices in $N(v)$ that do not appear in $W$, therefore, have to be covered by $X$. Thus, $N(v) \setminus W \subset X$.*

*It remains to show that there is only one such vertex $v$. Suppose this is false, i.e., there exist two vertices $u, v \in O \setminus outlet(\mathcal{T}')$ such that neither $N(u) \setminus W \subseteq N(v) \setminus W$ nor $N(v) \setminus W \subseteq N(u) \setminus W$ (see Figure 6f). Then we can replace the node associated with $X$ by two nodes associated with the bags $X \setminus \{v\}$ and $Y := X \setminus ((N(u) \setminus I) \setminus (N(v) \setminus I))$ to construct an alternative tree decomposition of $G$. The newly introduced bags are once again subsets of $X$, so by Lemma 9, one of them has to be equal to $X$. However, $X \setminus \{v\}$ does not contain $v$ and $u$ has a neighbor that is contained in $X$ but not in $Y$. Therefore, only one such vertex $v$ can exist and $X$ is of type 3 as claimed in the lemma.* ◀

These two lemmas are the basis for the algorithm. By enumerating all possible ptdurs and constructing all possible ptds with width at most $tw(G)$, a tree decomposition of $G$ is constructed eventually.

## 3.2 The Basic Algorithm

Algorithm 1 is a pseudocode version of the basic algorithm. In it, $P$ and $U$ are sets containing the ptds and ptdurs constructed by the algorithm, respectively. The `foreach` loops in lines 5 and 9 iterate over the mutated sets, i.e., also over elements that are added during the execution of the loops.

The algorithm constructs all ptds as stated in Lemmas 14 and 15 (lines 17–27) and all ptdurs by either setting the outlet of a ptd as the root bag and adding no further child (lines 6–7) or adding a new ptd as an additional child to a ptdur constructed before (lines 12–16). Notice that lines 9–10 ensure that the ptdur without an additional child is considered when constructing ptds from ptdurs in lines 17–27.

Intuitively, the algorithm constructs all ptds of height 0 by means of Lemma 14. Then we construct further ptds by constructing ptdurs having subsets of ptds as children: For each ptd $\mathcal{T}$ constructed, we construct a ptdur having $\mathcal{T}$ as its only child. Furthermore, we add $\mathcal{T}$ as an additional child to each ptdur already constructed. We remove all ptdurs that are not possibly usable anymore. Furthermore, we try to make a ptd out of each ptdur via Lemma 15. A formal proof of the correctness is as follows.

▶ **Theorem 16.** *Given $G$ and $k$, Algorithm 1 returns "YES" if and only if $tw(G) \leq k$.*

**Proof.** First, we notice that each element $\mathcal{T}$ added to $P$ is a tree decomposition of the graph induced by $V(T)$ and with a width of at most $k$. Further, all vercies of $V(\mathcal{T})$ adjacent to a node in $V(G) \setminus V(T)$ are in outlet($\mathcal{T}$). Hence, if the algorithm answers "YES", it has found a tree decomposition of $G$ with a width of $k$.

It remains to show that we find a tree decomposition of width $k$ if there exists one. Hence, we need to show that, at line 28, $P$ contains exactly those ptds of $G$ that have a width smaller or equal to $k$ and that line 28 is indeed reached.

By Lemma 14, all candidates for leaves are added to $P$ in lines 1–4. We now prove that, by the time the algorithm reaches line 28, $U$ contains all possibly usable ptdurs with a width of at most $k$ (and only those). Because lines 17–18, 19–22, and 23–27 correspond to step 2 in Lemma 15, it then follows that $P$ contains all ptds with width at most $k$ when line 28 is reached.

**Algorithm 1** Treewidth.

```
Input  : Graph $G$, positive integer $k$
Output : “YES” if $tw(G) \le k$, otherwise “NO”
foreach $v \in V(G)$ do
    if $|N[v]| \le k+1$ and $N[v]$ is pmc then
        $p_0$ := ptd with bag $N[v]$ as only bag
        add $p_0$ to $P$
foreach $\mathcal{T} \in P$ do
    $\widetilde{\mathcal{T}}$ := ptdur with $outlet(\mathcal{T})$ as root bag and $\mathcal{T}$ as only child
    add $\widetilde{\mathcal{T}}$ to $U$
    foreach $\mathcal{T}' \in U$ do
        if $\widetilde{\mathcal{T}} = \mathcal{T}'$ then
            $\widehat{\mathcal{T}} := \mathcal{T}'$
        else
            $\widehat{\mathcal{T}}$ := ptdur obtained from $\mathcal{T}'$ by adding $\mathcal{T}$ as a child and adding $outlet(\mathcal{T})$
              to the root bag
            if $\widehat{\mathcal{T}}$ is possibly usable and $bag(\widehat{\mathcal{T}})$ is cliquish and $|bag(\widehat{\mathcal{T}})| \le k+1$ then
                add $\widehat{\mathcal{T}}$ to $U$
            else
                continue
        if $bag(\widehat{\mathcal{T}})$ is pmc then
            add $\widehat{\mathcal{T}}$ to $P$
        foreach $v \in V(G) \setminus V(\widehat{\mathcal{T}})$ do
            if $bag(\widehat{\mathcal{T}}) \subseteq N[v]$ and $N[v]$ is pmc and $|N[v]| \le k+1$ then
                $p_2 := \widehat{\mathcal{T}}$, where the root bag is replaced by $N[v]$
                add $p_2$ to $P$
        foreach $v \in bag(\widehat{\mathcal{T}})$ do
            $B := bag(\widehat{\mathcal{T}}) \cup \left(N(v) \setminus inlet(\widehat{\mathcal{T}})\right)$
            if $B$ is pmc and $|B| \le k+1$ then
                $p_3 := \widehat{\mathcal{T}}$, where $B$ is added to the root bag
                add $p_3$ to $P$
if $P$ contains a ptd that covers $V(G)$ then
    return “YES”
else
    return “NO”
```

Denote by $p_1, p_2, \ldots$ the ptds constructed by the algorithm in the order that $P$ is iterated over.

▷ Claim 17. For each $n$, by the time the iteration over $p_n$ (line 5) finishes, $U$ contains all possibly usable ptdurs with a width of at most $k$, whose children are subsets of $\{p_1, \ldots, p_n\}$.

Proof. We prove the claim by induction.

*Base case $n = 1$:*

The ptdur with the only child $p_1$ has the same width as $p_1$ and is added to $U$ at line 7.

*Inductive hypothesis:*

Suppose the claim holds for all values of $n$ up to some $l, l > 1$.

*Inductive step:*

Let $n = l + 1$. By the inductive hypothesis, $U$ already contains all possibly usable ptdurs with a width of at most $k$ whose children are subsets of $\{p_1, \dots, p_{n-1}\}$. Notice that a ptdur that is not possibly usable does not become possibly usable by adding another child to it. Therefore, the only possibly usable ptdurs left to be constructed are of two types: (1) those that are already in $U$ but have $p_n$ as an additional child, and (2) the ptdur whose only child is $p_n$. The latter is added to $U$ at line 7, the rest is added at line 14. ◁

It remains to show that the algorithm terminates. Since $P$ and $U$ are sets and therefore do not contain duplicates, it is sufficient to argue that there exist only finitely many non equivalent ptds and ptdurs of $G$. As ptds $\mathcal{T}$ and $\mathcal{T}'$ are equivalent if $V(\tau) = V(\tau')$ there are at most $2^n$ ptds. Similarly, we have at most one ptdur with a certain vertex set as inlet and a certain vertex set as root bag and hence at most $2^{2n}$ in total. ◀

## 4 Reducing the number of PTDs and PTDURs

In the following sections we describe techniques for reducing the number of ptds and ptdurs considered. Some of these, namely those mentioned in Sections 4.1 to 4.4 and the technique regarding ptdurs with root bags of size $k + 1$ in Section 4.5, are also used by Tamaki's implementation. We review them here for the sake of completeness and to show how they can be utilized in the reinterpreted algorithm.

### 4.1 Rejecting PTDs with Non-Canonical Root Bags

If a vertex set is a potential maximal clique, it must be cliquish. Notice, however, that all of its subsets are cliquish as well. By contraposition, adding more vertices to a non-cliquish vertex set will not make it cliquish. We can use this fact to reject ptdurs whose roots are not cliquish, because none of the ptd candidates built from them can be actual ptds. Hence, we can reject those ptdurs immediately at line 12.

By Definition 7, the intersection between two adjacent bags in a ptd $\mathcal{T}$ is a minimal separator. For $\mathcal{T}$ to be a child of another ptd $\mathcal{T}'$, it is necessary that the intersection between $bag(\mathcal{T})$ and $bag(\mathcal{T}')$, namely $outlet(\mathcal{T})$, is a minimal separator. Consequently, all ptds whose outlet is not a minimal separator can be rejected.

### 4.2 Equivalence of PTDs and PTDURs

Let $\mathcal{T}$ be a canonical tree decomposition of $G$. For two adjacent bags $X$ and $Y$ of $\mathcal{T}$, denote by $\mathcal{T}^{X,Y}$ the tree that is obtained by splitting $\mathcal{T}$ at the edge between the nodes labeled $X$ and $Y$ and discarding the part containing the node labeled with $Y$. Notice that the $\mathcal{T}^{X,Y}$ can be replaced in $\mathcal{T}$ by any other tree $\mathcal{T}'$ with $V(\mathcal{T}') = V(\mathcal{T}^{X,Y})$. In this sense, $\mathcal{T}^{X,Y}$ and $\mathcal{T}'$ are equivalent. Consequently, we can adjust $P$ to accept a new ptd only if no equivalent ptd is already stored within.

To apply this principle also to ptdur's, we need to consider the size of their root bags as well. This is because a small root bag may be able to be extended to a pmc in more ways than a large root bag, when only bags of size at most $k + 1$ are allowed. See Figure 3 for an

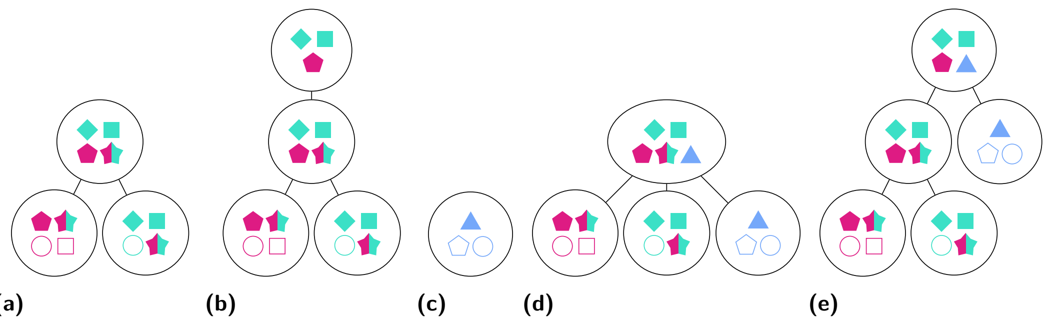


**Figure 3** We illustrate the equivalence of ptds. In this figure, each unique vertex is represented as a unique combination of a geometric shape and a color. Vertices in the outlet of a leaf are filled, while vertices in the inlet of a leaf are not. The ptdurs (a) and (b) cover the same vertex set, but are not equally useful. This becomes apparent when adding the ptd (c) as a child to them: the resulting ptdurs (d) and (e) have different widths. If the root bag of (e) is a pmc, then (e) is a ptd of width 3. Hence, during the iteration where $k = 3$, it would be wrongly skipped.

illustration of where this is relevant. Therefore, if we try to add a ptdur $\mathcal{T}$ with root bag $X$ to $U$, we first test whether an equivalent ptdur, whose root bag is a subset of $X$, exists already in $U$. If so, $\mathcal{T}$ is discarded. Otherwise, if a ptdur, whose root bag is a superset of $X$, exists in $U$, we replace that ptdur by $\mathcal{T}$. If none of these cases apply, $\mathcal{T}$ is added to $U$ as usual.

## 4.3 Choosing a Unique Root

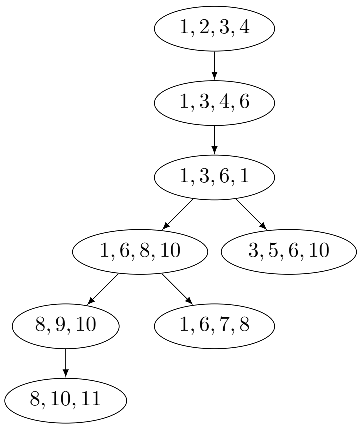
In order to define ptds in Section 3.1, we required the tree decomposition to be rooted. So far, we allow any bag to be the root. This means that we construct each tree decomposition not only once but once for every possible root (which can be any node). We overcome this problem by directing the edges of tree decompositions away from the root and rejecting ptds whose edges would not be consistent with that. We want to define a property called *incoming* for ptds that satisfies the following conditions:

1. for every two adjacent bags $X$ and $Y$ in $\mathcal{T}$, exactly one of $\mathcal{T}^{X,Y}$ and $\mathcal{T}^{Y,X}$ is incoming,
2. for every ptd with root bag $X$ and its neighbors $Y_1, ..., Y_k$ in $\mathcal{T}$, at most one of the $\mathcal{T}^{X,Y_i}, 1 \leq i \leq k$ is incoming, and
3. given a ptd, we can determine whether it is incoming using only information contained within it.

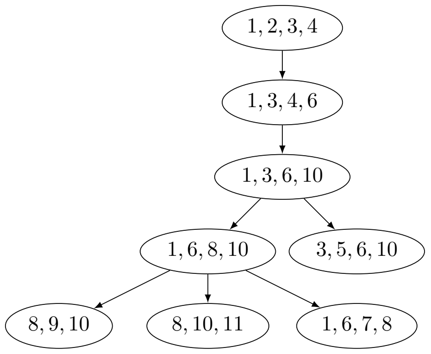
Condition 1 guarantees that each edge has a unique direction. Because we want the edges to be directed away from the root, every bag can have at most one incoming neighbor (condition 2), which is its parent. Condition 3 allows us to use this property in the algorithm. Only if a ptd is not incoming will we add it to $P$.

Assume that a total ordering $<$ on the vertices of $G$ is given. For a vertex set $U \subseteq V(G)$, we define $min(U)$ as its smallest element under $<$. If $U$ is empty, then we consider $min(U)$ to be smaller than any vertex. Then the following definition of incoming satisfies the conditions given above.

▶ **Definition 18.** *A ptd $\mathcal{T}$ is incoming if $min(inlet(\mathcal{T})) < min(V \setminus V(\mathcal{T}))$.*



**(a)** The tree decomposition from Figure 1b, where all ptds are required to be not incoming, thereby choosing $N[2] = \{1, 2, 3, 4\}$ as the unique root.



**(b)** The same tree decomposition, where all ptds are normalized. The bag $\{8, 10, 11\}$ has been attached to its former grandparent $\{1, 6, 8, 10\}$ instead.

**Figure 4** The tree decomposition shown in Figure 1b has leaves $N[2], N[5], N[7]$, and $N[11]$, so its unique root is chosen to be $N[2]$ (see Figure 4a). Notice, however, that $N[9] = \{8, 9, 10\}$ is also a candidate for a leaf, but is actually an inner node here. This means that it could have been chosen as the root under another ordering of vertices. While this is not a conflict when choosing a unique root, it illustrates another way in which some ptds are redundant. Consider the ptd consisting of only the bag $\{8, 10, 11\}$. Its outlet is $\{8, 10\}$, which is contained in its grandparent $\{1, 6, 8, 10\}$. It is therefore possible to attach the bag to its grandparent instead (see Figure 4b). We call a tree where this is not the case for any bag *normalized*.

It is not hard to see that the three properties mentioned above hold:

1. As $inlet(\tau^{X,Y}) = V \setminus V(\mathcal{T}^{Y,X})$ and vice versa.
2. The sets $inlet(\tau^{X,Y_i})$ are pairwise disjoint. Let $v$ be the smallest vertex in the union of these sets and $\ell^*$ be such that $v \in inlet(\tau^{X,Y_{\ell*}})$. For each $\ell \neq \ell^*$, we have $v \in V \setminus V(\mathcal{T}^{X,Y_\ell})$ and each node in $inlet(\tau^{X,Y_\ell})$ is larger than $v$. Hence $T^{X,Y_\ell}$ is not incoming.
3. As the definition uses only $inlet(\tau)$ and $V(\tau)$.

Since $\mathcal{T}$ is equivalent to ptds with the same vertex set or, equivalently, the same inlet, we also say that $inlet(\mathcal{T})$ is incoming if a ptd $\mathcal{T}$ is incoming. For every tree decomposition, this procedure will choose the leaf $N[v]$ as the root, for which $v$ is smallest out of all the leaves contained in that tree decomposition.

An example is shown in Figure 4a.

## 4.4 Normalization of PTDs

Notice that if the outlet of a ptd $\mathcal{T}'$ used in the construction of the ptd $\mathcal{T}$ is contained in several bags of $\mathcal{T}$, we can attach $\mathcal{T}'$ to any one of those bags. To avoid enumerating all these possibilities, we normalize the ptd $\mathcal{T}$ by enforcing that $\mathcal{T}'$ is attached as close to the root of $\mathcal{T}$ as possible. This is captured in the following definition.

▶ **Definition 19.** *Let $p(t)$ denote the parent of a node $t$ within a rooted tree.*

*A ptd $\mathcal{T} = (T, (X_t)_{t \in V(T)})$ is normalized if there is no $t \in V(T)$ such that $outlet(T_t^+) \subseteq X_{p(p(t))}$.*

This means that we can discard a ptd $\mathcal{T}$ if $outlet(\mathcal{T}_C) \subseteq outlet(\mathcal{T})$ for one of its children $\mathcal{T}_C$.

## 4.5 Rejecting PTDURs Whose Root Cannot Be Extended To a Potential Maximal Clique

The roots of some of the ptdurs created at line 12 are not pmcs. If no vertices can be added to them so that they finally become pmcs either, they are useless and can be discarded.

First, we consider a ptdur whose root contains $k+1$ vertices, i.e., we can add no more vertices to it. If it is not a pmc, then the ptdur is useless. Next, we consider a ptdur whose root $X$ contains $k$ vertices, i.e., we are allowed to add one more vertex to it. If $X$ is a pmc, we accept the ptdur as usual. If not, we can test if, for any candidate $v \in V(G) \setminus V(\mathcal{T})$, $X \cup \{v\}$ is a pmc. This test may take a lot of processing time, so we narrow down the list of candidates by studying necessary conditions on candidates.

Let $C$ be the component associated with $X$ that contains a candidate $v$. We can assume that $X$ is cliquish because otherwise it will be rejected at line 13. In order for $X \cup \{v\}$ to be cliquish as well, there has to exist a path between $v$ and $x$ for each $x \in X$ that does not lead through other vertices in $X$. This implies that $C$ must be a full component because otherwise there would exist $x \in X$ such that $x \notin N(C)$ and hence, every path from $v$ to $x$ would contain another vertex of $X$.

On the other hand, in order to be a pmc, $X \cup \{v\}$ can have no full components associated with it, which leaves us with two possibilities: either $v$ separates $G[C]$ into at least two components, neither of which is full in $G$ with respect to $X \cup \{v\}$. Or $v$ does not separate $G[C]$, in which case there needs to exist $x \in X$ such that $x \notin N(C \setminus \{v\})$, i.e., $v$ is the only neighbor of $x$ in $C$.

## 4.6 PTD Outlets With More Than 2 Associated Components

Let $S$ be a minimal separator of $G$ with associated components $C_1, C_2, ..., C_\ell$ such that $\ell \geq 3$. Without loss of generality, let $C_1$ be the incoming component, let $C_2$ be a full (non-incoming) component, and let $\mathcal{T}_2$ be a ptd with $\mathit{inlet}(\mathcal{T}_2) = C_2$. Because $C_2$ is a full component associated with $S$, we have $\mathit{outlet}(\mathcal{T}_2) = S$.

▶ **Lemma 20.** *If $\mathcal{T}_2$ is contained in a tree decomposition of $G$ with a width of at most $k$, then there exist non-incoming ptds $\mathcal{T}_3, ..., \mathcal{T}_\ell$, whose inlets are $C_3, ..., C_\ell$, respectively, and whose widths are at most $k$*

**Proof.** Notice that any tree decomposition of $G$ that contains the ptd $\mathcal{T}_2$ is also a tree decomposition of the graph $G\langle S\rangle$. Let $\mathcal{T}$ be such a tree decomposition and let $C$ be any component associated with $S$. From all bags of $\mathcal{T}$, remove all vertices except those contained in $S \cup C$. The resulting tree decomposition is a tree decomposition of $G\langle S\rangle[S \cup C]$ with a width of at most $k$. By Lemma 8, that graph also has a canonical tree decomposition of equal or smaller width. Interpreted as a ptd of $G$, it has inlet $C$ as claimed. ◀

The contrapositive of this lemma states that $\mathcal{T}_2$ is only useful if we can also construct $\mathcal{T}_3, ..., \mathcal{T}_\ell$ (or equivalent ptds). We can use this insight to delay the addition of $\mathcal{T}_2$ to $P$ until ptds covering all non-incoming components associated with $S$ are constructed.

## 4.7 Using Upper and Lower Bounds

As another approach, we tried to heuristically complete ptds (who cover only part of the graph) to a tree decomposition on the entire graph, using the min-degree and min-fill heuristics. If a heuristic succeeds in finding a tree decomposition of the complete graph with the given width, we can immediately stop the computation and return the result.

Furthermore, we can compute lower bounds for a tree decomposition that contains a given ptd. We complete the bag of the root to a clique and compute a lower bound of the resulting graph. If this lower bound is larger than the given treewidth, we can remove the ptd of our list. To improve the time to compute the lower bound, we could only consider the part of the graph that is not covered by the ptd. A very simple lower bound is the second lowest degree. As in [9], we determine a lower bound by heuristically computing a minor of $G$ and subsequently trying to increase this very simple lower bound.

## 5 Further Details of the Algorithm

We implemented several preprocessing methods that reduce the size of the graph before entering the main loop. More precisely, we implemented the simplicial vertex rule, almost simplicial vertex rule, buddy rule, and cube rule, all of which can be found in [4]. All these rules are based on criteria guaranteeing that the complete neighborhood of a vertex is contained in single bag in any tree decomposition. Completing the neighborhood to a clique and removing the vertex itself results in a graph with one vertex less such that any tree decomposition still contains the neighborhood in a single bag. Hence, we can compute the tree decomposition of the resulting graph and add a bag containing the vertex removed together with its neighborhood to the tree decomposition and add it at the appropriate position.

Furthermore, we implemented the heuristic to find safe separators of [14]. A separator $S$ for $G$ is called safe if the treewidth of $G$ is the maximum of the treewidth of the graphs $G[C \cup S]\langle S\rangle$ for the components $C$ of $G[V(G) \setminus S]$. Having computed a safe separator, we can compute tree decompositions independently for each subgraph and connect the trees at the bags containing the separator. Tamaki's heuristic tries to construct separators and use a sufficient criterion of a separator $S$ to be safe proven by Bodlaender and Koster [8], namely that for each component $C$ of $G[V(G) \setminus S]$ the graph $G[V(G) \setminus C]$ has $S$ as a labeled minor. Furthermore, we implemented an algorithm to find separators that are cliques or almost cliques as given in [8].

We extend the usage of safe separators as follows: whenever we construct a new ptd, we test whether its outlet is a safe separator with Tamaki's heuristic, unless the separator has already been tested before.

Instead of testing at the end of each iteration whether we have built a ptd covering $V(G)$, we test this condition for every ptd as it is added to $P$. If so, we return "YES" immediately.

To avoid trying to add every ptd constructed to every ptdur, we implemented (a slight variant of) the block sieve as presented in [14]. In this data structure, the ptdurs are sorted by the size of their root bags. If the size of the outlet of the current ptd plus the size of the root bag of a ptdur is larger than the current value of $k$, these sets have to coincide on a number of vertices. For each size, the ptdurs are stored in tries, which allow to efficiently iterate over the elements coinciding in a given number of vertices.

## 6 Experiments

We have compared the performance of our implementation against Tamaki's on the PACE2017 public instances in the *treewidth exact* track. The experiments were conducted in the following environment: Intel Core i5-6600K@3.5GHz CPU, 16GB DDR4 RAM, Windows 10 (64 bit), Java version jre1.8.0_271, .NET version 4.8. The time measured is the CPU time, which includes the time for garbage collection.

For the sake of a fair comparison, we initially enabled only those optimizations that Tamaki also uses in his implementation, namely the following: (1) the rejection of ptdurs with non-cliquish root bags, (2) the rejection of ptds whose outlet is not a minimal separator, (3) the rejection of ptds and ptdurs for whom equivalent ptds or ptdurs have already been built, (4) accepting only non-incoming ptds, (5) rejecting ptds that are not normalized, and (6) rejecting ptdurs whose root bag contains $k+1$ vertices but is not a pmc. Furthermore, we used only the safe separators found by Tamaki's heuristic for this experiment and no further preprocessing.

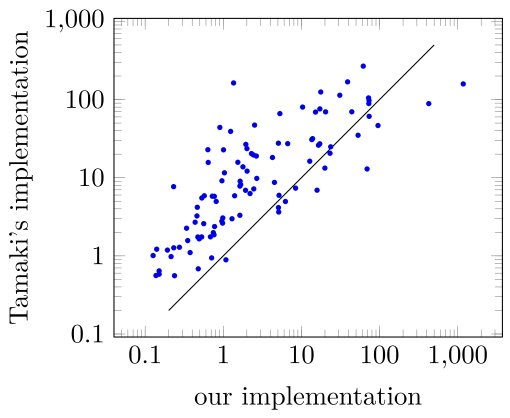
We plot the results in Figure 5a. Every dot corresponds to one of the test instances and its position corresponds to the running times of Tamaki's and our implementation on that instance. A point located above the diagonal line indicates that our implementation is faster. In total, Tamaki's implementation solves the instances in 46 minutes, 24 seconds, whereas we solve the instances in 44 minutes, 7 seconds. Although the difference is small, our implementation beats Tamaki's on 86 instances. This is mainly due to two instances that together make up more than half of the total running time of our implementation. Conversely, the remaining instances are often solved rather quickly.

In Figure 5b, we plot the running time of Tamaki's implementation against ours with the additional techniques of Section 4 and the preprocessing discussed in Section 5 enabled. Our implementation to compute lower bounds is not used as it is not efficient enough. Although the number of ptds and ptdurs are reduced by 5% and 49%, respectively, the running time increased by about 36%. The blue dots represent the running times, where all strategies for ptd and ptdur reduction are enabled, along with the further strategies discussed in Section 5. The strategy to reject ptdurs whose root bag contains $k$ vertices and cannot be extended to a ptd and the search for safe separators during the runtime of the algorithm turned out to be slightly disadvantageous, but only by a small margin. We therefore also plot in red the running time when they are disabled. The total running time with all and only the best strategies enabled, respectively, are 14 minutes, 31 seconds and 13 minutes, 8 seconds, outperforming Tamaki's implementation on 91 and 93 instances, respectively. On larger instances than the ones tested here, however, the techniques excluded from best could actually be beneficial.

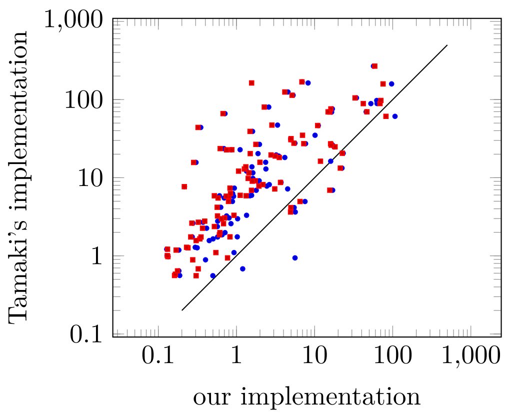
We have also evaluated the impact of the strategies to reject ptdurs whose root bag has $k$ vertices and cannot be extended to a pmc and to delay the addition of ptds to $P$ until ptds covering all of its non-incoming components are found. We counted the numbers of ptds and ptdurs enumerated[1] either until we stop as soon as the first (optimal) solution is found or until no further ptds and ptdurs with the current width could be found with various strategies enabled. The later numbers can be interpreted as the size of the search space for each graph. Since we implemented the algorithm such that it returns immediately when a tree decomposition is found, the first give the numbers of the actually enumerated objects which is often far less than that. Our machine ran out of memory while computing the size of the search space for the instance ex003.gr, so it is not included in the results.

The results are summarized in Table 1. The percentages give the average relative reduction of the ptds and ptdurs, respectively, compared to when only the other strategies discussed in this paper are used. Only a small impact can be attributed to the strategy that delays the addition of a ptd to $P$ until ptds covering all its other non-incoming components are found (abbreviated as *>2 comp.*). The strategy to reject ptdurs whose root bag contains $k$ elements

[1] Note that we split graphs at safe separators into smaller graphs, so the graphs we use to count them are actually only subgraphs of the test instances.



**(a)** A comparison, where both implementations use the same preprocessing and strategies for reducing combinatorial objects.



**(b)** A comparison, where our implementation uses all preprocessing discussed in Section 5 and all (blue) and only the best (red) strategies for reducing the amount of combinatorial objects.

**Figure 5** A comparison of running times in seconds between Tamaki's and our implementation. Every dot represents one instance.

**Table 1** The average relative reduction of ptd's and ptdur's when the newly developed strategies for reducing their numbers are employed. A dash signals that that strategy is not able to reduce the number of the respective objects by its conception.

| | until first solution found | | | until all ptd(ur)s are enumerated | | |
|---|---|---|---|---|---|---|
| average relative reduction... | pmc $k$ | >2 comp. | all | pmc $k$ | >2 comp. | all |
| ... of ptds | – | 4.61% | 4.61% | – | 0.08% | 0.08% |
| ... of ptdurs | 34.78% | 2.52% | 37.29% | 44.74% | 0.06% | 44.78% |

but cannot be extended to a pmc (abbreviated as *pmc k*), however, has a huge impact on the number of ptdurs enumerated. Unfortunately, as mentioned above, determining if any candidate vertices can extend such a bag to pmc takes a comparatively long time, which has a slightly negative impact on the total running time over all instances.

Finally, we tried to solve all DIMACS graph coloring instances. For each instance we set a time limit of 30$min$. Table 2 in the appendix gives a summary of our finding. For our implementation and the one by Tamaki, we give the running time and the lower bound obtained after the time limit.

## 7 Conclusion

We gave a description of Tamaki's algorithm to compute the treewidth of a graph that is considerably more accessible than the original formulation. This is archived by the interpretation of the enumerated structures as partial tree decompositions and partial tree decompositions with unfinished root. Furthermore, seeing this interpretation allows us to remove some of the structures from the enumeration. This results in an algorithm that is more efficient in practice.

In future work, we want to derive and implement further techniques to reduce the number of enumerated partial tree decompositions, including better or faster lower bounds. Furthermore, we plan to extend the usage of techniques known for preprocessing within the construction phase.

## References

1 Ernst Althaus and Sarah Ziegler. Optimal tree decompositions revisited: A simpler linear-time FPT algorithm. *CoRR*, abs/1912.09144, 2019. `arXiv:1912.09144`.

2 Ernst Althaus and Sarah Ziegler. Optimal tree decompositions revisited: A simpler linear-time fpt algorithm. In: Gentile, C., Stecca, G., Ventura, P. (eds) Graphs and Combinatorial Optimization: from Theory to Applications (CTW2020 Proceedings), 2020. AIRO Springer Series, vol 5. Springer, 2021.

3 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM JOURNAL OF DISCRETE MATHEMATICS*, 8(2):277–284, 1987.

4 Hans Bodlaender, Arie Koster, Frank Eijkhof, and Linda C. Gaag. Pre-processing for triangulation of probabilistic networks, April 2002.

5 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. `doi:10.1137/S0097539793251219`.

6 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016. `doi:10.1137/130947374`.

7 Hans L. Bodlaender and Arie M. C. A. Koster. Safe separators for treewidth. *Discret. Math.*, 306(3):337–350, 2006. `doi:10.1016/j.disc.2005.12.017`.

8 Hans L. Bodlaender and Arie M. C. A. Koster. Safe separators for treewidth. *Discrete Math.*, 306(3):337–350, 2006.

9 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. lower bounds. *Inf. Comput.*, 209(7):1103–1119, 2011. `doi:10.1016/j.ic.2011.04.003`.

10 Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.

11 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. `doi:10.1007/978-3-319-21275-3`.

12 Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics*, 23, 2018. `doi:10.1145/3173045`.

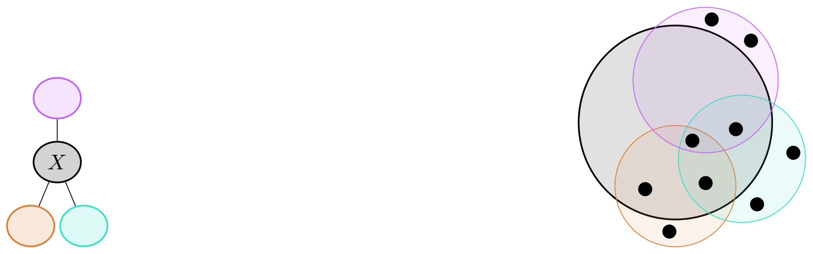
13 Neil Robertson and P.D Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. `doi:10.1016/0196-6774(86)90023-4`.

14 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 68:1–68:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ESA.2017.68`.

15 Hisao Tamaki. A heuristic use of dynamic programming to upperbound treewidth. *CoRR*, abs/1909.07647, 2019. `arXiv:1909.07647`.

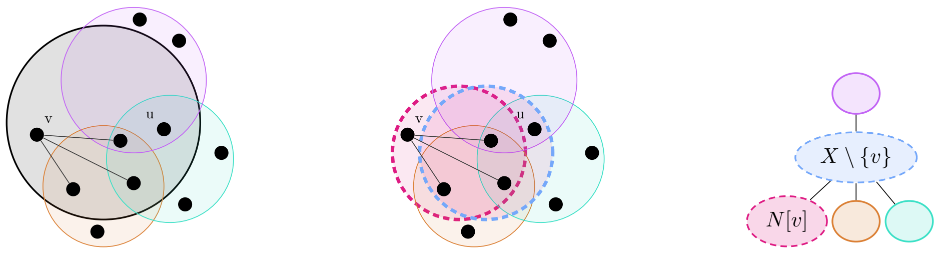
16 Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000.

## A Appendix



**(a)** A part of $\mathcal{T}$: the root node of $\mathcal{T}'$ with the bag $X$, its children, and its parent.

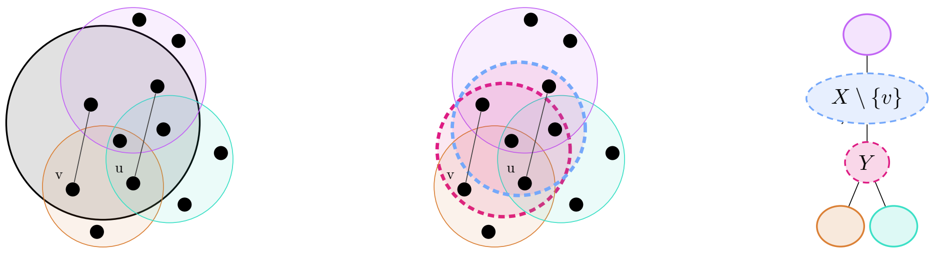
**(b)** Case $O = X$ of Lemma 15. Shown are vertices of $G$ and their membership of the bags depicted in Figure a.



**(c)** Case 1 of Lemma 15. Assuming that $u$ does not lie in $N[v]$, ...

**(d)** ... one can replace the node associated with $X$ by nodes associated with the bags depicted here as dotted circles, ...

**(e)** ... thus, the part of $\mathcal{T}$ shown in Figure (a) can be replaced by the one shown here.



**(f)** Case 2 of Lemma 15. Assuming that both $u$ and $v$ have different neighbors that are not covered in $\mathcal{T}'$s children, ...

**(g)** ... one can replace the node associated with $X$ by nodes associated with the bags depicted here as dotted circles, ...

**(h)** ... thus the part of $\mathcal{T}$ shown in Figure (a) can be replaced by the one shown here.

**Figure 6** Illustration of the proof of Lemma 15. Figure 6a shows the part of $\mathcal{T}$ induced by $X$ and its adjacent bags, that is, its children (if any) and possibly a parent. The Figures 6b, 6c and 6f show the vertices of $G$ that are covered by the bags in Figure 6 in three different cases. Each bag of the tree decomposition is represented by a circle of the corresponding color and contains all of the vertices within it. Edges are only shown where they are necessary for the argument.

**Table 2** The running time in seconds and the lower bound on the treewidth for our implementation and the one of Tamaki with a time limit of 30*min*. It should be noted that a slower computer was used for this comparison than for the calculations before.

| | Our impl. | | Tamaki | |
|---|---|---|---|---|
| Name | tw | time | tw | time |
| anna | 12 | 0,079 | 12 | 0,286 |
| david | 13 | 0,044 | 13 | 0,234 |
| DSJC1000.1 | >0 | 1800 | >187 | 1800 |
| DSJC1000.5 | >0 | 1800 | >739 | 1800 |
| DSJC1000.9 | >0 | 1800 | >986 | 1800 |
| DSJC125.1 | >34 | 1800 | >35 | 1800 |
| DSJC125.5 | >107 | 1800 | 108 | 829,5 |
| DSJC125.9 | 119 | 2,137 | 119 | 0,069 |
| DSJC250.1 | >62 | 1800 | >65 | 1800 |
| DSJC250.5 | >205 | 1800 | >210 | 1800 |
| DSJC250.9 | 243 | 30,26 | 243 | 0,993 |
| DSJC500.1 | >106 | 1800 | >113 | 1800 |
| DSJC500.5 | >369 | 1800 | >384 | 1800 |
| DSJC500.9 | 492 | 718,7 | 492 | 27,05 |
| DSJR500.1 | >22 | 1800 | >23 | 1800 |
| DSJR500.1c | 485 | 319,3 | 485 | 5,176 |
| DSJR500.5 | >225 | 1800 | 246 | 1187 |
| flat1000_50 | >0 | 1800 | >733 | 1800 |
| flat1000_60 | >0 | 1800 | >734 | 1800 |
| flat1000_76 | >0 | 1800 | >735 | 1800 |
| flat300_20 | >228 | 1800 | >244 | 1800 |
| flat300_26 | >230 | 1800 | >246 | 1800 |
| flat300_28 | >230 | 1800 | >246 | 1800 |
| fpsol2.i.1 | 66 | 16,28 | 66 | 1256 |
| fpsol2.i.2 | 31 | 32,51 | >0 | 1800 |
| fpsol2.i.3 | 31 | 31,68 | >0 | 1800 |
| games120 | >27 | 1800 | >28 | 1800 |
| homer | 30 | 1189 | >27 | 1566 |
| huck | 10 | 0,002 | 10 | 0,027 |
| inithx.i.1 | 56 | 65,80 | >0 | 1800 |
| inithx.i.2 | >31 | 1800 | >0 | 1800 |
| inithx.i.3 | 31 | 1751 | >0 | 1800 |
| jean | 9 | 0,002 | 9 | 0,008 |
| le450_15a | >73 | 1800 | >45 | 1052 |
| le450_15b | >75 | 1800 | >44 | 958,5 |
| le450_15c | >122 | 1800 | >130 | 1800 |
| le450_15d | >121 | 1800 | >129 | 1800 |
| le450_25a | >76 | 1800 | >23 | 581,6 |
| le450_25b | >75 | 1800 | >28 | 709,3 |
| le450_25c | >112 | 1800 | >105 | 1800 |

| | Our implementation | | Tamaki | |
|---|---|---|---|---|
| Name | tw | time | tw | time |
| le450_25d | >112 | 1800 | >109 | 1800 |
| le450_5a | >62 | 1800 | >58 | 1800 |
| le450_5b | >63 | 1800 | >59 | 1800 |
| le450_5c | >94 | 1800 | >98 | 1800 |
| le450_5d | >93 | 1800 | >97 | 1800 |
| miles1000 | 49 | 1,274 | 49 | 0,496 |
| miles1500 | 77 | 3,606 | 77 | 0,870 |
| miles250 | 9 | 0,012 | 9 | 0,023 |
| miles500 | 22 | 0,467 | 22 | 0,149 |
| miles750 | 36 | 1,885 | 36 | 0,348 |
| mulsol.i.1 | 50 | 0,531 | 50 | 159,8 |
| mulsol.i.2 | 32 | 64,28 | 32 | 1424 |
| mulsol.i.3 | 32 | 65,80 | 32 | 1480 |
| mulsol.i.4 | 32 | 67,16 | 32 | 1494 |
| mulsol.i.5 | 31 | 67,97 | 31 | 1572 |
| myciel2 | 2 | 0,001 | 2 | 0,001 |
| myciel3 | 5 | 0,002 | 5 | 0,002 |
| myciel4 | >0 | 1800 | 10 | 0,004 |
| myciel5 | 19 | 23,71 | 19 | 0,621 |
| myciel6 | >25 | 1800 | >34 | 1800 |
| myciel7 | >42 | 1616 | >33 | 1213 |
| queen10_10 | >66 | 1800 | >68 | 1800 |
| queen11_11 | >73 | 1800 | >76 | 1800 |
| queen12_12 | >80 | 1800 | >83 | 1800 |
| queen13_13 | >86 | 1800 | >90 | 1800 |
| queen14_14 | >92 | 1800 | >97 | 1800 |
| queen15_15 | >98 | 1800 | >103 | 1800 |
| queen16_16 | >104 | 1800 | >105 | 1800 |
| queen5_5 | 18 | 0,054 | 18 | 0,007 |
| queen6_6 | 25 | 0,103 | 25 | 0,024 |
| queen7_7 | 35 | 1,550 | 35 | 0,484 |
| queen8_12 | >63 | 1800 | 65 | 1613 |
| queen8_8 | 45 | 18,04 | 45 | 9,145 |
| queen9_9 | 58 | 1369 | 58 | 651,2 |
| school1 | >123 | 1800 | >122 | 1800 |
| school1_nsh | >112 | 1800 | >108 | 1800 |
| zeroin.i.1 | 50 | 8,668 | 50 | 39,48 |
| zeroin.i.2 | 32 | 1,635 | 32 | 185,4 |
| zeroin.i.3 | 32 | 1,634 | 32 | 186,9 |

# Practical Implementation of Encoding Range Top-2 Queries

Seungbum Jo
Chungbuk National University, Cheongju, South Korea

Wooyoung Park
Seoul National University, South Korea

Srinivasa Rao Satti
Norwegian University of Science and Technology, Trondheim, Norway

## Abstract

We design a practical variant of an encoding for *range Top-2 queries (RT2Q)*, and evaluate its performance. Given an array $A[1,n]$ of $n$ elements from a total order, the range Top-2 encoding problem is to construct a data structure that can answer RT2Q queries, which return the positions of the first and the second largest elements within a given query range of $A$, without accessing the array $A$ at query time. Davoodi et al. [Phil. Trans. Royal Soc. A, 2016] proposed a $(3.272n + o(n))$-bit encoding, which answers RT2Q queries in $O(1)$ time, while Gawrychowski and Nicholson [ICALP, 2015] gave an optimal $(2.755n + (n))$-bit encoding which doesn't support efficient queries. In this paper, we propose the first practical implementation of the encoding data structure for answering RT2Q. Our implementation is based on an alternative representation of Davoodi et al.'s data structure. The experimental results show that our implementation is efficient in practice, and gives improved time-space trade-offs compared to the indexing data structures (which keep the original array $A$ as part of the data structure) for range maximum queries.

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

**Keywords and phrases** Range top-2 query, Range minimum query, Cartesian tree, Succinct encoding

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.10

**Supplementary Material** *Software (Source Code)*: `https://github.com/wyptcs/R2MQ`
archived at `swh:1:dir:684698b8ae0bcc6ada509f22f8ff743411de26d9`

**Funding** Seungbum Jo was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1G1A1101477).

## 1 Introduction

Given an array $A[1,n]$ of $n$ elements from a total order, the *range maximum query* on $A[i,j]$ (denoted by $\mathsf{RMQ}(i,j)$) returns the position of the largest element in $A[i,j]$. We assume that all elements in $A$ are distinct (if there are equal elements, we can break the ties according to their positions, i.e., the leftmost one is considered as the largest value among them). The problem of constructing space and/or time-efficient data structures for answering RMQ is one of the fundamental problems in data structures, and has been extensively studied both theoretically and practically [2, 8, 9].

In general, the data structures for answering specific queries can be categorized into two types: (i) *indexing data structures*, and (ii) *encoding data structures*. In indexing data structures, one can access the input array $A$ at query time, while it is not allowed in encoding data structures. For many problems including RMQ problem, the minimum size for an encoding data structure (referred to as *effective entropy* [17]) is much less than the input size – for example, the effective entropy for answering *RMQ* on $A$ is $2n - o(n)$ bits [9], whereas storing $A$ requires at least $n \log n$ bits[1], if all the elements in $A$ are distinct. Thus,

[1] throughout the paper, we use log to denote the logarithm to the base 2

encoding data structures can be highly space-efficient in some cases compared to their indexing counterparts. Recent results [2, 8] show that encoding data structures for RMQ perform well both in theory and in practice.

In this paper, we consider the problem of answering *range Top-2 queries*, which extends the RMQ problem. The range Top-2 query on $A[i, j]$ (denoted by RT2Q) returns the positions of the largest and the second largest elements in $A[i, j]$. If $k = \mathsf{RMQ}(i, j)$, one can easily observe that the position of the second largest element in $A[i, j]$ is one of $k_1 = \mathsf{RMQ}(i, k-1)$ or $k_2 = \mathsf{RMQ}(k+1, j)$. Thus, any indexing data structure for answering RMQ also can answer RT2Q by comparing $A[k_1]$ and $A[k_2]$. Davoodi et al. [4] proposed the first encoding data structure for answering RT2Q in $O(1)$ time using $3.272n + o(n)$ bits, which is close to the effective entropy of $2.755n - \Theta(polylog(n))$ bits [11] for RT2Q. However, their encoding is not very practical since it represents the *Cartesian tree* [20] of $A$ succinctly using the *tree-covering* approach of Farzan and Munro [7], which is hard to implement (compared to other succinct tree representations [1]). In this paper, we give the first practical implementation of an encoding for RT2Q. Our implementation is based on the data structure of Davoodi et al. [4], but instead of using the tree-covering approach, we use the DFUDS representation [3] of *2d-max heap* [9] which is easier to implement, and works well in practice. Our implementation supports RT2Q in $\log n \cdot g(n)$ time, for any increasing function $g(n) = \omega(1)$, using at most $3.5n + o(n)$ bits. The experimental results show that our data structure gives a better space-time trade-off, compared to the indexing data structures for RT2Q (that have access to the input array $A$, along with an auxiliary data structure for answering the RMQ queries).[2]

## 2 Preliminaries

### 2.1 Range Maximum Queries and Cartesian Trees

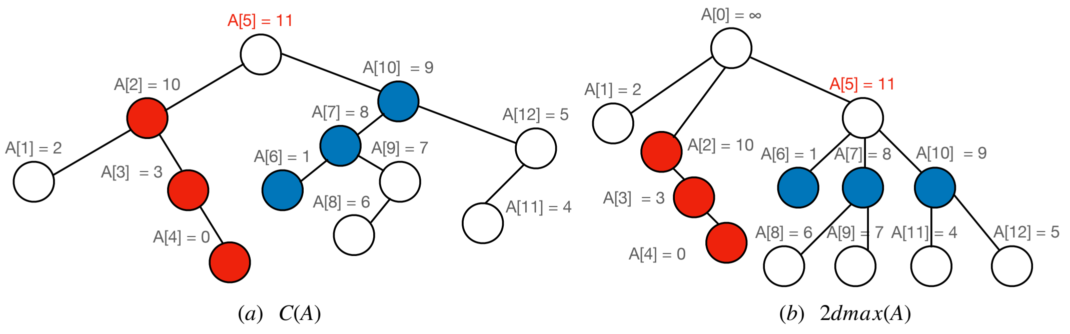
Given an array $A[1, n]$ of size $n$, the *Cartesian tree* [20] of $A$, denoted by $C(A)$, is a binary tree where (i) the root node of $C(A)$ corresponds to $A[i]$ where $i = \mathsf{RMQ}(1, n)$, and (ii) the left and right subtrees of $C(A)$ are the Cartesian trees of $A[1, i-1]$ and $A[i+1, n]$ respectively. From the definition, the $i$-th node in the inorder traversal of $C(A)$ corresponds to the $i$-th position of $A$ (see Figure 1 (a) for an example). In the rest of this paper, we refer to the nodes in the Cartesian tree by their inorder numbers (i.e., their corresponding positions in the array $A$). Also, one can convert the RMQ problem on $A$ into the LCA (lowest common ancestor) problem on $C(A)$ [10]. More precisely, for any $i, j \in [1, n]$, $\mathsf{RMQ}(i, j)$ is the same as $\mathsf{LCA}(i, j)$, which is the LCA of the node $i$ and $j$ in $C(A)$. This implies that one can support RMQ on $A$ by storing $C(A)$ instead if $A$ (thus this gives an encoding for answering RMQ on $A$). All the existing encoding data structures for answering RMQ use a Cartesian tree or its variants.

### 2.2 Davoodi et al.'s encoding data structure for RT2Q

We introduce the $(3.272n + o(n))$-bit data structure of Davoodi et al. [4], which answers RT2Q in $O(1)$ time on an array $A[1, n]$ of size $n$. Their data structure answers the $\mathsf{RT2Q}(i, j)$ query by performing the following three steps:

1. Compute and return the position $k = \mathsf{RMQ}(i, j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compare $A[k_1]$ and $A[k_2]$, and return $k_1$ if $A[k_1] > A[k_2]$, or $k_2$ otherwise.

[2] our implementation is available at `https://github.com/wyptcs/R2MQ`.



$$A[1,12] = 2 \ \ 10 \ \ 3 \ \ 0 \ \ 11 \ \ 1 \ \ 8 \ \ 6 \ \ 7 \ \ 9 \ \ 4 \ \ 5$$

$$D(2dmax(A)) = (\ (\ (\ (\ )\ )\ (\ )\ (\ )\ )\ (\ (\ (\ )\ )\ (\ (\ )\ )\ )\ (\ (\ )\ )\ )$$

$$S_2 = 1 \quad S_5 = 0\ 1\ 1\ 0\ 1 \quad S_7 = 1\ 1 \quad S_{10} = 0\ 0\ 1 \quad S_1, S_3, S_4, S_6, S_8, S_9, S_{11}, S_{12} = \epsilon$$

$$S = 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1$$

**Figure 1** Example of (a) $C(A)$ and (b) $2dmax(A)$ of the array $A[1, 12]$. Red and blue colored nodes are the nodes in linspine(5) and rinspine(5) of $C(A)$ respectively.

For answering $k = \mathsf{RMQ}(i, j)$, they maintain the *tree-covering* [7] representation of $C(A)$ to support LCA queries in $O(1)$ time, using $2n + o(n)$ bits. Next, to compare $A[k_1]$ and $A[k_2]$ without storing $A$, they store the *spine sequence* $S$ of $A$ defined as follows. For any node $i$ which has left child $i_l$ and right child $i_r$, let *left spine* (resp., *right spine*) of $i$, denoted by $\mathsf{lspine}(i)$ (resp., $\mathsf{rspine}(i)$), be the path from the node $i$ to the leftmost (resp., rightmost) descendant of $i$. Also, let *left inner spine* (resp., *right inner spine*) of $i$, denoted by $\mathsf{linspine}(i)$ (resp., $\mathsf{rinspine}(i)$), be the $\mathsf{rspine}(i_l)$ (resp., $\mathsf{lspine}(i_r)$), and define $L_i$, $R_i$, $l_i$, and $r_i$ to be a number of nodes in $\mathsf{lspine}(i)$, $\mathsf{rspine}(i)$, $\mathsf{linspine}(i)$, and $\mathsf{rinspine}(i)$ respectively. Then by the property of $C(A)$, the nodes $k_1$ and $k_2$ in $C(A)$ are always on $\mathsf{linspine}(k)$ and $\mathsf{rinspine}(k)$, respectively. Now we define the array $S_k[1, m_k]$ to be a bit array of size $m_k = \max(l_k + r_k - 1, 0)$ where $S_k[i] = 0$ if the $i$-th largest element of $A$ among the positions corresponding to $\mathsf{linspine}(k) \cup \mathsf{rinspine}(k)$ is in $\mathsf{linspine}(k)$, and 1 otherwise. Let $\mathsf{depth}(k)$ be the depth of the node $k$, and for any given pattern $b$ and sequence $S$, let $\mathsf{rank}_b(S, i)$ be the number of occurrences of $b$ in the first $i$ positions of $S$, and $\mathsf{select}_b(S, i)$ be the position of $i$-th occurrence of $b$ in $S$. Then one can compare $A[k_1]$ and $A[k_2]$ by comparing $\mathsf{select}_0(S_k, \mathsf{depth}(k_1) - (\mathsf{depth}(k) + 1) + 1)$ and $\mathsf{select}_1(S_k, \mathsf{depth}(k_2) - (\mathsf{depth}(k) + 1) + 1)$ (i.e., by checking which of the two bits corresponding to the nodes $k_1$ and $k_2$ come first in $S_k$). The sequence $S$ is simply defined by concatenating all $S_k$'s for all nodes $k \in C(A)$ in the increasing order of their inorder numbers. Finally, to locate the starting position of $S_k$ in $S$ efficiently, they introduce the following lemma.

▶ **Lemma 1** ([4]). *For any $u \in C(A)$,*

$$\sum_{j<u} m_j = 2u - L_\tau - l_u + \mathit{Ldepth}(u) - \mathit{Rdepth}(u) + 1 - (u - \mathit{Lleaves}(u))$$

In the above lemma, $\tau$ denotes the root of $C(A)$. Also, for any node $u \in C(A)$, $\mathsf{Ldepth}(u)$ (resp., $\mathsf{Rdepth}(u)$) denotes the number of nodes which have their left (resp., right) child, in the path from $\tau$ to $u$. Finally, $\mathsf{Lleaves}(u)$ denotes the number of leaf nodes $v \in C(A)$ which satisfies $v < u$.

Davoodi et al. [4] showed that all the operations used in the lemma can be computed in $O(1)$ time using the tree covering representation of $C(A)$ along with some auxiliary data structures. Furthermore, they showed that the size of $S$ is at most $1.5n$, which implies that there exists the data structure for answering RT2Q in $O(1)$ time using at most $3.5n + o(n)$ bits. With further optimization, they improved the space usage to $3.272n + o(n)$ bits while still supporting RT2Q in $O(1)$ time.

▶ **Example 2.** We show how to answer the RT2Q$(3, 9)$ on the array $A[1, 12]$ in Figure 1 using $C(A)$ with the spine sequence $S$ of $A$. First, we compute and return RMQ$(3, 9)$ = LCA$(3, 9) = 5$. Next, to compare $A[3]$ and $A[7]$ (note that RMQ$(3, 4) = 3$ and RMQ$(6, 9) = 7$), we first locate the starting position of $S_5$ in $S$ by $\sum_{j<5} m_j = 2 \cdot 5 - 3 - 3 + 0 - 0 + 1 - (5 - 2) = 2$. Since depth$(5) = 0$, depth$(3)$ = depth$(7) = 2$ and select$_0(S_5, 2) >$ select$_1(S_5, 2)$, we return 7 as the position of the second largest element in $A[3, 9]$.

## 3 A Practical Implementation

Davoodi et al.'s data structure [4] in the previous section uses the tree-covering method for encoding $C(A)$, which is not practical compared to other succinct tree representations such as BP (balanced parenthesis) [13] and DFUDS (depth-first unary degree sequence) [3]. In this section, we describe a practical implementation of Davoodi et al.'s data structure for answering RT2Q on $A[1, n]$, which uses the DFUDS representation of the *2d-max heap* of $A$ [9]. We first describe the general definition of DFUDS and 2d-max heap, and show how to convert Davoodi et al.'s data structure using these tools.

### 3.1 DFUDS and 2d-max heap

Given an ordinal tree $T$ with $n$ nodes, DFUDS of $T$ (denoted by $D(T)$) is a balanced parenthesis sequence of size $2n$ defined as follows. (i) if $n = 1$, $D(T)$ is (). (ii) Otherwise, if $T$ has $k$ subtrees $T_1, T_2, \ldots, T_k$, $D(T)$ is $(^{k+1})$ followed by $d(T_1), d(T_2), \ldots, d(T_k)$, where $d(T_i)$ is $D(T_i)$ with the first open parenthesis removed (see Figure 1 for an example). Since $D(T)[1, 2n]$ is a balanced parenthesis sequence, one can define two operations findopen$(i)$ / findclose$(i)$ which find the matching open / closed parenthesis of the closed / open parenthesis in $D(T)[i]$. It is known that by storing a $o(n)$-bit auxiliary structure along with $D(T)$, one can support rank, select, findopen and findclose operations in $O(1)$ time. This in turn enables us to represent $T$ to support a comprehensive list of navigation queries on $T$ in $O(1)$ time using $2n + o(n)$ bits [16] (see Table 2 in [1] for the list of operations).

One of the main reasons for using the tree-covering based approach for representing $C(A)$ in Davoodi's et al.'s structure, is to find the $i$-th node in the inorder traversal of $C(A)$ (let this operation be inorder$(i)$). To our best knowledge, one cannot support this operation on the BP or DFUDS of $C(A)$ (note that LCA can be supported in $O(1)$ time on both BP and DFUDS [1]). Sadakane [19] showed that (i) if the difference between any two consecutive values in $A$ is $\pm 1$, then one can answer RMQ on $A$ (we refer the such query as $\pm 1$RMQ) in $O(1)$ time using $2n + o(n)$ bits, and (ii) for general $A$, one can support both inorder and LCA operations on $D(C(A))$ (thus, RMQ on $A$) in $O(1)$ time using $4n + o(n)$ bits, by converting $C(A)$ into a ternary tree by adding a dummy leaf to each node in $C(A)$.

Fischer and Heun [9] proposed the *2d-max heap* to support RMQ without the need for the inorder operation. The 2d-max heap on $A$ (denoted by $2dmax(A)$) is an alternative representation of $C(A)$, defined as follows. $2dmax(A)$ is an ordered tree with $n + 1$ nodes, where for $1 \le i \le n$,

1. The $i$-th node in the preorder traversal of $2dmax(A)$ corresponds to $A[i - 1]$ (we assume that $A[0] = \infty$). In the rest of this paper, we refer to this node as node $(i-1) \in 2dmax(A)$. Therefore, the root of $2dmax(A)$ is 0.
2. For any non-root node $i \in 2dmax(A)$, the parent of $i$ is the node $j$ where $j$ is the rightmost position in $A[0, i - 1]$ such that $A[i] < A[j]$.

The above definition implies that for $1 \le i \le n$, the node $i \in C(A)$ and the node $i \in 2dmax(A)$ both correspond to the position $i$ in $A$. The example of Figure 1 (a) and (b) shows the $C(A)$ and $2dmax(A)$ of input array $A$ respectively. Fischer and Heun also showed that $\mathsf{RMQ}(i,j)$ operation can be supported in $O(1)$ time by using $D(2dmax(A))$ along with $o(n)$-bit auxiliary structures for supporting $\mathsf{rank}$, $\mathsf{select}$, $\mathsf{findopen}$, and $\pm 1\mathsf{RMQ}$ queries on $D(2dmax(A))$ – using $2n + o(n)$ bits in total.

## 3.2 Practical implementation of encoding RT2Q

In this section, we propose an alternative implementation of the data structure of [4] on $A$ using $D(2dmax(A))$. Since one can support $\mathsf{RMQ}$ using $D(2dmax(A))$ [9], it is enough to show how to find the position of the second largest element in $A[i,j]$. One can observe that, for any node $k$ in $C(A)$, the nodes on the $\mathsf{linspine}(k)$ in $C(A)$ are the same as the nodes on the right spine of the previous sibling of $k \in 2dmax(A)$. (The left/right spine of a node $i \in 2dmax(A)$ is defined as the path from node $i$ to the leftmost/rightmost descendant of $i$.) Also the nodes on $\mathsf{rinspine}(k)$ in $C(A)$ are the same as the children of $k \in 2dmax(A)$. We define the spine sequence $S$ of $A$, analogous to the same sequence in [4] (that is, concatenating all the $S_k$'s for each non-root node $k \in 2dmax(A)$ according to their preorder value in $2dmax(A)$). Then we can answer $\mathsf{RT2Q}(i,j)$ using the following procedure:

1. Compute and return the position $k = \mathsf{RMQ}(i,j)$.
2. Compute $k_1 = \mathsf{RMQ}(i, k-1)$ and $k_2 = \mathsf{RMQ}(k+1, j)$.
3. Compute two nodes $k_l = \mathsf{presibling}(k)$ and $k_r = \mathsf{childrank}(k_2)$ in $2dmax(A)$ where $\mathsf{presibling}(k)$ denotes the previous sibling of $k$, and $\mathsf{childrank}(k_2)$ denotes the number of left siblings of $k_2$.
4. Locate the starting position of $S_k$ in $S$, and return $k_1$ if $\mathsf{select}_0(S_k, \mathsf{depth}(k_1) - \mathsf{depth}(k_l) + 1) < \mathsf{select}_1(S_k, k_r)$, or $k_2$ otherwise.

Note that the operations used in the above procedure ($\mathsf{RMQ}$, $\mathsf{presibling}$, $\mathsf{childrank}$, and $\mathsf{depth}$) can be supported in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [14]. Also, to locate the position of $S_k$ in $S$, we need to compute $\sum_{1 \le j < k} m_j$ (recall that $m_j = |S_j|$). The following lemma shows that we can compute each of the terms in Lemma 1 (therefore, $\sum_{1 \le j < k} m_j$) using $2dmax(A)$.

▶ **Lemma 3.** *Given an array $A[1,n]$ of size $n$ where all elements in $A$ are distinct, the following properties hold for any node $k$ in the Cartesian tree, $C(A)$, of $A$.*

1. *$l_k$ (number of nodes in $\mathsf{linspine}(k)$) is equal to the number of nodes in $\mathsf{rspine}(k_l)$ in $2dmax(A)$, where $k_l = \mathsf{presibling}(k)$.*
2. *$\mathsf{Ldepth}(k)$ is equal to the number of right siblings of all the nodes on the path from node $k$ to the root in $2dmax(A)$.*
3. *$\mathsf{Rdepth}(k) = d_k - 1$, where $d_k$ is the depth of $k \in 2dmax(A)$.*
4. *$\mathsf{Lleaves}(k)$ is equal the number of leftmost children $u < k$ which are also leaves in $2dmax(A)$.*

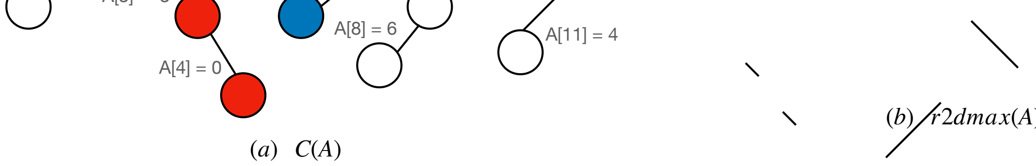
**Proof.**

1. Let $i_0 < k$ be the rightmost position of $A$ which satisfies $\mathsf{RMQ}(i_0, k) \neq k$. Then by the definition of $C(A)$, $\mathsf{linspine}(k)$ is composed of the nodes $\{i_1, i_2, \dots, i_{l_k}\}$ of $C(A)$ where $i_j = \mathsf{RMQ}(i_{j-1}+1, k-1)$. Thus, if $l_k > 0$, the node $k$ in $2dmax(A)$ always has the previous sibling $k_l$ (otherwise, $k$ has no left child in $C(A)$, which implies $l_k = 0$). Furthermore, since $k-1$ is the rightmost leaf of the subtree of $2dmax(A)$ rooted at $k_l$, all the nodes $i_1, i_2, \dots, i_{l_k}$ are on the $\mathsf{rspine}(k_l)$ in $2dmax(A)$.

2. Let $Lpath(k)$ be the set of nodes in $C(A)$ which have their left child in the path from $k$ to the root (hence, $|Lpath(k)|= \mathsf{Ldepth}(k)$). Now suppose $Lpath(k) = \{i_1, i_2, \ldots, i_{\mathsf{Ldepth}(k)}\}$ where $i_1 < i_2 < \cdots < i_{\mathsf{Ldepth}(k)}$. Then for any $j \in \{1, 2, \ldots, \mathsf{Ldepth}(k)\}$, (i) $i_j > k$, and (ii) $\mathsf{RMQ}(k, i_j) = i_j$. Therefore, for the node $k \in 2dmax(A)$, $Lpath(k)$ is the same as the set of nodes in $2dmax(A)$ which are the right siblings of the nodes on the path from the node $k$ to the root.
3. Similar to the case of $\mathsf{Ldepth}(k)$, let $Rpath(k)$ be the set of nodes in $C(A)$ which have their right child in the path from $k$ to the root (hence, $|Rpath(k)|= \mathsf{Rdepth}(k)$). Then $Rpath(k)$ consists all the nodes $i_j$ in $C(A)$ which satisfy: (i) $i_j < k$, and (ii) $\mathsf{RMQ}(i_j, k) = i_j$. Thus by the definition of $2dmax(A)$, $Rpath(k)$ is the same as the set of proper ancestors of $k$ in $2dmax(A)$.
4. Note that a node in $C(A)$ is a leaf if and only if its corresponding node in $2dmax(A)$ is a leftmost child which is also a leaf. Thus, set of all leaf nodes in $C(A)$ are the same as the set of all leftmost children $u < k$ which are also leaves in $2dmax(A)$. ◀

Now we describe how to compute each value in Lemma 1 using $D(2dmax(A))$ with Lemma 3.

1. $L_\tau$: The node $\tau = \mathsf{RMQ}(1, n)$ is the rightmost child of the node 0 (the root of $2dmax(A)$). Also all the nodes of $\mathsf{lspine}(\tau)$ in $C(A)$ are on the left siblings of $\tau$ in $2dmax(A)$. Thus this value can be computed in $O(1)$ time by $\mathsf{degree}(0)$ (note that $\mathsf{degree}$ can be computed in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures [1]).
2. $l_k$: By Lemma 3, $\mathsf{linspine}(k)$ of $C(A)$ are the same as the $\mathsf{rspine}(k_l)$ of $2dmax(A)$. Since the rightmost leaf of $k_l$ is $k - 1$, This can be computed in $O(1)$ time by $\mathsf{depth}(k - 1) - \mathsf{depth}(k_l) + 1$.
3. $\mathsf{Ldepth}(k)$: For $k \in 2dmax(A)$, let $L(k)$ be the number right siblings of the nodes on the path from the node $k$ to the root in $2dmax(A)$. Now we describe how to compute $L(k)$ using $D(2dmax(A))$. Let $d$ be a depth of $2dmax(A)$, and suppose $f(n) = \log n \cdot g(n)$ where $g(n)$ is any increasing function which satisfies $g(n) = \omega(1)$. Then we fix the value $0 \le i < f(n)$, and define the array $E$ which stores all the values of $L(k)$ for every node $k \in 2dmax(A)$ whose depth is $i + j \cdot f(n)$, for all $0 \le j \le \lfloor (d - i)/f(n) \rfloor$. The values in $E$ are stored according to the preorder number of corresponding nodes in $2dmax(A)$. By a simple counting argument, we can choose $i$ to satisfy $|E| \le n/f(n)$. Thus, at most $n/f(n) \cdot \log n = o(n)$ bits of space are necessary to store $E$. In addition to that, we maintain the bit array $B[1, n]$ of size $n$ where for $1 \le i \le n$, $B[i] = 1$ if and only if the $L(i)$ is stored in $E$. Using the data structure of Raman et al. [18], we can store $B$ using $\log \binom{n}{f(n)} + o(n) = o(n)$ bits while supporting $\mathsf{rank}$ queries in $O(1)$ time (we can also access any position of $B$ in $O(1)$ time by two $\mathsf{rank}$ queries). To answer $L(k)$, we initialize the counter $c = 0$, and start the scanning nodes on $Lpath(k)$ starting from the node $k$. During this scan, when we are at node $j$, we first check $B[j]$. If $B[j] = 0$, we increase $c$ to be $c + r$ where $r = \mathsf{degree}(\mathsf{parent}(j)) - \mathsf{childrank}(j)$ (note that $\mathsf{parent}$ can be computed in $O(1)$ time using $D(2dmax(A))$ [1]), and move to the parent of $j$. If $B[j] = 1$, we return $L(k) = c + \mathsf{rank}_1(B, j)$. Thus, using $D(2dmax(A))$ with $o(n)$-bit auxiliary structures, we can answer $L(k)$ in $O(f(n))$ time.
4. $\mathsf{Rdepth}(k)$: By Lemma 3, this is the same as the number of proper ancestors of $k$ in $2dmax(A)$, which can be computed $O(1)$ time by $\mathsf{depth}(k) - 1$.
5. $\mathsf{Lleaves}(k)$: By Lemma 3, this is the same as the number of leftmost children $u < k$ which are also leaves in $2dmax(A)$. This value can be computed by counting the number of occurrences of the pattern '()))' before the closing parenthesis corresponding to node $k$ in $O(1)$ time using $D(2dmax(A))$ with $o(n)$-bit auxiliary data structures [15].



(*a*) *C*(*A*) (*b*) *r*2*dmax*(*A*)

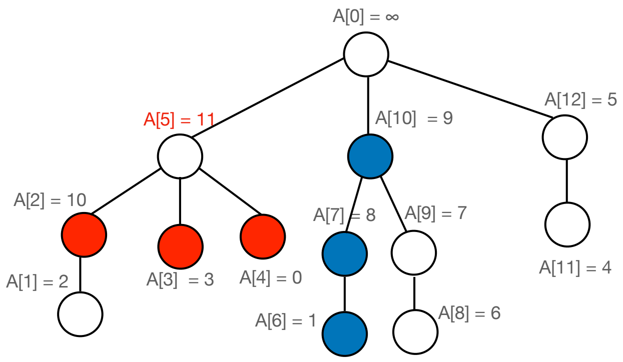


**Figure 2** $r2dmax(A)$ of the array in Figure 1. Red and Blue colored nodes correspond to the nodes in $C(A)$ in Figure 1 with the same colors.

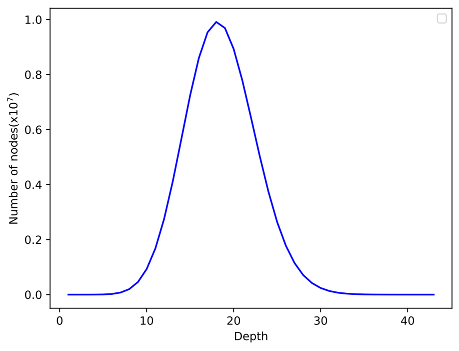
▶ **Example 4.** We show how to locate the starting position of $S_5$ in $S$ using the $2dmax(A)$ in Figure 1 (b). From node $5 \in 2dmax(A)$ in the figure, one can observe that $5_l = 2$, $L(5) = 0$, and $\mathsf{select}_{)}(D(2dmax(A)), 5) + 1 = 12$. Also $\mathsf{degree}(0) = 3$, $\mathsf{depth}(4) - \mathsf{depth}(2) + 1 = 3$, $\mathsf{depth}(5) - 1 = 0$, and $\mathsf{rank}_{())}(D(2dmax(A)), 12) = 2$. Thus, the starting position of $S_5$ in $S$ is $\sum_{j<5} m_j = 2 \cdot 5 - 3 - 3 + 0 - 0 + 1 - (5 - 2) = 2$.

We summarize the result in the following theorem.

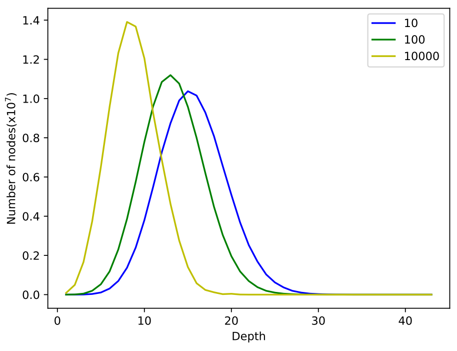
▶ **Theorem 5.** *Given an array $A[1, n]$ of size $n$, RT2Q on $A$ can be computed in $O(\log n \cdot g(n))$ time, for any increasing function $g(n) = \omega(1)$. The data structure uses at most $1.5n + o(n)$ additional bits, along with the DFUDS sequence of the 2d-max heap of $A$, $D(2dmax(A))$.*

**Alternative representation of $2dmax(A)$.** In practice, the performance of the data structure of Theorem 5 highly depends on the depth of $2dmax(A)$. To reduce the depth of $2dmax(A)$, Ferrada and Navarro [8] considered *rightmost-path* $2dmax(A)$ (denoted as $r2dmax(A)$), which can be obtained from $C(A)$ by applying $\tau_1$ (first-child, next-sibling) transformation [5]. Note that the original $2dmax(A)$ can be obtained from $C(A)$ by applying $\tau_4$ (previous-sibling, last-child) transformation [5]. One can observe that $i$-th position of $A$ corresponds to the node in $r2dmax(A)$ whose postorder number is $i$. (See Figure 2 for an example.) For example, if $A$ is strictly decreasing array from 1 to $n$, the depths of $2dmax(A)$ and $r2dmax(A)$ are $n$ and 1, respectively. Ferrada and Navarro [8] showed that one can answer RMQ queries on $A$ as using $r2dmax(A)$ with $o(n)$-bit auxiliary structures, which are different from the structures used for answering the same query using $2dmax(A)$. Baumstark et al. [2] showed that $r2dmax(A)$ is isomorphic to $2dmax(\overleftarrow{A})$, where $\overleftarrow{A}$ is an array of size $n$ constructed by reversing the all elements of $A$. Thus, one can simulate the $\mathsf{RMQ}(i, j)$ on $A$ using $r2dmax(A)$ by answering $\mathsf{RMQ}(n + 1 - j, n + 1 - i)$ on $\overleftarrow{A}$ using $2dmax(\overleftarrow{A})$ (note that in this case, one breaks the ties with rightmost policy when constructing $2dmax(\overleftarrow{A})$, i.e., among all the equal elements in a range, the rightmost element is considered as the largest).

To implement the data structure of Theorem 5, we first check the depth of $2dmax(A)$ and $2dmax(\overleftarrow{A})$ at pre-processing step, and maintain the one with smaller depth (along with the auxiliary structures).



**(a)**



**(b)**

**Figure 3** (a) The distribution of the depth of nodes in $2dmax(A)$. (b) The distribution of the depth of the nodes in $2dmax(A)$ which correspond to the RMQ of $A$.

## 4 Experimental results

Our data structure is implemented in C++ (compiled by g++ 9.3.0 with O3 optimization), and all the experiments were done on the Desktop PC (Intel i7-9900KS CPU with 128GB of RAM). We use the input array $A[1,n]$ which stores 32-bit unsigned integers. We consider three different types of input arrays: (a) *random*, (b) *pseudo-increasing*, and (c) *pseudo-decreasing*, where each $A[i]$ is randomly generated from the range (a) $[1,n]$, (b) $[i-\delta, i+\delta]$, and (c) $[n-i-\delta, n-i+\delta]$, respectively for given parameter $\delta > 0$. We compare the space usage (bits per element) and query time ($\mu$s) of our encoding structure (referred to as R2MQ-ENCODING) with the following four indexing data structures for answering RT2Q: (i) $A$ + RMQ encoding of Fisher and Heun [9] (FH-DFUDS), (ii) $A$ + RMQ encoding of Ferrada and Navarro [8] (FN-BP), (iii) $A$ + RMQ encoding of BaumStark et al. [8] (BGHL-BP), and (iv) $A$ + Fischer and Huen's indexing data structure for RMQ queries [9] (FH-INDEXING). Note that the encoding of (i) uses $D(2dmax(A))$, and both the encoding of (ii) and (iii) use the BP of $2dmax(A)$. For (i) and (ii), we use the implementation of Ferrada and Navarro [8][3], and for (iii), we use the implementation of BaumStark et al. [8][4]. Finally for (iv), we use our own implementation.

To support RMQ on $A$, and navigation queries on $2dmax(A)$ except depth, we use sdsl-lite [12] to support rank, select, findopen, findclose (for presibling operation), and $\pm 1$RMQ on $D(2dmax(A))$. Note that for findopen, findclose and $\pm 1$RMQ, we use a simplified RMM-tree [16] which maintains only the *min* field for these queries. For computing depth$(k)$ queries, we use the same data structure for computing $L(k)$. More precisely, if $L(k)$ is stored in $E$, we also store depth$(k)$ in a separate array $E'$ at the same position (thus, the same bit array $B$ can be used for $L(k)$ and depth$(k)$). For computing depth$(k)$, we perform the parent query iteratively until we find the node whose depth is stored in $E'$. Note that we do not keep any additional data structures for both depth and $L(K)$ queries if the depth of the tree is less than $\lceil \log n \rceil$.

[3] the codes are available `https://github.com/hferrada/rmqFischerDFUDS` and `https://github.com/hferrada/rmq`.

[4] the code is available at `https://github.com/kittobi1992/rmq-experiments`

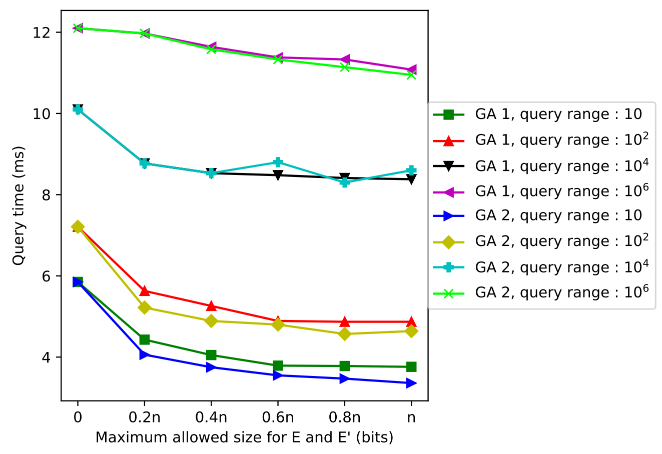


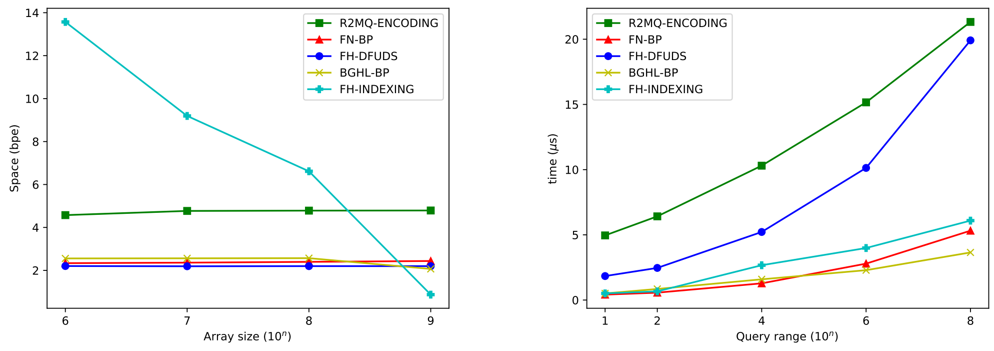
**Figure 4** Query time based on the allotted space for $E$ and $E'$.

Since the overhead for depth and $L(k)$ is the main drawback of our implementation, we do an empirical evaluation to decide the sizes of $E$ and $E'$. When $A$ is a randomly generated array of size $10^8$, the depth of $2dmax(A)$ is less than 50 in most cases (in theory, the expected depth of $C(A)$ for a random array $A$ is about $\Theta(\log n)$, and the depth of $2dmax(A)$ is at most the depth of $C(A)$ [6]), and the depth of nodes has close to the normal distribution (see Figure 3(a)). Next, we evaluate the distribution of the depth of the nodes $2dmax(A)$ which correspond to the RMQ of $A$ (note that we only need the value of depth$(k)$ and $L(k)$ when $k =$ RMQ$(i,j)$ for some $1 \leq i \leq j \leq n$). As shown in Figure 3(b), when the query range is $10^4$, the depth of all the nodes corresponding to RMQ is less than half of the depth of $2dmax(A)$. Furthermore, even for the small query ranges (10), still, the depth of 95.5% of the nodes is less than half of the depth of $2dmax(A)$. From the distribution of the nodes corresponding to RMQ of $A$, we consider two greedy algorithms for selecting the nodes to be stored in $E$ and $E'$. Suppose at most $N$ nodes can be stored in $E$ and $E'$, and let $D_N$ be the smallest depth where the number of the nodes with depth $D_N$ is more than $N$ (if there is no such depth, $D_N$ is the depth of $2dmax(A)$); and let $d$ be the value $\min(\lfloor D/2 \rfloor, D_N - 1)$, where $D$ is the depth of $2dmax(A)$. Then the greedy algorithm 1 (GA1) repeats the following procedure from $i = 0$ to $d$:

1. Choose all the nodes with depth $i$, if the total number of chosen nodes is at most $N$.
2. Increase $i$ by 1.

Similarly, the greedy algorithm 2 (GA2) repeats the first step of the above procedure by decreasing the value $i$ from $d$ to 0.

We evaluate the time for answering RT2Q with different amounts of space allotted for $E$ and $E'$. As shown in Figure 4, increasing the allotted space does not significantly improve the query time when the size of the query range is $10^6$ since most of the nodes corresponding to the answer of RMQ are close to the root node. The same tendency is shown for other sizes of query ranges (10, $10^2$, and $10^4$) when allotting more than $0.4n$ bits for $E$ and $E'$, since both GA1 and GA2 cannot significantly increase the number of nodes to be stored (note that the number of nodes increases roughly exponentially with the depth, from 1 to $d$). In our implementation, we choose GA2 which shows better query time for small query ranges. Also, the space allotted for storing $E$ and $E'$ is determined based on the maximum values



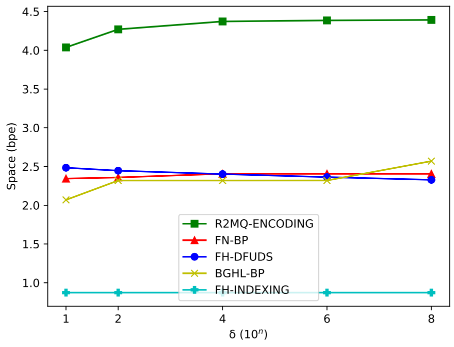
**(a)** **(b)**

**Figure 5** (a) The space (without the input array) on random array, and (b) Query time on the random array of size $10^9$.

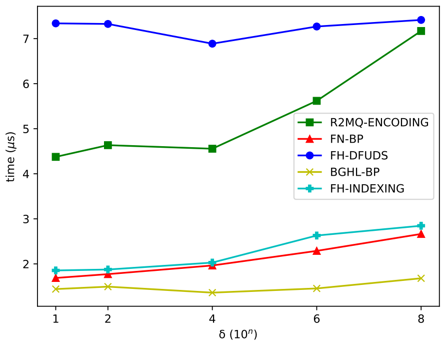
(either 8, 16 or 32-bit values) stored in those arrays, as follows. We allot $n/\lfloor \log \log n \rfloor$ bits if the maximum values of $E$ and $E'$ are both at most $2^8$ (in this case, we use 8-bit integer arrays for storing these). In general, if the maximum values of $E$ and $E'$ are at most $2^{8c}$ and $2^{8c'}$, respectively, for some $c, c' \in \{1, 2, 4\}$, then we allot $(\frac{c+c'}{2})n$ bits for storing these arrays. For example, if 32 and 8-bit integer arrays are necessary to store $E$ and $E'$ respectively for an array $A$ of size $10^8$, we use $(\frac{4+1}{2}) \cdot n/\lfloor \log \log 10^8 \rfloor = 0.625n$ bits for storing $E$ and $E'$.

Next, we evaluate the space usage on randomly generated arrays of size $n = 10^7$ to $n = 10^9$ (see Figure 5 (a)). Our structure uses up to 4.6 and 4.8 bpe (bits per element) for $n = 10^7$ and $n = 10^9$ respectively. This shows that our data structure's average space is not much changed by increasing the array size, like other indexing structures except FH-INDEXING. For FH-INDEXING, each pre-computed value needs 32 bits even for an array of size $10^6$ (note that $\lceil \log 10^6 \rceil$ is 19), which is wasteful in terms of space. Since the input array is necessary to answer $\mathsf{RT2Q}$ queries using indexing data structures, our data structure takes at least 7.1 times less space than the existing indexing data structures. Next, we fix the size of the (randomly-generated) input array to be $10^9$, and evaluate query time for various query ranges (see Figure 5 (b)). Our data structure and FH-DFUDS are highly dependent on the query range, compared to BP-based indexing structures. This is because, in the implementation, the running time of $\mathsf{findopen}$ operation is an increasing function of the range (note that $\mathsf{findopen}$ operation is used for computing $\mathsf{RMQ}$, $\mathsf{depth}$, and $L(k)$ when 2d-max heap is represented by DFUDS). Interestingly, when the query range is changed from $10^6$ to $10^8$, the query time of FH-DFUDS increases much rapidly than our data structures. This shows that the overhead for answering $\mathsf{RMQ}$ on FH-DFUDS is more than computing $L(k)$ and $\mathsf{depth}(k)$ for the nodes with small depths. The query time on FH-INDEXING is also rapidly increased by increasing the query range because the structure needs to access more sub-structures when the query range increases. Compared to the fastest indexing solution (BGHL-BP and FH-INDEXING), our data structure shows up to 10 and 4.1 times slower query times when the query range is 10 and $10^8$ respectively and shows better time-space trade-off for most cases except the small query ranges up to 100.

Next, we evaluate the space and query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ with various $\delta$ values (see Figure 6 and 7. the size of the query range is fixed to $\sqrt{n}$.). Note that when $A$ is pseudo-increasing (resp. pseudo-decreasing), $\overleftarrow{A}$ is pseudo-decreasing (resp. pseudo-increasing). Thus, our data structure and BGHL-BP show

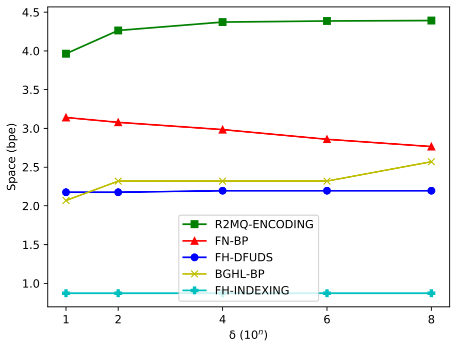


**(a)**

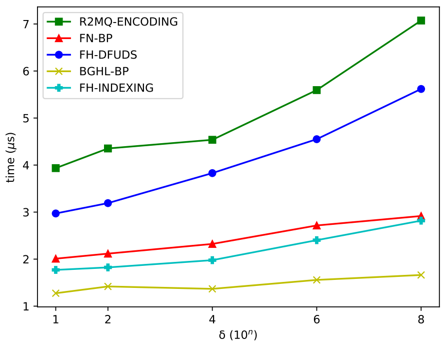


**(b)**

**Figure 6** The (a) space usage (without the input array) and (b) query time on the pseudo-increasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil\sqrt{n}\rceil = 31623$.



**(a)**

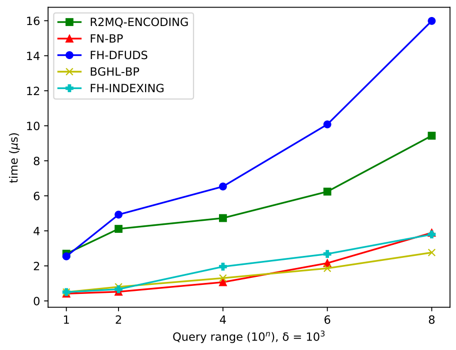


**(b)**

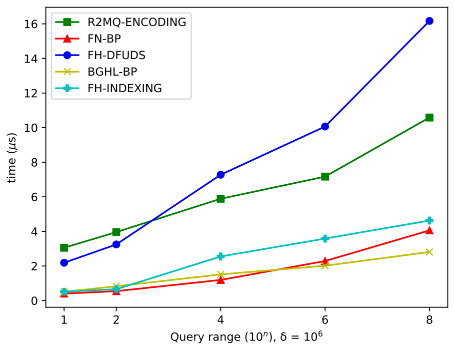
**Figure 7** The (a) space usage (without the input array) and (b) query time on the pseudo-decreasing array of size $n = 10^9$. The size of the query range is fixed to $\lceil\sqrt{n}\rceil = 31623$.

similar space usage and query time on both pseudo-increasing and pseudo-decreasing arrays (note that for FH-DFUDS, the query time on pseudo-increasing arrays is up to 3 times slower than the query time on pseudo-decreasing arrays because of the depth of $2dmax(A)$. Note that the average distance between two matching parenthesis in DFUDS decreases proportional to the depth of $2dmax(A)$). The space usage of our data structure is not much affected by $\delta$ (up to 4.03 bpe to 4.39 bpe) since we do not maintain the arrays $E$ and $E'$ for all the cases (the depth of $2dmax(A)$ is still less than $\log 10^9 \sim 30$ even for large $\delta = 10^8$). Also, the query time for our data structure increases with $\delta$ because the average depth of the nodes corresponding to RMQ is increases with $\delta$. Overall, our data structure shows better time-space trade-off (takes up to 7.5 times less space while spending up to 4.2 times slower the query time) than all other indexing data structures in the evaluation.

Finally, for $\delta = 10^3$ and $10^6$, we evaluate the query time for pseudo-increasing and pseudo-decreasing arrays of size $n = 10^9$ for various query ranges (see Figure 8 and 9). Again, DFUDS-based implementations (R2MQ-ENCODING and FH-DFUDS) highly depend on the depth of $2dmax(A)$ and query ranges because of findopen operation, whereas the BP-based implementations (FN-BP and BGHL-BP) have similar results compared to the random array

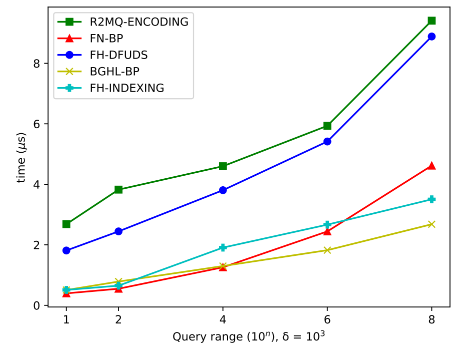


**(a)**

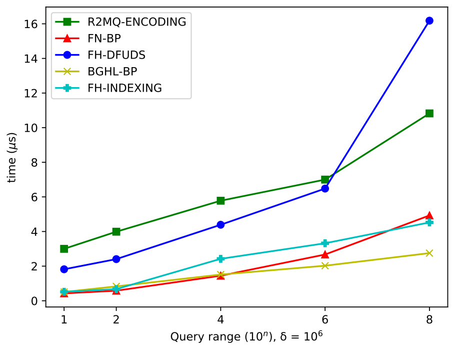


**(b)**

**Figure 8** Query time on the pseudo-increasing array with (a) $\delta = 10^3$, and (b) $delta = 10^6$.



**(a)**



**(b)**

**Figure 9** (Query time on the pseudo-decreasing array with (a) $\delta = 10^3$, and (b) $delta = 10^6$.

case. Especially compared to the random array case, our data structure supports much faster (up to 2.2 times) queries on pseudo-increasing and decreasing arrays for most query ranges since the extra overhead for accessing $E$ and $E'$ does not occur for both cases.

## 5 Conclusion

In this paper, we propose a practical implementation of an encoding for answering RT2Q queries. Our data structure takes much less space than the current indexing data structure implementations, while still giving better time-space trade-off for most cases in practice. An interesting open problem is to implement the data structure based on the BP of $2dmax(A)$ – here, an efficient and practical implementation of degree and childrank queries would be a challenging problem.

## References

1 Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proceedings of ALENEX 2010*, pages 84–97, 2010.

2 Niklas Baumstark, Simon Gog, Tobias Heuer, and Julian Labeit. Practical range minimum queries revisited. In *SEA 2017*, pages 12:1–12:16, 2017.

3 David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

4 Pooya Davoodi, Gonzalo Navarro, Rajeev Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A*, 372(2016):20130131, 2014.

5 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. On succinct representations of binary trees. *Math. Comput. Sci.*, 11(2):177–189, 2017.

6 Luc Devroye. On random cartesian trees. *Random Struct. Algorithms*, 5(2):305–328, 1994.

7 A. Farzan and J. I. Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, January 2014.

8 Héctor Ferrada and Gonzalo Navarro. Improved range minimum queries. *J. Discrete Algorithms*, 43:72–80, 2017.

9 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.

10 Harold N. Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of STOC 1984*, pages 135–143, 1984.

11 Pawel Gawrychowski and Patrick K. Nicholson. Optimal encodings for range top-k, selection, and min-max. In *ICALP 2015, Proceedings, Part I*, pages 593–604, 2015.

12 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.

13 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS 1989*, pages 549–554, 1989.

14 Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.*, 78(2):619–631, 2012.

15 J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

16 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

17 Rajeev Raman. Encoding data structures. In *WALCOM 2015*, pages 1–7, 2015.

18 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding $k$-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

20 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

# On Computing the Diameter of (Weighted) Link Streams

**Marco Calamai**
University of Florence, Italy

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

**Andrea Marino** ✉
University of Florence, Italy

## Abstract

A weighted link stream is a pair $(V, \mathbb{E})$ comprising $V$, the set of nodes, and $\mathbb{E}$, the list of temporal edges $(u, v, t, \lambda)$, where $u, v$ are two nodes in $V$, $t$ is the starting time of the temporal edge, and $\lambda$ is its travel time. By making use of this model, different notions of diameter can be defined, which refer to the following distances: earliest arrival time, latest departure time, fastest time, and shortest time. After proving that any of these diameters cannot be computed in time sub-quadratic with respect to the number of temporal edges, we propose different algorithms (inspired by the approach used for computing the diameter of graphs) which allow us to compute, in practice very efficiently, the diameter of quite large real-world weighted link stream for several definitions of the diameter. Indeed, all the proposed algorithms require very often a very low number of single source (or target) best path computations. We verify the effectiveness of our approach by means of an extensive set of experiments on real-world link streams. We also experimentally prove that the temporal version of the well-known 2-sweep technique, for computing a lower bound on the diameter of a graph, is quite effective in the case of weighted link stream, by returning very often tight bounds.

**2012 ACM Subject Classification** Theory of computation → Shortest paths

**Keywords and phrases** Temporal graph, shortest path, diameter

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.11

**Supplementary Material** *Software (Source Code)*: `https://github.com/marcocalamai/Link-stream-diameter` archived at `swh:1:dir:b33c6ae74c0739836c1874ca3fc7ac30f3893be9`

**Funding** Partially supported by MIUR under PRIN Project n. 20174LF3T8 AHeAD (Efficient Algorithms for HArnessing Networked Data), and by the University of Florence under Project GRANTED (GRaph Algorithms for Networked TEmporal Data).

**Acknowledgements** We thank Filippo Brunelli and Laurent Viennot for several discussions concerning the complexity of computing single source (target) best paths in temporal graphs. We also thank Roberto Grossi for kindly providing us the computing platform.

## 1 Introduction

**Link stream, distance, eccentricity, and diameter.** A time-dependent network is a graph $G = (V, E)$ in which each edge $e \in E$ has associated an arrival function specifying, for each starting time, the corresponding arrival time (see [4] for a classification of different types of time-dependent networks). A weighted *link stream* [14] (also called temporal graph in [22, 23] and point-availability time-dependent network in [4]) is a time-dependent network in which the domain of the arrival functions is a finite set $T$ of time instants (in this paper, we will assume that $T$ is a set of integer numbers). A weighted link stream is commonly represented as a list $\mathbb{E}$ of *temporal edges* $(u, v, t, \lambda)$, where $u$ and $v$ are two nodes in $V$, $t \in T$ is the *starting* time of the edge, and $\lambda$ denotes the *travel* time of the edge: according to this representation, the arrival time of the edge $(u, v)$ at time $t$ is equal to $t + \lambda$. In this paper, we refer to this representation by assuming that the list is sorted either in non-decreasing or in non-increasing order with respect to the edge starting times: in the former case we will denote the list as $\overrightarrow{\mathbb{E}}$, while in the latter case we will denote the list as $\overleftarrow{\mathbb{E}}$. We will also assume that the travel time of any temporal edge is an integer *positive* value. Moreover, we will assume that if the link stream is *undirected*, then a temporal edge $(u, v, t, \lambda) \in \mathbb{E}$ implicitly implies that the temporal edge $(v, u, t, \lambda)$ is also in the link stream. Finally, we will say that the link stream is *unweighted* if the travel time of all temporal edges is equal to 1.

When dealing with weighted link streams, the definition of path asks to satisfy, besides the typical constraints of a path in a graph, some natural time constraints. In particular, a *path* from a node $u$ to a node $v$ is a sequence of temporal edges

$$(u = w_1, w_2, t_1, \lambda_1), (w_2, w_3, t_2, \lambda_2), \ldots, (w_{k-1}, w_k, t_{k-1}, \lambda_{k-1}), (w_k, w_{k+1} = v, t_k, \lambda_k)$$

such that, for each $i$ with $1 < i \leq k$, $t_i \geq t_{i-1} + \lambda_{i-1}$. The *departure time* of the path is $t_1$, its *arrival time* is $t_k + \lambda_k$, its *duration* is $t_k + \lambda_k - t_1$, and its *travel time* is $\sum_{i=1}^{k} \lambda_i$. In the following, for any time interval $[t_\alpha, t_\omega]$, we will say that the path is $[t_\alpha, t_\omega]$-compatible, if its departure time is no earlier than $t_\alpha$ and its arrival time is no later than $t_\omega$.

By making use of the above four path cost functions, we can then define the following four corresponding distances between two nodes $u$ and $v$, in a specific time interval $[t_\alpha, t_\omega]$ (for all distances, we assume that their value is $\infty$, if there is no $[t_\alpha, t_\omega]$-compatible path) [24, 22, 23].

**Earliest arrival time** $d_{\text{EAT}}^{[t_\alpha, t_\omega]}(u, v)$ is the minimum arrival time of any $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$ minus $t_\alpha$.

**Latest departure time** $d_{\text{LDT}}^{[t_\alpha, t_\omega]}(u, v)$ is $t_\omega$ minus the maximum departure time of any $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$.

**Fastest time** $d_{\text{FT}}^{[t_\alpha, t_\omega]}(u, v)$ is the minimum duration time of any $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$.

**Shortest time** $d_{\text{ST}}^{[t_\alpha, t_\omega]}(u, v)$ is the minimum travel time of any $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$.

In the following, for the sake of simplicity, we will almost always avoid to specify the time interval in the superscript, and assume that it is the interval in which $t_\alpha$ is the minimum departure time of all temporal edges, and $t_\omega$ is the maximum arrival time.

Once a notion of distance is adopted, the corresponding notions of eccentricity and of diameter can be introduced, analogously to the case of standard graphs. That is, for any $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$, the (forward) *eccentricity* $\text{ECCF}_{\text{D}}(u)$ of a node $u$ is its maximum finite distance to any other node, and the *diameter* $\diameter_{\text{D}}$ of a weighted link stream is the maximum (defined) eccentricity of all its nodes.[1] *The goal of this paper is to analyse the problem of computing the diameter of a weighted link stream, in the case of the four previously defined distances.* To this aim, let us first note that several algorithms have been proposed in the literature in order to compute the distance of a source node $s$ to all other nodes, for each of the distance definitions considered in this paper. For any $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$, let $\texttt{ssbp}_{\text{D}}$ be an algorithm that, given in input $\overrightarrow{\mathbb{E}}$ and a source node $u$, returns an array containing, for each node $v$, the value $d_{\text{D}}(u, v)$, and let $\text{S-TIME}_{\text{D}}$ (respectively, $\text{S-SPACE}_{\text{D}}$) be the worst-case time (respectively, space) complexity of this algorithm, as a function of the number $n$ of nodes and of the number $m$ of temporal edges in the weighted link stream. In this paper, we

[1] In this paper, we use the symbol $\diameter$ since it is obtained by the command `diameter` in LaTeX, and since graphically it reminds the notion of a diameter.

**Table 1** The time and space complexity of the single source best path and the single target best path algorithms with input $\overrightarrow{\mathbb{E}}$ and $\overleftarrow{\mathbb{E}}$, respectively (without considering the time and space necessary for sorting the link stream).

| D | S-TIME$_{\text{D}}$ | S-SPACE$_{\text{D}}$ | Ref. | T-TIME$_{\text{D}}$ | T-SPACE$_{\text{D}}$ | Ref. |
|---|---|---|---|---|---|---|
| EAT | $O(m)$ | $O(n)$ | [22, 23] | $O(m)$ | $O(m)$ | [9, 10] |
| LDT | $O(m)$ | $O(m)$ | [9, 10] | $O(m)$ | $O(n)$ | [22, 23] |
| FT | $O(m)$ | $O(m)$ | [9, 22, 23] | $O(m)$ | $O(m)$ | [9, 22, 23] & Lemma 1 |
| ST | $O(m \log m)$ | $O(m)$ | [22, 23] | $O(m \log m)$ | $O(m)$ | [22, 23] & Lemma 1 |

mostly refer to the algorithms proposed in [9, 10, 22, 23]: as far as we know, they have the best time and space complexity (see the second and third column of Table 1). Nevertheless, our results can be easily adapted to other algorithms proposed in the literature, that usually have the same time and space complexity. In the following, we will assume that both $\overrightarrow{\mathbb{E}}$ and $\overleftarrow{\mathbb{E}}$ are available: otherwise, all time complexities of Table 1 become $O(m \log m)$, and all space complexities become $O(m)$.

**Computing the diameter.** For any $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$, in order to compute the corresponding diameter of a weighted link stream, we can execute the algorithm $\texttt{ssbp}_{\text{D}}$, for each source node $u$ (we will refer to such "text-book" approach as the algorithm $\text{TB}_{\text{D}}$). However, the time complexity of this approach is $O(n \cdot \text{S-TIME}_{\text{D}}(n, m))$: by looking at Table 1, we have that this time complexity is not affordable whenever we have thousands or millions of nodes, and millions or billions of temporal edges in the link stream. Unfortunately, *our first result consists in showing that it is very unlikely that there exists an algorithm computing any of the four diameters in time sub-quadratic in the number of temporal edges*. In other words, it is reasonable to conjecture that the known algorithms for computing any of the four diameters are, indeed, optimal.

In order to deal with this complexity conjecture, in this paper we propose to follow the approach that has been adopted while computing the diameter in real-world large graphs [7, 18, 8, 6, 19, 3] (and other distance-based measures like hyperbolicity [1]). This approach consists in *sorting the nodes* of the graph in a "clever" way, in computing, for each node in the given order, the eccentricity of the node, and in *updating a lower bound* on the value of the diameter and *an upper bound* on the value of the eccentricities of the remaining nodes, until the upper bound becomes less than or equal to the lower bound. The first main difficulty of this approach is, hence, to determine which order should be used in order to stop the process as soon as possible. For example, in the case of unweighted undirected connected graphs, the iFUB algorithm [6] first performs a breadth-first search starting from a random node $x$, and subsequently visits the nodes in the breadth-first search tree in a bottom-up fashion (this approach can be generalised to strongly connected directed graphs by executing both a forward and a backward breadth-first search starting from $x$, and by then combining paths entering $x$ with paths exiting $x$ [8]). The second main difficulty of this approach is finding a lower and an upper bound, which are easy to compute and to update, and that also allows the algorithm to stop as soon as possible. In the case of the iFUB algorithm, for instance, the lower bound is simply the maximum eccentricity computed so far, while the upper bound is a simple value connected to the level of the breadth-first search tree at which the algorithm is arrived.

In this paper, we show that this approach can also be applied to the computation of the diameter of a weighted link stream, for any of the four previously defined distances. To this aim, as in the case of directed graphs [8], we will make use of a "backward" variation of the single source best path algorithms. For any $\textsc{d} \in \{\textsc{eat}, \textsc{ldt}, \textsc{ft}, \textsc{st}\}$, let $\texttt{stbp}_{\textsc{d}}$ be an algorithm that, given in input $\overleftarrow{\mathbb{E}}$ and a target node $v$, returns an array containing, for each node $u$, the value $d_{\textsc{d}}(u, v)$, and let $\textsc{t-time}_{\textsc{d}}$ (respectively, $\textsc{t-space}_{\textsc{d}}$) be the worst-case time (respectively, space) complexity of this algorithm, as a function of the number $n$ of nodes and of the number $m$ of temporal edges in the weighted link stream. Once again, in this paper we mostly refer to the algorithms proposed in [9, 10, 22, 23]: as far as we know, they have the best time and space complexity (see the fifth and sixth column of Table 1). In the following, we will denote with $\textsc{eccb}_{\textsc{d}}(u)$ the *backward eccentricity* of a node $u$, that is its maximum finite distance from any other node: note that the diameter $\varnothing_{\textsc{d}}$ of a weighted link stream can also be defined as the maximum (defined) backward eccentricity of all its nodes.

## 1.1 Our results

**Computing $\varnothing_{\mathsf{eat}}$ and $\varnothing_{\mathsf{ldt}}$.** In the case of the earliest arrival time (respectively, latest departure time) distance, we propose to sort the nodes according to the maximum arrival (respectively, minimum departure) time of the edges entering (respectively, exiting) a node, in a non-increasing (respectively, non-decreasing) order. This choice is mostly inspired by the fact that, in the case of collaboration or citation link streams (such as the IMDB [11] and the DBLP-Citation [16, 20] networks), the earliest arrival time (respectively, latest departure time) diameter of the link stream coincides with the collaboration or citation of one of the last (respectively, first) nodes which entered the time-dependent network. Surprisingly, we will show that this intuition leads us, in practice, to a very efficient algorithm for computing the earliest arrival time (respectively, latest departure time) diameter of weighted link streams of different types (such as, public transport networks). For each analysed node, the algorithm executes a single target (respectively, source) best path computation, and update, in an appropriate way, both the lower and the upper bound: in particular, the lower bound is always the maximum distance seen so far, while the upper bound becomes the maximum arrival (respectively, minimum departure) time of the edges entering (respectively, exiting) the next node. As a result, our algorithm is able to compute the $\textsc{eat}$ and the $\textsc{ldt}$ diameter of large public transport networks using much lower visits with respect to the text book algorithm. This performance improvement is particularly significant for the three biggest public transport networks in our dataset, as the number of performed visits becomes smaller than 0.5% of the number of nodes.

**Computing $\varnothing_{\mathsf{ft}}$ and $\varnothing_{\mathsf{st}}$.** In the case of the fastest and the shortest time distance, the situation is more complicated. Indeed, aiming at adopting the approach used for weighted directed graphs by the iFUB-like algorithm, we need to guarantee that the paths leading to a node $x$, whose forward and backward distances to and from all other nodes have been computed, are temporally compatible with the paths exiting from $x$ and, that they can, hence, be "temporally" combined. Unfortunately, this is not always the case. This situation is similar to the one that arises when the iFUB approach is applied to weakly connected graphs: indeed, in this case either the diameter of the largest strongly connected component only is computed, or the component graph is computed in order to choose a pivot node for each strongly connected component, to bound the eccentricities of pivot nodes, and to propagate these bounds within each strongly connected component. The definition of (strongly) connected component in the case of link streams is somehow more involved (see [14]) and, as far as we know, no efficient algorithm capable of computing the analogue of the component graph has been proposed so far. For this reason, in this paper we have chosen to adopt the first solution, that is, restricting ourselves to the computation of what we have called the *pivot-diameter*. In our case, a pivot is a node at a given time instant, that is, a pair $(u, t)$, where $u \in V$ and $t \in T$. Given a set $P$ of pivots, the pivot-diameter of a weighted link stream with respect to $P$ is the diameter restricted to the set $R(P)$ of pair of nodes in $V$ such that, for any pair $(u, v) \in R(P)$, there is a path from $u$ to $v$ which passes through one the pivots in $P$ (our notion of pivot-diameter is connected to the notion of "pivotability" introduced in [5]). Note that, in some real-world weighted link streams, it should be possible to find a small pivot set such that almost all pairs of nodes are included in $R(P)$. For example, in the case of public transport networks, such a pivot set could be formed by the central station taken at different times of the day. In any case, once the set of pivots are given, we propose an algorithm that is able to compute the pivot-diameter of large link streams, improving very often the text-book algorithm of at least one order of magnitude. The worst case running time can be quadratic, but this seems to be unavoidable, as witnessed by the conditional lower bound we prove for this problem. It is also worth noting that, in the case of public transport networks, the pivot-diameter is, in many cases, very close to the diameter of the link stream, when the pivots are the top out-degree nodes, taken at few time instants.

**Computing lower bounds on the diameter.** Thanks to the backward and the forward visits discussed in Section 1.2 and whose complexities are summarized in Table 1, we are able to extend a well-known method for computing lower bounds for the diameter of static graphs to the case of temporal graphs. This method is called double-sweep and can be applied both to directed and undirected graphs: it selects a random node $r$ (sometimes chosen as a high degree node [8]), it performs a forward visit from $r$ obtaining the node $a_1$ which is the farthest from $r$, and set $lb_1 = \max_{b \in V} d(b, a_1)$. Then it performs a backward visit from $r$ obtaining the node $a_2$ which is the farthest from $r$, and set $lb_2 = \max_{b \in V} d(a_2, b)$. Finally, it returns the maximum between $lb_1$ and $lb_2$. This method naturally extends to the case of weighted link streams, for all the distances EAT, LDT, FT, ST, by using for each distance D the corresponding forward and backward best path search, i.e. using $\texttt{ssbp}_{\text{D}}$ and $\texttt{stbp}_{\text{D}}$. We analyse the performance of this revised temporal double sweep in our experiments for all these distances, and we show that the computed lower bound is very often tight, when performing $O(\log_2 n)$ double sweeps. In the case of ST, the lower bounds are often tight when dealing with public transport networks, while they seem to be rarely tight in the case of social networks. In any case, also in this latter case, double sweep significantly outperforms random sampling approaches.

## 1.2 Two useful link stream transformations

In [9], the authors introduce a simple transformation from a weighted linked stream $(V, \mathbb{E})$ to an unweighted link stream $(V \cup I, \mathbb{F})$, where $I$ is a set of at most $|\mathbb{E}|$ "intermediate" nodes. Intuitively, this transformation changes the travel time of a temporal edge into a waiting time in the intermediate nodes. More precisely, for each temporal edge $e = (u, v, t, 1) \in \mathbb{E}$, $e$ is also included in $\mathbb{F}$. For each temporal edge $e = (u, v, t, \lambda) \in \mathbb{E}$ with $\lambda > 1$, a new node $i_e$ is inserted in $I$, and the two temporal edges $(u, i_e, t, 1)$ and $(i_e, v, t + \lambda - 1, 1)$ are included in $\mathbb{F}$. It is easy to verify that, for any two nodes $u, v \in V$, $d_{\text{D}}(u, v)$ in $(V, \mathbb{E})$ is equal to $d_{\text{D}}(u, v)$ in $(V \cup I, \mathbb{F})$, where $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}\}$. This implies that the diameter of $(V, \mathbb{E})$ is equal to the maximum (defined) eccentricity of the nodes in $V$ computed in $(V \cup I, \mathbb{F})$.

The following lemma (whose proof is given in Appendix), instead, introduces another transformation which will allow us to easily design a backward version of a best path search and to relate the EAT distance and the LDT distance.

▶ **Lemma 1.** *Given a weighted link stream $(V, \mathbb{E})$, let $(V, \mathbb{F})$ be the weighted link stream obtained by substituting each temporal edge $e = (u, v, t, \lambda) \in \mathbb{E}$ with the temporal edge $\rho(e) = (v, u, -t-\lambda, \lambda)$. Then, for any two nodes $u, v \in V$, $d_D^{[t_\alpha, t_\omega]}(u, v)$ in $(V, \mathbb{E})$ is equal to $d_D^{[-t_\omega, -t_\alpha]}(v, u)$ in $(V, \mathbb{F})$, where $\textsc{d} \in \{\textsc{ft}, \textsc{st}\}$. Moreover, $d_{\text{EAT}}^{[t_\alpha, t_\omega]}(u, v)$ in $(V, \mathbb{E})$ is equal to $d_{\text{LDT}}^{[-t_\omega, -t_\alpha]}(v, u)$ in $(V, \mathbb{F})$, and $d_{\text{LDT}}^{[t_\alpha, t_\omega]}(u, v)$ in $(V, \mathbb{E})$ is equal to $d_{\text{EAT}}^{[-t_\omega, -t_\alpha]}(v, u)$ in $(V, \mathbb{F})$.*

In the rest of the paper, we are going to use the above two transformations for the following three purposes. First, we can ignore the LDT distance, since computing $\varnothing_{\text{LDT}}$ is equivalent to computing $\varnothing_{\text{EAT}}$. Second, we can obtain an algorithm $\texttt{stbp}_{\text{FT}}$ by applying the transformation $\rho$ of the lemma, by then applying the transformation in [9] to reduce to unitary weights, and by finally applying the algorithm in [22]. Third, we can obtain an algorithm $\texttt{stbp}_{\text{ST}}$ by applying the transformation $\rho$ of the lemma and by then using the algorithm $\texttt{ssbp}_{\text{ST}}$ described in [22].

## 2 Negative results

The *Strong Exponential Time Hypothesis* (in short, *SETH*) states that there is no algorithm for solving the $k$-Sat problem in time $O((2-\epsilon)^n)$, where $\epsilon > 0$ does not depend on $k$ [12]. This hypothesis has been repeatedly used in the last few years in order to prove the hardness of polynomial-time solvable problem (see, for example, [21], which is one of the first papers along this line of research, where the authors address the hardness of many problems, like computing all pairs shortest paths and finding triangles in a graph). We will here use it in order to prove that the diameter of an unweighted undirected link stream cannot be computed in time sub-quadratic with respect to the number of temporal edges.

To this aim, we will refer to the *k-Big Two Disjoint Set* (in short, *k-BTDS*) problem, which is defined as follows. Given a set $X$ and a collection $\mathcal{C}$ of subsets of $X$ such that $|X| \leq \log^k(|\mathcal{C}|)$, the solution is 1 if there are two disjoint sets $c, c' \in \mathcal{C}$, 0 otherwise. Clearly, this problem can be solved in quadratic time. It is also known that, for any $k$, the $k$-BTDS problem is not solvable in time $\tilde{O}(|\mathcal{C}|^{2-\epsilon})$, unless the SETH is false [2] (where the $\tilde{O}$ notation ignores poly-logarithmic factors).

▶ **Theorem 2.** *For any $\textsc{d} \in \{\textsc{eat}, \textsc{ft}, \textsc{st}\}$, computing the diameter $\varnothing_{\textsc{d}}$ of a linked stream $(V, \mathbb{E})$ cannot be done in time $\tilde{O}(|\mathbb{E}|^{2-\epsilon})$ for any $\epsilon > 0$, unless the SETH is false, even if the link stream is unweighted and undirected.*

**Proof.** We show that the $k$-BTDS problem is reducible (in quasi-linear time) to the link stream diameter computation problem, even in the case in which the diameter is equal either to 2 or to 3. Given an input $(X = \{x_1, \ldots, x_{|X|}\}, \mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\})$ of $k$-BTDS with $|X| \leq \log^k(|\mathcal{C}|)$, we construct an unweighted undirected link stream $(X \cup \mathcal{C}, \mathbb{E})$, where the set $\mathbb{E}$ of temporal edges is defined as follows (see also Figure 3 in Appendix).

- For each $x_i, x_j \in X$, $\mathbb{E}$ contains the two temporal edges $(x_i, x_j, 1, 1)$ and $(x_i, x_j, 2, 1)$.
- For each $c_j$ and for each $x_i \in c_j$, $\mathbb{E}$ contains the three temporal edges $(x_i, c_j, 1, 1)$, $(x_i, c_j, 2, 1)$, and $(x_i, c_j, 3, 1)$.

For any $\textsc{d} \in \{\textsc{eat}, \textsc{ft}, \textsc{st}\}$, let us now compute the eccentricities of all nodes, by distinguishing between nodes in $X$ and nodes in $\mathcal{C}$.

**Nodes in $X$** For each $x_i, x_j \in X$, $d_{\textsc{d}}(x_i, x_j) = 1$ (since we can use the temporal edge with starting time equal to 1). For each $x_i \in X$ and $c_j \in \mathcal{C}$, $d_{\textsc{d}}(x_i, c_j) = 1$ if $x_i \in c_j$ (since we can use the temporal edge with starting time equal to 1). Otherwise, $d_{\textsc{d}}(x_i, c_j) = 2$ (since we can first use the temporal edge from $x_i$ to $x_k$ with starting time equal to 1, for some $x_k \in c_j$, and then use the temporal edge from $x_k$ to $c_j$ with starting time equal to 2). Hence, for each $x_i \in X$, we have that $\textsc{eccf}_{\textsc{d}}(x_i) = 2$.

**Nodes in $\mathcal{C}$** For each $c_i \in \mathcal{C}$ and $x_j \in X$, we have already shown that $d_{\text{D}}(c_i, x_j) \leq 2$. For each other $c_j \in \mathcal{C}$, $d_{\text{D}}(c_i, c_j) = 2$ if $c_i \cap c_j \neq \emptyset$ (since we can first use the temporal edge from $c_i$ to $x_k$ with starting time equal to 1, for some $x_k \in c_i \cap c_j$, and then use the temporal edge from $x_k$ to $c_j$ with starting time equal to 2). Otherwise (that is, if $c_i \cap c_j = \emptyset$), we have that $d_{\text{D}}(c_i, c_j) \leq 3$ (since we can first use the temporal edge from $c_i$ to $x_k$ with starting time equal to 1, for some $x_k \in c_i$, then use the temporal edge from $x_k$ to $x_l$ with starting time equal to 2, for some $x_l \in c_j$, and finally use the temporal edge from $x_l$ to $c_j$ with starting time equal to 3). Note that, in this case, $d_{\text{D}}(c_i, c_j) = 3$ since there is no way of arriving in $c_j$ starting from $c_i$ before time 3, since no neighbor of $c_i$ is also a neighbor of $c_j$, and, hence, we are forced to pass through two nodes in $X$. Hence, for each $c_i \in \mathcal{C}$, $\text{ECCF}_{\text{D}}(c_i) = 3$ if there exists $c_j \in \mathcal{C}$ such that $c_i \cap c_j = \emptyset$, otherwise $\text{ECCF}_{\text{D}}(c_i) = 2$.

We can the conclude that the diameter $\varnothing_{\text{D}}$ of the link stream is either 2 or 3: it is 3 if and only if there exist two $c_i, c_j \in \mathcal{C}$ which are disjoint.

Since $|X| \leq \log^k(|\mathcal{C}|)$, the reduction can be executed in $\tilde{O}(|\mathcal{C}|)$ time, and $|\mathbb{E}| = \tilde{O}(|\mathcal{C}|)$. Hence, if we can compute the diameter of the link stream in $\tilde{O}(|\mathbb{E}|^{2-\epsilon})$ for some $\epsilon > 0$, then we could solve the $k$-BTDS in $\tilde{O}(|\mathcal{C}|^{2-\epsilon})$ for some $\epsilon > 0$. From the result of [2], it follows that the SETH would be falsified, and the theorem is proved. ◀

Note that, from the above theorem and from Lemma 1, it follows that the same result holds for the LDT distance. Moreover, the proof of the above theorem gives strong evidence that a sub-quadratic $(3/2 - \epsilon)$-approximation algorithm for the diameter may be very hard to find, even for undirected unweighted link streams.

## 3 Computing the EAT diameter

In this section we focus on the EAT distance, and we propose a quite simple algorithm for computing the diameter of a weighted link stream. As we already said in the introduction, the algorithm follows the approach used in the case of graphs, which consists in sorting the nodes of the link stream, in computing, for each node in the given order, the eccentricity of the node, and in updating a lower bound on the value of the diameter and an upper bound on the backward eccentricities of the remaining nodes, until the upper bound becomes less than or equal to the lower bound. In the case of the EAT distance, nodes are sorted with respect to their last "entering" time, i.e. $\delta(v) = \max_{t,\lambda:\exists(u,v,t,\lambda)\in\mathbb{E}}\{t+\lambda\} - t_\alpha$, the lower bound is the maximum backward eccentricity computed so far, and the upper bound is the difference between the entering time of the next node that has been examined and $t_\alpha$. This strategy is formalised in Algorithm 1.

▶ **Lemma 3.** *Let $v_1, \ldots, v_n$ be the ordering of the nodes in $V$ with respect to $\delta(\cdot)$. For each $i \geq 1$, $\max_{j=1}^{i} \text{ECCB}_{\text{EAT}}(v_j) \leq \varnothing_{\text{EAT}}$ and $\max_{j=i}^{n} \text{ECCB}_{\text{EAT}}(v_j) \leq \delta(v_i)$.*

**Proof.** The first inequality is obvious, since $\varnothing_{\text{EAT}} = \max_{i=1}^{n} \text{ECCB}_{\text{EAT}}(v_i)$. The second inequality follows from the fact that, for each $v_j \in V$, we have $\text{ECCB}_{\text{EAT}}(v_j) \leq \delta(v_j)$, and that, because of the ordering, for each $j \geq i$, we have $\delta(v_j) \leq \delta(v_i)$, which implies $\max_{j=i}^{n} \text{ECCB}_{\text{EAT}}(v_j) \leq \max_{j=i}^{n} \delta(v_j) \leq \delta(v_i)$. ◀

▶ **Theorem 4.** *Algorithm 1 correctly computes the EAT diameter.*

**Proof.** Because of the previous lemma, we have that, at the end of each iteration of the **while** loop, the value of $lb$ is a lower bound on $\varnothing_{\text{EAT}}$, while the value of $ub$ is an upper bound on the backward eccentricity of all the remaining nodes to be visited. Since $\varnothing_{\text{EAT}} =$

$\max_{i=1}^{n} \text{ECCB}_{\text{EAT}}(v_i)$, sooner or later the value of $lb$ has to become equal to $\varnothing_{\text{EAT}}$, and the value of $ub$ has to become less than or equal to $\varnothing_{\text{EAT}}$. When this happens, the loop stops, and the algorithm correctly returns the value of $lb$. ◀

**Algorithm 1** EAT diameter.

```
   Input  : Weighted link stream $(V, \overrightarrow{\mathbb{E}})$ with $n$ nodes
   Output : Diameter $\varnothing_{\text{EAT}}$
 1 foreach $v \in V$ do
 2     $\delta(v) \leftarrow \max_{t,\lambda:\exists(u,v,t,\lambda)\in\mathbb{E}}\{t+\lambda\} - t_\alpha$
 3 sort nodes $v_1, \ldots, v_n$ in non-increasing ordering with respect to $\delta(\cdot)$
 4 $lb \leftarrow 0$
 5 $i \leftarrow 1$
 6 $ub \leftarrow \delta(v_i)$
 7 while $ub > lb$ do
 8     $lb \leftarrow \max\{lb, \text{ECCB}_{\text{EAT}}(v_i)\}$
 9     $i \leftarrow i+1$ if $i \le n$ then $ub \leftarrow \delta(v_i)$
10 return $lb$
```

**Algorithm 2** Pivot-diameter.

```
   Input  : Weighted link stream $(V, \mathbb{E})$, set $P \subseteq V \times T$, distance $\text{D} \in \{\text{EAT}, \text{FT}, \text{ST}\}$
   Output : Pivot-diameter $\varnothing_{\text{D}}^{P}$
 1 $lb \leftarrow 0,\ ub \leftarrow \infty,\ \hat{A}_P \leftarrow \emptyset,\ \hat{B}_P \leftarrow \emptyset$
 2 while $ub > lb$ do
 3     $\text{M} \leftarrow \arg\max_{p\in P} \text{UB}_{\text{D}}(p)$
 4     $ub \leftarrow \text{UB}_{\text{D}}(\text{M})$
 5     if $ub = -\infty$ then return $lb$
 6     $(v,w) \leftarrow \arg\max_{z\in A_P\setminus\hat{A}_P, y\in B_P\setminus\hat{B}_P} \text{UB}_{\text{D}}(\text{M}, z, y)$
 7     $lb \leftarrow \max\{lb, \text{GETLOWERBOUND}(v,w,ub)\}$
 8 return $lb$

 9 Function GETLOWERBOUND$(v,w,ub)$
10     $y \leftarrow \text{ECCB}_{\text{D}}^{P}(w)$, Add $w$ to $\hat{B}_P$
11     if $y \ge ub \lor \text{D} = \text{EAT}$ then return $y$
12     $z \leftarrow \text{ECCF}_{\text{D}}^{P}(v)$, Add $v$ to $\hat{A}_P$
13     return $\max\{z, y\}$
```

Note that, by applying Lemma 1, Algorithm 1 can also be used to compute the LDT diameter. It should be clear, that, in the worst case, Algorithm 1 has to execute a backward best path search starting from each node of the link stream. That is, the worst-case time complexity of the algorithm is $O(n \cdot \text{T-TIME}_{\text{D}}(n,m))$. However, we will experimentally show that, in the case of real-world link streams, the number of searches that have to be performed is much lower than the number of nodes, thus making the algorithm extremely efficient.

## 4 Pivot-diameter

Let $(V, \mathbb{E})$ be a weighted link stream, and let $T$ be the set of the starting times of all temporal edges in $\mathbb{E}$. Given a subset $P$ of $V \times T$, let $R(P) \subseteq V \times V$ be the set of pairs defined as follows: $R(P) = \{(u,v) \mid \exists(x,t) \in P : d_{\text{EAT}}^{[t_\alpha,t]}(u,x) < \infty \land d_{\text{EAT}}^{[t,t_\omega]}(x,v) < \infty\}$. In other words, $P$ is a set of pivots, and $R(P)$ contains only pairs of nodes $(u,v)$ such that $u$ can reach $v$ passing through the node $x$ at time $t$, for some pivot $(x,t) \in P$. By restricting our attention to the set $R(P)$, we are sure that the pairs we are considering are connected by at least one path (in particular, a path passing through a pivot). However, while analyzing the diameter restricted to these pairs $(u,v)$, we will consider *all* the possible paths and not only the ones passing through the pivots. Formally, for any node $u$, let $A_P(u) = \{v : (v,u) \in R(P)\}$ and $B_P(u) = \{v : (u,v) \in R(P)\}$. Moreover, let $A_P = \{u : B_P(u) \neq \emptyset\}$ and $B_P = \{u : A_P(u) \neq \emptyset\}$. For any $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$,

- the *forward pivot-eccentricity* of a node $u \in A_P$ is $\text{ECCF}_{\text{D}}^{P}(u) = \max_{v\in B_P(u)} d_{\text{D}}(u,v)$, and
- the *backward pivot-eccentricity* of a node $u \in B_P$ is $\text{ECCB}_{\text{D}}^{P}(u) = \max_{v\in A_P(u)} d_{\text{D}}(v,u)$.

The *pivot-diameter* is then defined as $\varnothing_{\text{D}}^{P} = \max_{(u,v)\in R(P)} d_{\text{D}}(u,v) = \max_{u\in A_P} \text{ECCF}_{\text{D}}^{P}(u) = \max_{u\in B_P} \text{ECCB}_{\text{D}}^{P}(u)$. In this section, we focus on the pivot-diameter computation problem, that is, *given a weighted link stream and a set $P \subseteq V \times T$, compute the pivot-diameter with respect to any $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$* (note that thanks to Lemma 1, we will neglect the LDT distance, as this case can be reduced to the EAT distance case after a suitable transformation of the graph). Let us observe that the pivot-diameter computation problem is also hard to compute in time sub-quadratic to the number of temporal edges: indeed, the very same reduction described in the proof of Theorem 2 can be used, by choosing $P = X \times \{1, 2, 3\}$. In the following, we will assume that $T \subseteq \mathbb{N}^+$, that is, all time instants are positive integers (in order to deal with the negative time instants introduced by the transformation in Lemma 1, it is sufficient to perform a "temporal shift" of the link stream).

A simple algorithm for computing the pivot-diameter (denoted as $\text{PIVOT-TB}^P_{\text{D}}$) first computes the sets $A_P$ and $B_P$, by simply performing, for each pivot $p = (x, t) \in P$, a backward best path search in the interval $[t_\alpha, t]$ and a forward best path search in the interval $[t, t_\omega]$, both starting from $x$. Once computed $A_P$ and $B_P$, if $|A_P| \leq |B_P|$, then the algorithm computes, for each node $u$ in $A_P$, the value $\text{ECCF}^P_{\text{D}}(u)$, and returns the maximum among all such values. Otherwise (that is, $|B_P| < |A_P|$) the algorithm computes, for each node $u$ in $B_P$, the value $\text{ECCB}^P_{\text{D}}(u)$, and return the maximum among all such values.

We now propose another algorithm (called $\text{PIVOT-IFUB}_{\text{D}}$) which is, once again, inspired by the approach used for computing the diameter of graphs (see Algorithm 2). This algorithm, once computed the sets $A_P$ and $B_P$, sorts the nodes of the link stream, computes, for each node in the given order, the (forward or backward) pivot-eccentricity of the node, and updates a lower bound on the value of the pivot-diameter and an upper bound on the pivot-eccentricities of the remaining nodes, until the upper bound becomes less than or equal to the lower bound. Once again, a key ingredient of the algorithm is the order of the nodes, which must be able to guarantee an effective non-trivial upper bound. Moreover, this upper bound should be easily obtainable in order to do not burden the computation. In the case of $\text{PIVOT-IFUB}_{\text{D}}$, all these aspects are guided by the pivots. In particular, each pivot ensures an upper bound on the distance between pairs of nodes it connects, and, for all pairs of nodes $(v, w)$ in $R(P)$, we have at least one upper bound on their distance, which is given by the temporal paths passing through a pivot. Hence, we first select the pivot $\text{M}$ giving the worst upper bound (this upper bound becomes the new upper bound of the algorithm), and we then select the corresponding pair of nodes $(v, w)$. We then compute the forward pivot-eccentricity of $v$ and the backward pivot-eccentricity $w$, eventually improving the lower bound of the algorithm. It is worth noting that selecting the pivot $\text{M}$ and the pair $(v, w)$ can be done in a fast way, as we can, during a preprocessing phase of the algorithm, compute the distances from and to all the pivots in $P$ and, for each pivot $p \in P$, we can sort the nodes in $A_{\{p\}}$ and in $B_{\{p\}}$ in non-increasing order with respect to their distance from and to $p$, respectively.

The above description clearly depends on the distance $\text{D}$. In the following, we first precisely instantiate, for each distance, the above bounds, we then prove properties on the lower bound and on the upper bound, in order to asses the correctness of the algorithm. Given a pivot $p = (x, t) \in P$ and two nodes $u$ and $v$ such that $d^{[t_\alpha,t]}_{\text{EAT}}(u, x) < \infty$ and $d^{[t,t_\omega]}_{\text{EAT}}(x, v) < \infty$, we define $\text{UB}_{\text{EAT}}(p, u, v) = d^{[t,t_\omega]}_{\text{EAT}}(x, v)$, $\text{UB}_{\text{FT}}(p, u, v) = d^{[t_\alpha,t]}_{\text{LDT}}(u, x) - t_\omega + d^{[t,t_\omega]}_{\text{EAT}}(x, u)$, and $\text{UB}_{\text{ST}}(p, u, v) = d^{[t_\alpha,t]}_{\text{ST}}(u, x) + d^{[t,t_\omega]}_{\text{ST}}(x, u)$. For any $\text{D} \in \{\text{EAT}, \text{FT}, \text{ST}\}$, $d_{\text{D}}(u, v) \leq \text{UB}_{\text{D}}(p, u, v)$. Indeed, $\text{UB}_{\text{D}}(p, u, v)$ is the "cost" of a path going from $u$ to $x$ and then from $x$ to $v$, where the cost is computed accordingly to the distance $\text{D}$. This concatenation of paths is a valid temporal path as the path arrives in $x$ at most at time $t$ and leaves from $x$ at least at time $t$. For completeness, if $d^{[t_\alpha,t]}_{\text{EAT}}(u, x) = \infty$ or $d^{[t,t_\omega]}_{\text{EAT}}(x, v) = \infty$, we define $\text{UB}_{\text{D}}(p, u, v) = -\infty$ (this maybe non-intuitive definition allows us to deal with the search of the maximum values in a more compact way). In Algorithm 2, $\hat{A}_P \subseteq A_P$ and $\hat{B}_P \subseteq B_P$ are the set of nodes for which a $\texttt{ssbp}_{\text{D}}$ and a $\texttt{stbp}_{\text{D}}$, respectively, has been executed. Moreover, in the algorithm $\text{UB}_{\text{D}}(p)$ denotes the value $\max_{z \in A_P \setminus \hat{A}_P, y \in B_P \setminus \hat{B}_P} \text{UB}(p, z, y)$.

▶ **Lemma 5.** *At any iteration of the* ***while*** *loop of Algorithm 2, if* $\text{D} \in \{\text{FT}, \text{ST}\}$*, then* $lb = \max\{\max_{v \in \hat{B}_P}\{ECCB^P_{D}(v)\}, \max_{v \in \hat{A}_P}\{ECCF^P_{D}(v)\}\}$*, otherwise* $lb = \max_{v \in \hat{B}_P}\{ECCB^P_{EAT}(v)\}$*.*

**Proof.** It immediately follows from the definition of the GETLOWERBOUND function. ◀

▶ **Lemma 6.** *Let* $\textsc{d} \in \{\textsc{ft}, \textsc{st}, \textsc{eat}\}$*. At any iteration of the* ***while*** *loop of Algorithm 2, for any node* $u \in A_P \setminus \hat{A}_P$ *and* $v \in B_P \setminus \hat{B}_P$*,* $d_{\textsc{d}}(u, v) \leq ub$*.*

**Proof.** We have already observed that, for any pivot $p \in P$, $d_{\textsc{d}}(u, v) \leq \textsc{ub}_{\textsc{d}}(p, u, v)$. Hence, $d_{\textsc{d}}(u, v) \leq \max_{p\in P} \textsc{ub}(p, u, v)$. Since $u \in A_P$ and $v \in B_P$, then there is at least one pivot $p = (x, t)$ such that $d^{[t_\alpha,t]}_{\textsc{eat}}(u, x) < \infty$ and $d^{[t,t_\omega]}_{\textsc{eat}}(x, v) < \infty$. This implies that $\max_{p\in P} \textsc{ub}(p, u, v) \neq -\infty$. Moreover, we have that

$$\max_{p\in P} \textsc{ub}(p, u, v) \leq \max_{p\in P} \max_{z\in A_P \setminus \hat{A}_P, y\in B_P \setminus \hat{B}_P} \textsc{ub}(p, z, y) = \max_{p\in P} \textsc{ub}(p) = ub,$$

where the inequality holds since $u \in A_P \setminus \hat{A}_P$ and $v \in B_P \setminus \hat{B}_P$, while the remaining two equalities follow from the definition of $\textsc{ub}(p)$ and from the assignment at Line 4 of Algorithm 2. The lemma thus follows. ◀

▶ **Lemma 7.** *At any iteration of the* ***while*** *loop of Algorithm 2, if* $\textsc{d} \in \{\textsc{ft}, \textsc{st}\}$*, then, for any node* $u \in B_P \setminus \hat{B}_P$ *(resp.* $A_P \setminus \hat{A}_P$*),* $\textsc{eccb}^P_{\textsc{d}}(u)$ *(resp.* $\textsc{eccf}^P_{\textsc{d}}(u)$*) is bounded by* $\max\{lb, ub\}$*. Otherwise, for any node* $u \in B_P \setminus \hat{B}_P$*,* $\textsc{eccb}^P_{\textsc{eat}}(u)$ *is bounded by* $\max\{lb, ub\}$*.*

**Proof.** Let us suppose there is a node $v \in B_P \setminus \hat{B}_P$ such that $\textsc{eccb}^P_{\textsc{d}}(v) > ub$, with $\textsc{d} \in \{\textsc{eat}, \textsc{ft}, \textsc{st}\}$. Let $u$ be the node in $A_P$ such that $d_{\textsc{d}}(u, v) = \textsc{eccb}^P_{\textsc{d}}(v)$. Note that $u$ must exist and it is such that $\textsc{eccf}^P_{\textsc{d}}(u) \geq \textsc{eccb}^P_{\textsc{d}}(v)$. If $u \in A_P \setminus \hat{A}_P$ (this is the only possible case when $\textsc{d} = \textsc{eat}$, since $\hat{A}_P = \emptyset$), then from Lemma 6 it follows that $d_{\textsc{d}}(u, v) \leq ub$, which is a contradiction. Otherwise (that is, $u \in \hat{A}_P$), from Lemma 5 it follows that $lb \geq \textsc{eccf}^P_{\textsc{d}}(u) \geq \textsc{eccb}^P_{\textsc{d}}(v)$. For $\textsc{d} \in \{\textsc{ft}, \textsc{st}\}$, the proof for nodes in $A_P \setminus \hat{A}_P$ is similar. The lemma is thus proved. ◀

▶ **Theorem 8.** *Algorithm 2 correctly computes the pivot-diameter.*

**Proof.** It immediately follows from Lemma 6 and Lemma 7. ◀

The proof of the next theorem is given in Appendix.

▶ **Theorem 9.** *Algorithm 2 computes the pivot-diameter in* $O(|A_P| \cdot \textsc{s-time}_{\textsc{d}}(n, m) + |B_P| \cdot \textsc{t-time}_{\textsc{d}}(n, m) + |P| \cdot (\textsc{s-time}_{\textsc{d}}(n, m) + \textsc{t-time}_{\textsc{d}}(n, m) + n \log n))$ *time, using space* $O(|P| \cdot n + \textsc{s-space}_{\textsc{d}}(n, m) + \textsc{t-space}_{\textsc{d}}(n, m))$*.*

As we will see in the next section, an effective choice of the cardinality of $P$ is logarithmic in the number of nodes: hence, in the worst case, the time complexity of our algorithm is the same as the one of the $\textsc{tb}_{\textsc{d}}$ algorithm (if poly-logarithmic factors are ignored). This time is clearly bounded by $O(n \cdot (\textsc{s-time}_{\textsc{d}}(n, m) + \textsc{t-time}_{\textsc{d}}(n, m)) + n \log^2 n)$, and, looking at the costs in Table 1, it does not contradict the computational lower bound.

## 5 Experimental Results

This section is devoted to show our experimental results for the different notions of distance we have considered. After introducing our experimental testbed, we organize the results as follows. We show the performance of the temporal double sweep described in the introduction. We then show the performance of our algorithm for computing $\varnothing_{\textsc{eat}}$ and $\varnothing_{\textsc{ldt}}$, which is described in Section 3. We finally focus on the pivot-diameter, whose algorithm has been described in Section 4.

**Table 2** Our dataset. The meaning of the columns is described in Section 5.

| Network name | $n$ | $m$ | $t_\omega$-$t_\alpha$ | $\mathcal{R}$ | $\varnothing_{\text{EAT}}$ | $\varnothing_{\text{LDT}}$ | $\varnothing_{\text{FT}}$ | $\varnothing_{\text{ST}}$ | Ref. |
|---|---|---|---|---|---|---|---|---|---|
| PUBLIC TRANSPORT NETWORKS | | | | | | | | | |
| KUOPIO | 549 | 30 574 | 73 920 | 216 630 | 60 120 | 73 260 | 53 940 | 36 240 | [13] |
| RENNES | 1 407 | 107 713 | 73 320 | 1 641 208 | 63 660 | 70 980 | 42 540 | 8 760 | [13] |
| GRENOBLE | 1 547 | 113 437 | 78 780 | 1 265 735 | 75 540 | 75 180 | 42 180 | 12 300 | [13] |
| VENICE | 1 874 | 113 933 | 89 160 | 2 354 707 | 79 080 | 85 020 | 67 020 | 8 040 | [13] |
| BELFAST | 1 917 | 121 195 | 67 920 | 3 040 354 | 66 360 | 66 900 | 61 560 | 9 360 | [13] |
| CANBERRA | 2 764 | 122 690 | 65 760 | 5 754 287 | 62 280 | 65 700 | 43 320 | 8 700 | [13] |
| TURKU | 1 850 | 131 684 | 75 625 | 2 966 925 | 65 615 | 75 545 | 53 225 | 10 795 | [13] |
| LUXEMBOURG | 1 367 | 178 052 | 72 780 | 1 818 761 | 62 160 | 72 720 | 50 820 | 4 920 | [13] |
| NANTES | 2 353 | 194 572 | 76 680 | 4 320 287 | 74 040 | 72 360 | 61 680 | 18 780 | [13] |
| DETROIT | 5 683 | 214 853 | 90 660 | 29 990 674 | 76 260 | 87 660 | 54 352 | 17 174 | [13] |
| TOULOUSE | 3 329 | 222 749 | 73 920 | 9 525 234 | 72 300 | 73 200 | 53 940 | 12 900 | [13] |
| PALERMO | 2 176 | 224 260 | 76 200 | 4 734 976 | 27 505 | 35 659 | 9 669 | 7 139 | [13] |
| BORDEAUX | 3 435 | 236 489 | 78 365 | 9 933 813 | 76 025 | 76 020 | 56 520 | 9 620 | [13] |
| WINNIPEG | 5 079 | 332 522 | 77 808 | 25 519 193 | 67 488 | 75 977 | 56 545 | 6 594 | [13] |
| BRISBANE | 9 645 | 386 175 | 76 860 | 70 598 864 | 74 220 | 75 060 | 64 320 | 17 700 | [13] |
| DUBLIN | 4 571 | 399 875 | 75 471 | 15 801 421 | 72 591 | 72 892 | 60 929 | 9 141 | [13] |
| ADELAIDE | 7 548 | 402 933 | 76 200 | 50 007 666 | 73 745 | 76 200 | 54 498 | 14 810 | [13] |
| LISBON | 7 073 | 525 114 | 87 424 | 14 065 989 | 82 088 | 84 578 | 68 582 | 12 434 | [13] |
| PRAGUE | 5 147 | 621 545 | 90 660 | 19 181 301 | 90 660 | 90 660 | 84 900 | 17 280 | [13] |
| HELSINKI | 6 986 | 664 507 | 87 960 | 41 724 039 | 86 520 | 85 320 | 72 720 | 27 720 | [13] |
| BERLIN | 4 601 | 1 019 012 | 91 200 | 20 931 583 | 83 880 | 90 300 | 79 920 | 7 260 | [13] |
| ROME | 7 869 | 1 049 202 | 90 382 | 56 395 585 | 80 648 | 89 677 | 72 017 | 11 470 | [13] |
| MELBOURNE | 19 493 | 1 089 555 | 86 400 | 294 973 451 | 79 260 | 82 500 | 61 260 | 34 595 | [13] |
| SYDNEY | 24 063 | 1 234 097 | 91 440 | 416 037 155 | 90 780 | 91 260 | 81 840 | 50 936 | [13] |
| PARIS | 11 950 | 1 807 200 | 81 660 | 90 427 303 | 80 340 | 79 620 | 71 460 | 19 560 | [13] |
| SOCIAL NETWORKS | | | | | | | | | |
| TOPOLOGY | 34 759 | 99 019 | 2 016 004 | 146 028 435 | 2 016 004 | 2 016 004 | 2 016 004 | 20 | [17] |
| ELEC | 7 119 | 103 675 | 119 088 241 | 5 736 331 | 119 084 221 | 119 088 241 | 119 038 621 | 10 | [17] |
| FACEBOOK-WOSN-WALL | 46 953 | 876 020 | 137 462 861 | 617 698 414 | 137 461 023 | 132 709 413 | 132 527 766 | 39 | [17] |
| COLLEGE | 1 900 | 59 835 | 16 736 182 | 1 794 245 | 16 736 043 | 16 621 304 | 16 113 324 | 17 | [15] |
| Sx-mathoverflow-a2q | 88 577 | 107 581 | 203 068 634 | 47 765 186 | 203 068 634 | 203 068 634 | 202 876 639 | 18 | [15] |
| Sx-mathoverflow-c2a | 88 577 | 195 330 | 203 055 529 | 33 076 882 | 203 055 529 | 203 055 529 | 202 094 959 | 20 | [15] |
| Sx-mathoverflow-c2q | 88 581 | 203 639 | 202 990 935 | 24 548 566 | 202 983 123 | 202 778 885 | 201 772 254 | 17 | [15] |
| Email-Eu-core | 1 005 | 332 334 | 69 459 255 | 770 833 | 69 430 695 | 69 439 852 | 44 641 379 | 15 | [15] |
| Sx-askubuntu-a2q | 515 274 | 280 102 | 225 833 890 | *210 462 953 | 225 833 890 | 225 833 890 | ≥208 943 421 | ≥19 | [15] |
| Sx-askubuntu-c2q | 515 281 | 327 513 | 176 894 703 | *56 918 856 | 176 894 645 | 176 893 633 | ≥176 893 575 | ≥18 | [15] |
| Sx-askubuntu-c2a | 515 255 | 356 822 | 208 942 104 | *410 760 018 | 208 939 593 | 208 940 523 | ≥208 401 245 | ≥18 | [15] |
| Sx-superuser-a2q | 567 309 | 430 033 | 239 613 340 | *305 607 123 | 239 613 340 | 239 613 340 | ≥237 851 331 | ≥25 | [15] |
| Sx-mathoverflow | 88 581 | 506 550 | 203 069 368 | *191 433 809 | 203 068 737 | 202 999 190 | ≥202 991 055 | ≥20 | [15] |
| Sx-superuser-c2q | 567 316 | 479 067 | 239 293 899 | *83 514 878 | 239 293 899 | 239 293 899 | ≥225 626 623 | ≥22 | [15] |
| Sx-superuser-c2a | 567 301 | 534 239 | 236 358 777 | *967 867 967 | 236 354 503 | 236 358 777 | ≥235 553 110 | ≥20 | [15] |
| Sx-askubuntu | 515 281 | 964 437 | 225 834 463 | *4 098 469 692 | 225 834 443 | 223 600 507 | ≥223 599 719 | ≥23 | [15] |
| Sx-superuser | 567 316 | 1 443 339 | 239 614 929 | *7 720 583 649 | 239 614 929 | 239 614 929 | ≥239 425 973 | ≥25 | [15] |
| Wiki-talk-temporal | 1 140 149 | 7 833 140 | 200 483 883 | *80 939 303 499 | 200 483 883 | 196 754 851 | ≥186 398 389 | ≥24 | [15] |

**Computing Platform and Source Code.** Our computing platform is a machine with Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40 GHz, 24 virtual cores, 128 GB RAM, and running Ubuntu Linux version 4.4.0-22-generic (machine available at Dipartimento di Informatica, Università di Pisa, Italy). The code has been written in Python 3 and it is available at `github.com/marcocalamai/Link-stream-diameter`.

**Dataset.** Table 2 reports the set of link streams we have used for our experiments. The upper part refers to public transport networks, while the lower part refers to social networks. The former link streams are weighted while the latter are unweighted. We report the number of nodes $n$, the number of temporal edges $m$, the spectrum $t_\omega - t_\alpha$ of times where edges appear, and the diameter $\varnothing_{\text{D}}$ for $\text{D} \in \{\text{EAT}, \text{LDT}, \text{FT}, \text{ST}\}$. We also report $\mathcal{R}$, which is the number of pairs $(u, v)$ such that $d_{\text{D}}(u, v) < \infty$, as this quantity, together with the diameter values will be compared to our results concerning the pivot-diameter. The values of $\varnothing_{\text{D}}$ have been computed by making use of $\text{TB}_{\text{D}}$: whenever, in the table, we write $\varnothing_{\text{D}} \geq y$, it means that we are reporting as $y$ the best lower bound computed so far during our experiments.

For computing $\mathcal{R}$, we have used $\text{TB}_{\text{EAT}}$, when possible. In the cases marked with $^\star$, the value of $\mathcal{R}$ has been estimated using the method in [9].

**Table 3** Number of times the lower bound returned by $2\text{SW}_{\text{D}}(k)$ and $\text{RS}_{\text{D}}(k)$ is tight.

| D | NETWORKS (Number of Nets whose $\varnothing_{\text{D}}$ is known) | Number of times the returned lower bound is tight | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $k = 4$ | | $k = 4\log_2 n$ | | $k = 8\log_2 n$ | | $k = 16\log_2 n$ | |
| | | $2\text{SW}_{\text{D}}(k)$ | $\text{RS}_{\text{D}}(k)$ | $2\text{SW}_{\text{D}}(k)$ | $\text{RS}_{\text{D}}(k)$ | $2\text{SW}_{\text{D}}(k)$ | $\text{RS}_{\text{D}}(k)$ | $2\text{SW}_{\text{D}}(k)$ | $\text{RS}_{\text{D}}(k)$ |
| EAT | PUBLIC TRANSPORT (25) | 10 | 3 | 15 | 6 | 19 | 9 | 22 | 10 |
| LDT | PUBLIC TRANSPORT (25) | 10 | 0 | 19 | 1 | 23 | 1 | 23 | 2 |
| FT | PUBLIC TRANSPORT (25) | 4 | 0 | 18 | 0 | 19 | 1 | 21 | 2 |
| | SOCIAL (8) | 1 | 0 | 6 | 0 | 7 | 0 | 7 | 0 |
| ST | PUBLIC TRANSPORT (25) | 13 | 0 | 22 | 1 | 23 | 2 | 24 | 3 |
| | SOCIAL (8) | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

As it can be seen and as already pointed out in the introduction, in the case of social networks, $\varnothing_{\text{EAT}}$ and $\varnothing_{\text{LDT}}$ are both very close to $t_\omega - t_\alpha$. This is expected because of the meaning of the link streams in the case of this kind of networks, whose behaviour is induced by new users being added. This fact makes the computation of $\varnothing_{\text{EAT}}$ and $\varnothing_{\text{LDT}}$ very easy: for this reason, we decided to exclude social networks when reporting our experimental results concerning $\varnothing_{\text{EAT}}$ and $\varnothing_{\text{LDT}}$.

**Methods.** In the following, we summarize our methods and competitors used in the remainder of the section. The subscript D refers to distances in $\{\text{EAT}, \text{LDT}\, \text{FT}, \text{ST}\}$.

**Lower bounds for the diameter** The following methods compute *lower bounds* for $\varnothing_{\text{D}}$.

- $2\text{SW}_{\text{D}}(k)$: select a set of $k/4$ nodes randomly chosen and return the best lower bound found by a double sweep, which have been described in the introduction. As each double sweep requires 4 visits, $2\text{SW}_{\text{D}}(k)$ performs exactly $k$ visits in total.
- $\text{RS}_{\text{D}}(k)$: select a set of $k$ nodes randomly chosen $v_1, \ldots, v_k$, and return $\max_i \text{ECCF}_{\text{D}}(v_i)$. Also this methods requires exactly $k$ visits in total.

**Computing exactly the diameter** The following methods compute $\varnothing_{\text{EAT}}$ and $\varnothing_{\text{LDT}}$ *exactly*.

- EAT-ALG: apply Algorithm 1, described in Section 3.
- LDT-ALG: apply the transformation in Lemma 1 and then Algorithm 1.
- $\text{TB}_{\text{D}}$: for each node $v \in V$, compute $\text{ECCF}_{\text{D}}(v)$ and return the maximum value found.

**Computing exactly the pivot-diameter** The following methods, given a set of pivots $P \subseteq V \times T$, compute the pivot-diameter introduced in Section 4.

- $\text{PIVOT-IFUB}_{\text{D}}$: apply Algorithm 2.
- $\text{PIVOT-TB}_{\text{D}}$: this algorithm has been described in Section 4.

In order to evaluate the considered methods independently from the used platform, their performance have been expressed in terms of number of visits. For the sake of completeness, however, a rough estimation of the running time in seconds (on our computing platform) can be easily obtained using the running times of the visits reported in Table 8.

## Computing lower bounds

This section is devoted to show the performance of $2\text{SW}_{\text{D}}(k)$ compared to the one of $\text{RS}_{\text{D}}(k)$, for different values of $k$. We evaluate the performance of both methods for $k \in \{4, 4\log_2 n, 8\log_2 n, 16\log_2 n\}$ and we summarize our results in Table 3 and in Table 6 in Appendix. Table 3 reports the number of times the lower bound returned by each method is tight, i.e. it is equal to $\varnothing_{\text{D}}$. We were able to do this only for the link streams whose diameter is known, i.e. the 25 public transport networks and 8 social networks. Looking at Table 3, we can see that, as expected, the performance of both algorithms improves by increasing $k$, i.e. the number of performed visits.

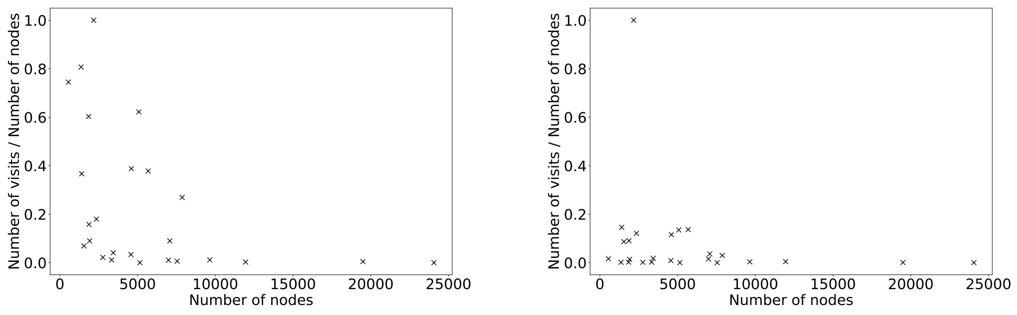
For EAT and LDT we report the results only for the 25 public transport networks in our dataset as we have seen that in the case of social networks the diameter is easy to find. For the public transport networks, $2\text{SW}_{\text{EAT}}(k)$ (resp. $2\text{SW}_{\text{LDT}}(k)$) is able to get very often tight lower bounds, especially with $k = 16\log_2 n$. For this value of $k$, in only 3 cases (resp. 2 cases) $2\text{SW}_{\text{EAT}}(k)$ (resp. $2\text{SW}_{\text{LDT}}(k)$) was not able to find a tight lower bound for $\varnothing_{\text{EAT}}$ (resp. $\varnothing_{\text{LDT}}$), namely for KUOPIO, MELBOURNE, and PARIS (resp. TOULOUSE and SYDNEY). Also in the case of FT, we can clearly see a very good performance of $2\text{SW}_{\text{FT}}(k)$. In the case $k = 16\log_2 n$, it is able to find 21 tight lower bounds over the 25 public transport networks, and 7 tight lower bounds over the 8 social networks. The exceptional social network is EMAIL-EU-CORE, in which $2\text{SW}_{\text{FT}}(k)$ returns a lower bound 4% lower than the real value of the diameter.

In the case of $2\text{SW}_{\text{ST}}(k)$ we confirm the good performance of the double sweep in the case of public transport networks. However, we observe that its behaviour is worse in the case of social networks, as in only one case over 8, it returns a tight lower bound. In any case, we have verified that in the great majority of the cases, $2\text{SW}_{\text{ST}}(k)$ returns a lower bound greater than or equal to the one obtained by $\text{RS}_{\text{ST}}(k)$. This can be seen in Table 6 in Appendix, where we report such number of cases for each distance D. In this case, we were able to include all our 18 social networks, as we do not need to known the exact value of the diameter to perform the comparison. All in all, we can see that, fixing the number of visits, it is always more convenient to run $2\text{SW}_{\text{ST}}(k)$ instead of $\text{RS}_{\text{ST}}(k)$.

## Computing $\varnothing_{\text{eat}}$ and $\varnothing_{\text{ldt}}$

In this section, we discuss the performance of EAT-ALG and LDT-ALG, i.e. the algorithms discussed in Section 3. These methods respectively compute $\varnothing_{\text{EAT}}$ and $\varnothing_{\text{LDT}}$, and have been compared respectively to $\text{TB}_{\text{EAT}}$ and $\text{TB}_{\text{LDT}}$. The comparison is done in terms of number of visits performed: in particular, we report the ratio between the number of visits performed by our methods and $n$, which is indeed the number of visits required by $\text{TB}_{\text{D}}$. We report the results only for public transport networks.

Our results are summarized in Figure 1a, and further detailed in Table 7 in Appendix. In the case of EAT-ALG we can see that in 14 cases over 25 it performs less than 10% of the visits performed by $\text{TB}_{\text{EAT}}$, while in 20 cases over 25 it performs less than 50% of the visits. The cases where EAT-ALG had worst performance were KUOPIO, TURKU, LUXEMBOURG, WINNIPEG, and PALERMO. On the other hand, the cases where EAT-ALG performs better correspond to the three biggest link streams, namely MELBOURNE, SYDNEY, and PARIS, where it performs less than 1% visits. These results are even better if we look at the performance of LDT-ALG in Figure 1b. In this case we perform less than 15% of the visits required by $\text{TB}_{\text{LDT}}$ for all the link streams except for PALERMO. In 16 cases over 25, we perform less than 3% of the visits. The reason behind these performances are deeply related to how the EAT-ALG and LDT-ALG work. Starting from the nodes with biggest $\delta(v)$, they perform one visit after the other, stopping when processing a node such that $\delta(w) = \varnothing_{\text{EAT}}$ (resp. $\delta(w) = \varnothing_{\text{LDT}}$). For this reason, both the methods need to compute the eccentricity of all the nodes $v$ having $\delta(v) > \varnothing_{\text{D}}$. These nodes are relatively few in general, as shown in the case of ROME in Figures 4a and 4b in Appendix, respectively for EAT and LDT. In particular, Figure 4a and Figure 4b show for each $\delta$ the number of nodes in the link stream ROME having such $\delta$. The values of $\delta$ are different for the plots, as the ones of Figure 4b refer to the values in the graph transformed applying Lemma 1. In both the cases, starting from the right of each plot, we need to perform the visit from all the nodes whose $\delta$ is at the right of the arrow marked as "diameter", whose number can be visually observed by the black mass at the right of the arrow. In the case of PALERMO, we have verified that the diameter is very low and all the nodes are at the right of the arrow both for EAT and LDT (see Figures 4c and 4d in Appendix).



**(a)** Visits performed by EAT-ALG wrt to $n$. **(b)** Visits performed by LDT-ALG wrt to $n$.

**Figure 1** Ratio between the number of visits performed by EAT-ALG (resp. LDT-ALG) and $n$, where $n$ is the number of visits required by $\text{TB}_{\text{EAT}}$ (resp. $\text{TB}_{\text{LDT}}$), as a function of the number of nodes.

## Computing the pivot-diameter

In the following, we focus on the computation of the pivot-diameter, using our algorithm $\text{PIVOT-IFUB}_{\text{D}}$ and the text-book algorithm $\text{PIVOT-TB}_{\text{D}}$. For the sake of brevity, in the following we discuss only the case where $\text{D} \in \{\text{FT}, \text{ST}\}$. For public transport networks, we have also computed the pivot-diameter for $\text{D} \in \{\text{LDT}, \text{EAT}\}$: we just report the results in Table 4.

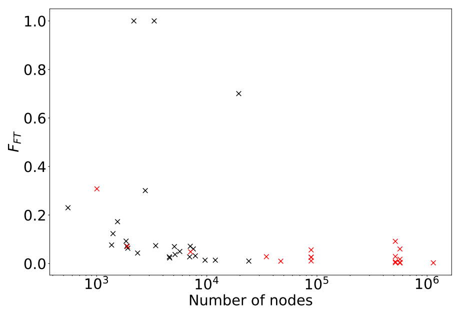
**Choice of the pivots.** We report our experiments in the case in which pivots are chosen as follows (we have analysed the performance of $\text{PIVOT-IFUB}_{\text{D}}$ with several choices of the pivots, with similar performance results: the choice we show here is a choice leading a sufficiently large coverage of the number of pairs analysed, corresponding to the case in which the pivot-diameter is more likely to be harder to compute). The set $P$ is defined as $V' \times T'$, where $V'$ are the top-$\log_2 n$ vertices with respect to the out-degree, i.e. number of temporal edges exiting from the vertex, and $T'$ are 4 times equally spaced in the interval $[t_\alpha, t_\omega]$, i.e. $T' = \{t : \ t = t_\alpha + i/5 \cdot (t_\omega - t_\alpha), i \in \{1, 2, 3, 4\}\}$. For each graph in our dataset, we report the ratio $|P(H)|/|\mathcal{R}|$ in the second column of Table 4 and 5 (respectively, for public transport networks and social networks), that is the ratio between the number of pairs considered by the pivot-diameter and the number of pairs considered by diameter. As it can be seen, in the case of public transport networks, this choice of the pivots lead to a coverage of the pairs almost always very high, namely, for almost all the link streams, we cover more than 93% of the pairs. The exceptions are TOULOUSE, LISBON, and MELBOURNE, which are for this reason highlighted with a grey row. We have verified that this poor coverage is due to the fact that in these networks the nodes with the highest number of exiting temporal edges have a small out-degree in the underlying directed graph (see the concluding remarks). We also report the values of $|A_P|$ and $|B_P|$, which correspond to the number of vertices reaching (and reached by) the pivots (see Section 4). These values are crucial as $\min\{|A_P|, |B_P|\}$ is the number of visits needed by $\text{PIVOT-TB}_{\text{D}}$ (we report the ratio $\min\{|A_P|, |B_P|\}/n$ in Table 4 and 5). In the cases in which $\min\{|A_P|, |B_P|\}$ is constant or relatively small in practice, when compared to the number of nodes (like in the case of TOULOUSE and MELBOURNE), both $\text{PIVOT-TB}_{\text{D}}$ and $\text{PIVOT-IFUB}_{\text{D}}$ are effective, as they both spend linear time to find the pivot-diameter (see Theorem 9). In the case of social networks, the ratio $|R(B)|/|\mathcal{R}|$ seems to be in general smaller as also $\min\{|A_P|, |B_P|\}/n$ is very often below 10%. Even if one could be tempted to run $\text{PIVOT-TB}_{\text{D}}$, we will see that this choice is not convenient in any case, as $\text{PIVOT-IFUB}_{\text{D}}$ will perform much less visits to discover the pivot-diameter.

**Pivot-diameter vs diameter.** In the case in which pivots are chosen as above, the pivot-diameter is often very close to the diameter of the link stream (when we could verify it).

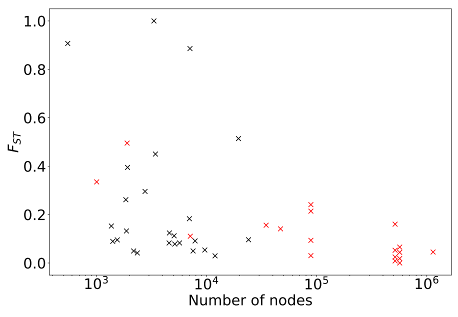
**Table 4** Results for the pivot-diameter in the case of public transport networks (the choice of $P$ is explained in the text). Rows in shadow gray correspond to graphs where the selected pivots lead to a small ratio $|R(P)|/|\mathcal{R}|$ (see also the concluding remarks).

| PUBLIC TRANSPORT NETWORKS | $\frac{\lvert R(P)\rvert}{\lvert\mathcal{R}\rvert}$ | $\lvert A_P\rvert$ | $\lvert B_P\rvert$ | $\frac{\min\{\lvert A_P\rvert,\lvert B_P\rvert\}}{n}$ | $\frac{\varnothing^P_{\text{EAT}}}{\varnothing_{\text{EAT}}}$ | $F_{\text{EAT}}$ | $\frac{\varnothing^P_{\text{LDT}}}{\varnothing_{\text{LDT}}}$ | $F_{\text{LDT}}$ | $\frac{\varnothing^P_{\text{FT}}}{\varnothing_{\text{FT}}}$ | $F_{\text{FT}}$ | $\frac{\varnothing^P_{\text{ST}}}{\varnothing_{\text{ST}}}$ | $F_{\text{ST}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| KUOPIO | 98.12% | 452 | 472 | 82.33% | 1 | 1 | 0.93 | 0.16 | 1 | 0.23 | 0.093 | 0.9 |
| RENNES | 98.51% | 1298 | 1308 | 92.25% | 1 | 0.23 | 0.94 | 0.07 | 0.91 | 0.12 | 0.97 | 0.09 |
| GRENOBLE | 93.56% | 1 194 | 1 232 | 77.18% | 0.72 | 0.3 | 0.94 | 0.11 | 1 | 0.17 | 1 | 0.1 |
| VENICE | 95.30% | 1 531 | 1 512 | 80.68% | 1 | 0.1 | 0.97 | 0.06 | 1 | 0.07 | 0.84 | 0.13 |
| BELFAST | 98.27% | 1 707 | 1 780 | 89.04% | 0.99 | 0.05 | 0.96 | 0.08 | 1 | 0.06 | 0.89 | 0.4 |
| CANBERRA | 98.59% | 2 307 | 2 469 | 83.46% | 0.99 | 0.05 | 0.98 | 0.04 | 1 | 0.3 | 1 | 0.3 |
| TURKU | 97.93% | 1 722 | 1 690 | 91.35% | 1 | 0.26 | 0.97 | 0.05 | 1 | 0.09 | 0.42 | 0.26 |
| LUXEMBOURG | 99.74% | 1 336 | 1 358 | 97.73% | 1 | 0.36 | 0.94 | 0.06 | 1 | 0.08 | 1 | 0.15 |
| NANTES | 97.14% | 2 170 | 2 201 | 92.22% | 0.91 | 0.05 | 0.98 | 0.04 | 1 | 0.04 | 1 | 0.04 |
| DETROIT | 99.04% | 5 527 | 5 434 | 95.62% | 0.98 | 0.38 | 0.95 | 0.1 | 1 | 0.05 | 1 | 0.08 |
| TOULOUSE | 0.00% | 20 | 20 | 0.60% | 0.41 | 1 | 0.07 | 1 | 0.03 | 1 | 0.13 | 1 |
| PALERMO | 100.00% | 2 176 | 2 176 | 100% | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0.05 |
| BORDEAUX | 98.50% | 3 166 | 3 119 | 90.80% | 0.9 | 0.44 | 0.89 | 0.86 | 1 | 0.07 | 0.97 | 0.45 |
| WINNIPEG | 97.81% | 5 054 | 4 946 | 97.38% | 1 | 0.4 | 1 | 0.02 | 1 | 0.07 | 1 | 0.11 |
| BRISBANE | 98.51% | 8 228 | 8 657 | 85.31% | 0.99 | 0.01 | 0.94 | 0.02 | 1 | 0.01 | 0.99 | 0.05 |
| DUBLIN | 99.15% | 4 007 | 3 918 | 85.71% | 0.97 | 0.03 | 0.97 | 0.03 | 1 | 0.03 | 1 | 0.08 |
| ADELAIDE | 98.82% | 7 148 | 6 956 | 92.16% | 0.9 | 0.47 | 0.93 | 0.06 | 1 | 0.06 | 0.90 | 0.05 |
| LISBON | 28.61% | 2 052 | 2 059 | 29.01% | 0.97 | 0.06 | 0.96 | 0.05 | 1 | 0.07 | 0.27 | 0.89 |
| PRAGUE | 98.27% | 4 366 | 4 412 | 84.83% | 0.99 | 0.02 | 0.99 | 0.02 | 1 | 0.04 | 0.96 | 0.08 |
| HELSINKI | 99.32% | 6 375 | 6 552 | 91.25% | 0.91 | 0.23 | 1 | 0.02 | 1 | 0.03 | 0.34 | 0.18 |
| BERLIN | 99.58% | 4 573 | 4 561 | 99.13% | 1 | 0.02 | 0.98 | 0.02 | 1 | 0.02 | 1 | 0.12 |
| ROME | 99.86% | 7 511 | 7 499 | 95.30% | 1 | 0.03 | 0.95 | 0.01 | 1 | 0.03 | 1 | 0.09 |
| MELBOURNE | 8.19% | 17 545 | 1 459 | 7.48% | 0.97 | 0.17 | 0.96 | 0.2 | 0.96 | 0.7 | 1 | 0.51 |
| SYDNEY | 93.94% | 19 933 | 21 269 | 82.84% | 0.97 | 0 | 0.99 | 0 | 1 | 0.01 | 0.53 | 0.1 |
| PARIS | 99.60% | 9 878 | 9 858 | 82.49% | 0.95 | 0.03 | 0.97 | 0.02 | 1 | 0.01 | 1 | 0.03 |

This is particularly evident for FT, while there are more exceptions for ST. To see this, for the cases in which we have the diameter of the link stream, in Table 4 and 5 we report, for each distance, the ratio between $\varnothing^P_{\text{D}}$ and $\varnothing_{\text{D}}$, where the former is computed by using our PIVOT-IFUB$_{\text{D}}$. In the case of public transport networks (Table 4), this can be easily explained by the fact that, very often, there is a large ratio $|R(P)|/|\mathcal{R}|$. In the case of social networks (Table 5), even though the ratio $|R(P)|/|\mathcal{R}|$ is lower, the ratio between $\varnothing^P_{\text{D}}$ and $\varnothing_{\text{D}}$ is not low, for FT and sometimes for ST.



**(a)** $F_{\text{FT}}$ as a function of the number of nodes.



**(b)** $F_{\text{ST}}$ as a function of the number of nodes.

**Figure 2** For each link stream of $n$ nodes, where the percentage of performed visits of PIVOT-IFUB$_{\text{D}}$ is $F_{\text{D}}$, we draw a cross (black for public transport networks and red for social networks) in position $(n, F_{\text{D}})$.

**Table 5** Results for the pivot-diameter in the case of public transport networks (the choice of $P$ is explained in the text). EAT and LDT distances are here neglected.

| SOCIAL NETWORKS | $\frac{\lvert R(P)\rvert}{\lvert\mathcal{R}\rvert}$ | $\lvert A_P\rvert$ | $\lvert B_P\rvert$ | $\frac{\min\{\lvert A_P\rvert,\lvert B_P\rvert\}}{n}$ | $\frac{\varnothing^P_{\text{FT}}}{\varnothing_{\text{FT}}}$ | $F_{\text{FT}}$ | $\frac{\varnothing^P_{\text{ST}}}{\varnothing_{\text{ST}}}$ | $F_{\text{ST}}$ |
|---|---|---|---|---|---|---|---|---|
| TOPOLOGY | 39.75% | 14 402 | 4 316 | 12.42% | 1 | 0.03 | 0.95 | 0.16 |
| ELEC | 63.86% | 3 576 | 2 169 | 30.47% | 1 | 0.05 | 0.9 | 0.11 |
| FACEBOOK-WOSN-WALL | 64.87% | 13 720 | 32 591 | 29.22% | 0.99 | 0.01 | 0.79 | 0.14 |
| COLLEGE | 80.52% | 1 268 | 1 354 | 66.74% | 1 | 0.07 | 0.71 | 0.5 |
| SX-MATHOVERFLOW-A2Q | 85.42% | 4 866 | 11 753 | 5.49% | 1 | 0.03 | 0.89 | 0.21 |
| SX-MATHOVERFLOW-C2A | 82.79% | 7 993 | 5 082 | 5.74% | 1 | 0.03 | 0.8 | 0.09 |
| SX-MATHOVERFLOW-C2Q | 86.40% | 2 355 | 12 927 | 2.66% | 1 | 0.06 | 0.88 | 0.24 |
| EMAIL-EU-CORE | 95.31% | 818 | 924 | 81.39% | 1 | 0.31 | 0.6 | 0.33 |
| SX-ASKUBUNTU-A2Q | 73.86% | 6 572 | 52 406 | 1.27% | | 0.03 | | 0.05 |
| SX-ASKUBUNTU-C2Q | 80.12% | 1 671 | 52 590 | 0.32% | | 0.09 | | 0.16 |
| SX-ASKUBUNTU-C2A | 72.49% | 27 345 | 20 994 | 4.07% | | 0.01 | | 0.02 |
| SX-SUPERUSER-A2Q | 76.27% | 8 669 | 54 668 | 1.53% | | 0.02 | | 0.02 |
| SX-MATHOVERFLOW | 87.79% | 11 941 | 19 441 | 13.48% | | 0.01 | | 0.03 |
| SX-SUPERUSER-C2Q | 89.36% | 2 543 | 61 294 | 0.45% | | 0.06 | | 0.07 |
| SX-SUPERUSER-C2A | 78.00% | 39 088 | 31 982 | 5.64% | | 0.004 | | 0.04 |
| SX-ASKUBUNTU | 75.10% | 44 245 | 114 766 | 8.59% | | 0.003 | | 0 |
| SX-SUPERUSER | 80.35% | 67 470 | 136 455 | 11.89% | | 0.002 | | 0.01 |
| WIKI-TALK-TEMPORAL | 54.93% | 53 408 | 1 037 170 | 4.68% | | 0.003 | | 0.04 |

**Performance of the algorithms.** In Figure 2, we report the ratio between the number of visits performed by PIVOT-IFUB$_{\text{D}}$ and $\min\{|A_P|, |B_P|\}$, which are the visits required by TB$_{\text{D}}$ (in the following, we denote this ratio as $F_{\text{D}}$). For each link stream of $n$ nodes, we draw a cross in position $(n, F_{\text{D}})$ (black for public transport networks and red for social networks). The plot on the left refers to FT, while the plot on the right refers to ST. For the sake of completeness, the values of $F_{\text{D}}$ are also reported in Table 4 and 5. As it can be seen, $F_{\text{FT}}$ and $F_{\text{ST}}$ indicates that PIVOT-IFUB$_{\text{D}}$ performs a number of visits which is very often much less than the ones performed by TB$_{\text{D}}$. In particular, for FT (Figure 2a), for public transport networks, $F_{\text{FT}}$ is almost always less than 0.2 and for social networks it is less than 0.1. Exceptions correspond to TOULOUSE, MELBOURNE and EMAIL-EU-CORE, and this is not surprising as we can observe a relatively small $\min\{|A_P|, |B_P|\}$, which means that both PIVOT-IFUB$_{\text{D}}$ and TB$_{\text{D}}$ require linear time. In the case of ST (Figure 2b), the performance seems to be worse with respect to FT, but there is not doubt that running PIVOT-IFUB$_{\text{D}}$ is far more convenient than running TB$_{\text{D}}$. In the case of public transport networks $F_{\text{ST}}$ is always smaller than 0.52, except for KUOPIO (the smallest graph), TOULOUSE, and LISBON, where both PIVOT-IFUB$_{\text{D}}$ and TB$_{\text{D}}$ are effective because of the few pairs in $R(P)$. In the case of social networks, $F_{\text{ST}}$ seems to behave better, and, apart from COLLEGE, $F_{\text{ST}}$ is always bounded by 0.34. In any case, the advantage of PIVOT-IFUB$_{\text{D}}$, for both FT and ST, is more evident when the number of nodes increases. Indeed, with social networks with more than 500 thousands nodes, the ratio $F_{\text{D}}$ is always less than 0.07, apart from SX-ASKUBUNTU-C2Q (where $F_{\text{D}}$ is bounded by 0.17).

## 6 Concluding remarks

In this paper, we have introduced the concept of pivot-diameter, we have given algorithms to compute it efficiently in practice, and we have seen that our choice of the pivots, i.e. choosing vertices with the maximum number of exiting temporal edges, leads very often to a large coverage of pairs. For the networks in our dataset with low coverage, we have additionally verified that by simply choosing as pivots the top-degree vertices in the underlying directed static graph the coverage becomes higher than 98%. Even if a discussion about the many possible choices of pivots in order to maximize the coverage is outside the scope of this paper, we think that this problem deserves further (both theoretical and experimental) investigations to be addressed in a future work.

## References

1 M. Borassi, D. Coudert, P. Crescenzi, and A. Marino. On computing the hyperbolicity of real-world graphs. In *Proc. 23rd Annual European Symposium on Algorithms*, pages 215–226, 2015.

2 M. Borassi, P. Crescenzi, and M. Habib. Into the square: On the complexity of some quadratic-time solvable problems. *Electron. Notes Theor. Comput. Sci.*, 322:51–67, 2016.

3 M. Borassi, P. Crescenzi, M. Habib, W. A. Kosters, A. Marino, and F. W. Takes. Fast diameter and radius bfs-based computation in (weakly connected) real-world graphs: With an application to the six degrees of separation games. *Theor. Comput. Sci.*, 586:59–80, 2015.

4 F. Brunelli, P. Crescenzi, and L. Viennot. On computing pareto optimal paths in weighted time-dependent networks. *Inf. Proc. Let.*, 168:106086, 2021.

5 A. Casteigts, J. G. Peters, and J. Schoeters. Temporal cliques admit sparse spanners. In *Proc. 46th ICALP*, pages 134:1–134:14, 2019.

6 P. Crescenzi, R. Grossi, M. Habib, L. Lanzi, and A. Marino. On computing the diameter of real-world undirected graphs. *Theor. Comput. Sci.*, 514:84–95, 2013.

7 P. Crescenzi, R. Grossi, C. Imbrenda, L. Lanzi, and A. Marino. Finding the diameter in real-world graphs - experimentally turning a lower bound into an upper bound. In *Proc. 18th Annual European Symposium on Algorithms*, pages 302–313, 2010.

8 P. Crescenzi, R. Grossi, L. Lanzi, and A. Marino. On computing the diameter of real-world directed (weighted) graphs. In *Proc. 11th International Symposium on Experimental Algorithms*, pages 99–110. Springer, 2012.

9 P. Crescenzi, C. Magnien, and A. Marino. Approximating the temporal neighbourhood function of large temporal graphs. *Algorithms*, 12(10):211, 2019.

10 P. Crescenzi, C. Magnien, and A. Marino. Finding top-k nodes for temporal closeness in large temporal graphs. *Algorithms*, 13(9):211, 2020.

11 IMDb. IMDb Datasets. `http://www.imdb.com/interfaces`.

12 R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

13 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, 2018.

14 M. Latapy, T. Viard, and C. Magnien. Stream graphs and link streams for the modeling of interactions over time. *Soc. Netw. Anal. Min.*, 8(1):61:1–61:29, 2018.

15 J. Leskovec. Stanford Large Network Dataset Collection (SNAP). `http://snap.stanford.edu/data/index.html`.

16 M. Ley. DBLP - some lessons learned. *Proc. VLDB Endow.*, 2(2):1493–1500, 2009.

17 Institute of Web Science and Technologies. The Koblenz Network Collection. Available online: `http://konect.uni-koblenz.de`.

18 F. W. Takes and W. A. Kosters. Determining the diameter of small world networks. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management*, pages 1191–1196, 2011.

19 F. W. Takes and W. A. Kosters. Computing the eccentricity distribution of large graphs. *Algorithms*, 6(1):100–118, 2013.

20 Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. Arnetminer: extraction and mining of academic social networks. In *Proc. 14th KDD*, pages 990–998, 2008.

21 V. Vassilevska Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE FOCS*, pages 645–654, 2010.

22 H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proc. of the VLDB Endowment*, 7(9):721–732, 2014.
23 H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu. Efficient algorithms for temporal path computation. *IEEE Trans. on Knowl. and Data Eng.*, 28(11):2927–2942, 2016.
24 B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. of Found. of Comp. Sci.*, 14(02):267–285, 2003.

## A Proofs

### A.1 Proof of Lemma 1

In order to prove the first assertion, it suffices to show that there exists a $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$ in $(V, \mathbb{E})$ whose duration (respectively, travel time) is $\tau$ if and only if there exists a $[-t_\omega, -t_\alpha]$-compatible path from $v$ to $u$ in $(V, \mathbb{F})$ whose duration (respectively, travel time) is $\tau$. Let $P = e_1 e_2 \dots e_k$ be a $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$ in $(V, \mathbb{E})$, where $e_i = (u_i, v_i, t_i, \lambda_i)$ for any $i$ with $1 \leq i \leq k$, $u_1 = u$, $v_k = v$, $t_1 \geq t_\alpha$, $t_k + \lambda_k \leq t_\omega$, and, for each $i$ with $1 < i \leq k$, $u_i = v_{i-1}$ and $t_i \geq t_{i-1} + \lambda_{i-1}$. Since $-t_k - \lambda_k \geq -t_\omega$ and $-t_1 - \lambda_1 + \lambda_1 \leq -t_\alpha$, and since $t_i \geq t_{i-1} + \lambda_{i-1}$ if and only if $-t_{i-1} - \lambda_{i-1} \geq -t_i - \lambda_i + \lambda_i$, we have that $\rho(P) = \rho(e_k)\rho(e_{k-1}) \dots \rho(e_1)$ is a $[-t_\omega, -t_\alpha]$-compatible path from $v$ to $u$ in $(V, \mathbb{F})$. Since the travel times of the temporal edges have not been changed, we have that the travel time of $P$ is equal to the travel time of $\rho(P)$. Moreover, the duration of $P$ is equal to $t_k + \lambda_k - t_1$: since $t_k + \lambda_k - t_1 = -t_1 - \lambda_1 + \lambda_1 - (-t_k - \lambda_k)$, we have that $P$ and $\rho(P)$ have also the same duration. The opposite direction can be proved similarly.

In order to prove the second assertion, it suffices to show that there exists a $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$ in $(V, \mathbb{E})$ whose arrival (respectively, latest departure) time is $\tau$ if and only if there exists a $[-t_\omega, -t_\alpha]$-compatible path from $v$ to $u$ in $(V, \mathbb{F})$ whose departure (respectively, arrival) time is $-\tau$. As before, let $P = e_1 e_2 \dots e_k$ be a $[t_\alpha, t_\omega]$-compatible path from $u$ to $v$ in $(V, \mathbb{E})$, and let $\rho(P) = \rho(e_k)\rho(e_{k-1}) \dots \rho(e_1)$ be the corresponding $[-t_\omega, -t_\alpha]$-compatible path from $v$ to $u$ in $(V, \mathbb{F})$. The arrival (respectively, departure) time of $P$ is $t_k + \lambda_k$ (respectively, $t_1$), while the departure (respectively, arrival) time of $P$ is $-t_k - \lambda_k = -(t_k + \lambda_k)$ (respectively, $-t_1 - \lambda_1 + \lambda_1 = -t_1$). The opposite direction can be proved similarly, and this concludes the proof of the lemma.

### A.2 Proof of Theorem 9

In the worst case, the number of iterations of the while loop is $O(n)$. The computation of the lower bound requires $O(\text{S-TIME}_{\text{D}}(n, m) + \text{T-TIME}_{\text{D}}(n, m))$ and it is the dominant part of the **while** loop, if we can speed up the computation of M at Line 3. To this aim we can perform the following precomputation. Let us define $\overline{\text{EAT}} = \overline{\text{FT}} = \text{LDT}$, $\underline{\text{EAT}} = \underline{\text{FT}} = \text{EAT}$, $\overline{\text{ST}} = \underline{\text{ST}} = \text{ST}$. For $\text{D} \in \{\text{EAT}, \text{FT}, \text{ST}\}$ and for each $p = (x, t) \in P$, we define $\pi_p$ as the sequence of nodes $v \in A_{\{p\}}$ sorted in non-increasing order with respect to $d_{\overline{\text{D}}}^{[t_\alpha, t]}(v, x)$, and $\gamma_P$ as the sequence of nodes $v \in B_{\{p\}}$ sorted in non-decreasing order with respect to $d_{\underline{\text{D}}}^{[t_i, t_\omega]}(x, v)$. This precomputation can be performed in $O(|P| \cdot (\max\{\text{S-TIME}_{\text{D}}(n, m), \text{T-TIME}_{\text{D}}(n, m)\} + n \log n))$. When Line 3 is performed, it is sufficient to consider, for each pivot $p$, the pair of nodes $(u, v)$, where $u$ is the leftmost element of $\pi_p$ not in $\hat{A}_P$ and $v$ is the leftmost element of $\gamma_p$ not in $\hat{B}_P$. Hence, this line costs $O(|P|)$ time. Once M has been selected, the other lines, i.e. Line 4 and Line 6, cost $O(1)$ time. As a result, we obtain the time and space bounds of the theorem, where the space overhead $O(|P| \cdot n)$ is due to the space required for maintaining the result of the preprocessing.

## B Tables and figures

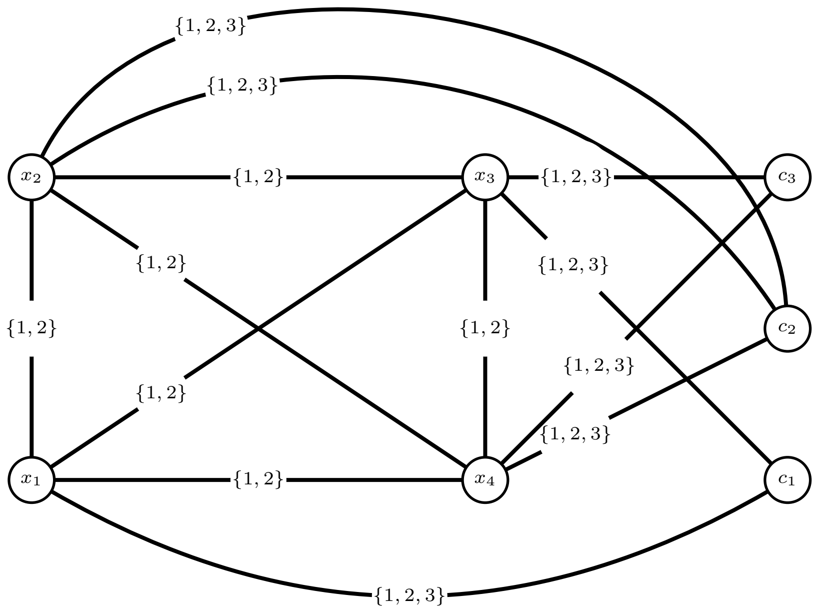
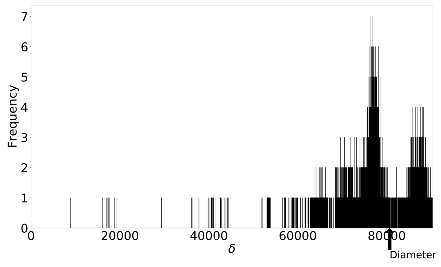


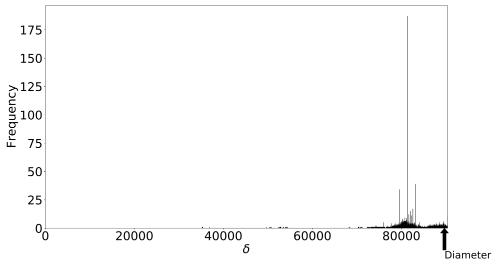
**Figure 3** The reduction from disjoint sets to diameter computation. In this case, $c_1 = \{x_1, x_3\}$, $c_2 = \{x_2, x_4\}$, and $c_3 = \{x_3, x_4\}$. All temporal edges have travel time equal to 1. For any distance, the diameter is 3, and, indeed, $c_1$ and $c_2$ are disjoint.

**Table 6** Number of times the lower bound returned by 2SW$_{\text{D}}(k)$ is at least the one of RS$_{\text{D}}(k)$.

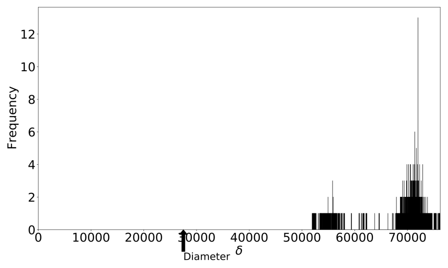
| D | NETWORKS (Number of nets) | Number of times the lower bound returned by 2SW$_{\text{D}}(k)$ is $\geq$ than the one of RS$_{\text{D}}(k)$ | | | |
|---|---|---|---|---|---|
| | | $k = 4$ | $k = 4\log_2 n$ | $k = 8\log_2 n$ | $k = 16\log_2 n$ |
| EAT | PUBLIC TRANSPORT (25) | 23 | 25 | 25 | 25 |
| LDT | PUBLIC TRANSPORT (25) | 23 | 25 | 25 | 25 |
| FT | PUBLIC TRANSPORT (25) | 25 | 25 | 25 | 25 |
| | SOCIAL (18) | 15 | 16 | 18 | 18 |
| ST | PUBLIC TRANSPORT (25) | 22 | 25 | 25 | 25 |
| | SOCIAL (18) | 16 | 16 | 18 | 18 |



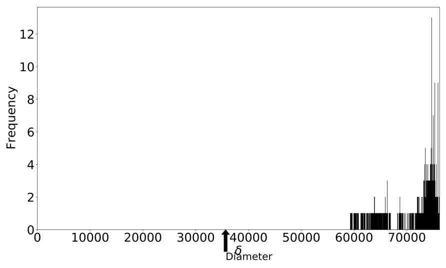
**(a)** Distribution of $\delta$ values for EAT in ROME.



**(b)** Distribution of $\delta$ values for LDT in ROME.



**(c)** Distribution of $\delta$ values for EAT in PALERMO.



**(d)** Distribution of $\delta$ values for LDT in PALERMO.

**Figure 4** Distribution of $\delta$ values for EAT and LDT, for ROME and PALERMO. For each $x$ the amount of vertices $v$ having $\delta(v) = x$.

**Table 7** Number of visits performed by EAT-ALG and LDT-ALG wrt to $n$, where $n$ is the number of visits required by $\text{TB}_{\text{EAT}}$ and $\text{TB}_{\text{LDT}}$. These values are plot in Figure 1a and Figure 1b as a function of $n$.

| NETWORK | visits/$n$ | | NETWORK | visits/$n$ | |
|---|---|---|---|---|---|
| | EAT | LDT | | EAT | LDT |
| KUOPIO | 74.50% | 1.64% | WINNIPEG | 62.20% | 13.51% |
| RENNES | 36.67% | 14.57% | BRISBANE | 1.17% | 0.36% |
| GRENOBLE | 6.92% | 8.66% | DUBLIN | 3.37% | 0.88% |
| VENICE | 15.80% | 9.07% | ADELAIDE | 0.62% | 0.01% |
| BELFAST | 9.02% | 1.41% | LISBON | 8.99% | 3.62% |
| CANBERRA | 2.17% | 0.14% | PRAGUE | 0.02% | 0.02% |
| TURKU | 60.32% | 0.22% | HELSINKI | 1.06% | 1.55% |
| LUXEMBOURG | 80.69% | 0.15% | BERLIN | 38.80% | 11.56% |
| NANTES | 17.98% | 12.11% | ROME | 26.93% | 2.97% |
| DETROIT | 37.83% | 13.67% | MELBOURNE | 0.47% | 0.10% |
| TOULOUSE | 1.14% | 0.21% | SYDNEY | 0.02% | 0.02% |
| PALERMO | 100.00% | 100.00% | PARIS | 0.27% | 0.46% |
| BORDEAUX | 4.10% | 1.80% | | | |

**Table 8** Running time of our implementations of the visits reported in Table 1 for each of the networks in our dataset (mean in seconds and variance, over a random sample of 100 visits).

| NETWORK | SINGLE SOURCE | | | | | | | | SINGLE TARGET | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | EAT | | LDT | | FT | | ST | | EAT | | LDT | | FT | | ST | |
| | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR | MEAN(S) | VAR |
| KUOPIO | 0.237 | 0.004 | 0.213 | 0.001 | 0.177 | 0.002 | 0.198 | 0.001 | 0.232 | 0.001 | 0.098 | 0.001 | 0.21 | 0.001 | 0.202 | 0.001 |
| RENNES | 0.776 | 0.039 | 0.759 | 0.004 | 0.637 | 0.016 | 0.658 | 0.005 | 0.778 | 0.003 | 0.336 | 0.001 | 0.732 | 0.01 | 0.671 | 0.006 |
| GRENOBLE | 0.754 | 0.085 | 0.721 | 0.002 | 0.528 | 0.016 | 0.53 | 0.009 | 0.73 | 0.002 | 0.351 | 0.001 | 0.6 | 0.014 | 0.573 | 0.008 |
| VENICE | 0.698 | 0.058 | 0.785 | 0.004 | 0.675 | 0.02 | 0.691 | 0.014 | 0.804 | 0.003 | 0.366 | 0.001 | 0.668 | 0.018 | 0.68 | 0.015 |
| BELFAST | 0.821 | 0.058 | 0.816 | 0.003 | 0.749 | 0.021 | 0.717 | 0.016 | 0.849 | 0.003 | 0.383 | 0.001 | 0.764 | 0.018 | 0.718 | 0.011 |
| CANBERRA | 0.629 | 0.028 | 0.838 | 0.005 | 0.793 | 0.035 | 0.797 | 0.02 | 0.893 | 0.002 | 0.381 | 0.001 | 0.847 | 0.027 | 0.769 | 0.015 |
| TURKU | 0.798 | 0.081 | 0.991 | 0.004 | 0.979 | 0.031 | 0.991 | 0.033 | 1.006 | 0.005 | 0.412 | 0.001 | 0.964 | 0.048 | 0.957 | 0.025 |
| LUXEMBOURG | 1.216 | 0.13 | 1.347 | 0.003 | 1.445 | 0.022 | 1.243 | 0.013 | 1.419 | 0.003 | 0.578 | 0.001 | 1.438 | 0.018 | 1.279 | 0.015 |
| NANTES | 1.315 | 0.136 | 1.374 | 0.008 | 1.192 | 0.053 | 1.175 | 0.028 | 1.391 | 0.006 | 0.603 | 0.001 | 1.159 | 0.045 | 1.14 | 0.023 |
| DETROIT | 1.565 | 0.142 | 1.666 | 0.007 | 1.653 | 0.026 | 1.536 | 0.019 | 1.693 | 0.012 | 0.67 | 0.001 | 1.661 | 0.066 | 1.576 | 0.034 |
| TOULOUSE | 1.431 | 0.158 | 1.598 | 0.012 | 1.444 | 0.082 | 1.408 | 0.056 | 1.621 | 0.011 | 0.694 | 0.001 | 1.492 | 0.02 | 1.353 | 0.023 |
| PALERMO | 1.961 | 0.089 | 1.843 | 0.007 | 1.835 | 0.01 | 1.718 | 0.017 | 1.836 | 0.012 | 0.718 | 0.001 | 1.898 | 0.02 | 1.692 | 0.014 |
| BORDEAUX | 1.743 | 0.16 | 1.747 | 0.009 | 1.666 | 0.095 | 1.771 | 0.087 | 1.801 | 0.015 | 0.736 | 0.001 | 1.761 | 0.055 | 1.744 | 0.111 |
| WINNIPEG | 2.782 | 0.206 | 2.636 | 0.028 | 2.985 | 0.08 | 3.062 | 0.169 | 2.674 | 0.017 | 1.045 | 0.001 | 2.884 | 0.21 | 3.002 | 0.207 |
| BRISBANE | 1.732 | 0.261 | 2.797 | 0.05 | 2.68 | 0.472 | 2.598 | 0.366 | 2.881 | 0.031 | 1.207 | 0.001 | 2.669 | 0.34 | 2.491 | 0.222 |
| DUBLIN | 1.566 | 0.017 | 2.992 | 0.062 | 3.241 | 0.392 | 3.118 | 0.391 | 3.097 | 0.06 | 1.281 | 0.001 | 3.002 | 0.578 | 3.174 | 0.493 |
| ADELAIDE | 1.598 | 0.014 | 2.975 | 0.026 | 3.387 | 0.311 | 3.675 | 0.278 | 3.268 | 0.038 | 1.261 | 0.001 | 3.297 | 0.411 | 3.592 | 0.55 |
| LISBON | 1.816 | 0.023 | 3.433 | 0.054 | 2.723 | 0.408 | 2.696 | 0.226 | 3.644 | 0.047 | 1.653 | 0.001 | 2.841 | 0.47 | 2.792 | 0.441 |
| PRAGUE | 2.504 | 0.057 | 4.595 | 0.112 | 4.46 | 1.142 | 4.038 | 0.404 | 4.891 | 0.143 | 2.097 | 0.021 | 4.671 | 1.181 | 3.987 | 0.42 |
| HELSINKI | 2.8 | 0.066 | 4.889 | 0.134 | 5.248 | 0.953 | 5.026 | 0.65 | 5.256 | 0.105 | 2.139 | 0.001 | 5.328 | 0.642 | 4.933 | 0.531 |
| BERLIN | 4.481 | 0.039 | 7.744 | 0.101 | 8.784 | 0.357 | 7.471 | 0.416 | 8.286 | 0.191 | 3.249 | 0.004 | 8.702 | 0.873 | 7.428 | 0.739 |
| ROME | 4.451 | 0.039 | 7.954 | 0.114 | 9.861 | 1.236 | 7.235 | 0.751 | 8.654 | 0.26 | 3.292 | 0.001 | 9.591 | 2.471 | 7.528 | 0.651 |
| MELBOURNE | 4.223 | 0.12 | 7.936 | 0.222 | 8.182 | 3.522 | 9.109 | 4.072 | 8.324 | 0.245 | 3.419 | 0.001 | 8.123 | 1.798 | 7.694 | 2.435 |
| SYDNEY | 4.739 | 0.2 | 9.164 | 0.369 | 8.641 | 4.41 | 9.529 | 3.859 | 9.402 | 0.61 | 3.963 | 0.001 | 8.749 | 3.64 | 8.852 | 3.057 |
| PARIS | 6.691 | 0.274 | 14.891 | 0.893 | 12.154 | 6.056 | 11.798 | 7.146 | 12.42 | 0.769 | 5.694 | 0.006 | 12.167 | 6.889 | 11.752 | 5.771 |
| TOPOLOGY | 0.185 | 0.001 | 0.371 | 0.001 | 0.258 | 0.007 | 0.291 | 0.008 | 0.312 | 0.001 | 0.19 | 0.001 | 0.263 | 0.006 | 0.313 | 0.009 |
| ELEC | 0.167 | 0.001 | 0.346 | 0.001 | 0.173 | 0.001 | 0.187 | 0.001 | 0.307 | 0.001 | 0.16 | 0.001 | 0.177 | 0.001 | 0.212 | 0.001 |
| FACEBOOK-WOSN-WALL | 1.436 | 0.008 | 3.338 | 0.061 | 1.597 | 0.117 | 1.82 | 0.197 | 2.752 | 0.031 | 1.37 | 0.001 | 1.91 | 0.14 | 2.255 | 0.235 |
| COLLEGE | 0.099 | 0.001 | 0.202 | 0.001 | 0.121 | 0.001 | 0.124 | 0.001 | 0.196 | 0.001 | 0.094 | 0.001 | 0.137 | 0.001 | 0.169 | 0.001 |
| SX-MATHOVERFLOW-A2Q | 0.164 | 0.001 | 0.498 | 0.001 | 0.218 | 0.001 | 0.23 | 0.001 | 0.417 | 0.001 | 0.168 | 0.001 | 0.229 | 0.001 | 0.26 | 0.001 |
| SX-MATHOVERFLOW-C2A | 0.294 | 0.001 | 0.789 | 0.001 | 0.362 | 0.001 | 0.386 | 0.002 | 0.659 | 0.001 | 0.305 | 0.001 | 0.369 | 0.002 | 0.436 | 0.003 |
| SX-MATHOVERFLOW-C2Q | 0.299 | 0.001 | 0.806 | 0.001 | 0.363 | 0.002 | 0.395 | 0.001 | 0.711 | 0.003 | 0.313 | 0.001 | 0.388 | 0.002 | 0.449 | 0.002 |
| EMAIL-EU-CORE | 0.541 | 0.001 | 1.078 | 0.004 | 0.746 | 0.031 | 0.843 | 0.05 | 0.96 | 0.005 | 0.509 | 0.001 | 0.876 | 0.017 | 1.034 | 0.032 |
| SX-ASKUBUNTU-A2Q | 0.43 | 0.001 | 1.915 | 0.006 | 0.783 | 0.001 | 0.88 | 0.003 | 1.605 | 0.005 | 0.441 | 0.001 | 0.85 | 0.008 | 0.899 | 0.005 |
| SX-ASKUBUNTU-C2Q | 0.5 | 0.001 | 2.01 | 0.009 | 0.847 | 0.001 | 0.948 | 0.002 | 1.727 | 0.007 | 0.514 | 0.001 | 0.92 | 0.005 | 0.995 | 0.009 |
| SX-ASKUBUNTU-C2A | 0.554 | 0.001 | 2.09 | 0.008 | 0.947 | 0.004 | 1.017 | 0.006 | 1.816 | 0.003 | 0.561 | 0.001 | 0.951 | 0.006 | 1.053 | 0.009 |
| SX-SUPERUSER-A2Q | 0.658 | 0.001 | 2.639 | 0.011 | 1.16 | 0.004 | 1.138 | 0.003 | 2.165 | 0.005 | 0.676 | 0.001 | 1.161 | 0.008 | 1.286 | 0.016 |
| SX-MATHOVERFLOW | 0.771 | 0.001 | 1.845 | 0.008 | 0.91 | 0.025 | 0.992 | 0.032 | 1.565 | 0.009 | 0.785 | 0.001 | 0.946 | 0.032 | 1.104 | 0.039 |
| SX-SUPERUSER-C2Q | 0.731 | 0.001 | 2.596 | 0.016 | 1.129 | 0.002 | 1.273 | 0.006 | 2.254 | 0.014 | 0.755 | 0.001 | 1.203 | 0.006 | 1.321 | 0.011 |
| SX-SUPERUSER-C2A | 0.814 | 0.001 | 2.875 | 0.012 | 1.331 | 0.017 | 1.385 | 0.008 | 2.424 | 0.011 | 0.826 | 0.001 | 1.346 | 0.016 | 1.51 | 0.024 |
| SX-ASKUBUNTU | 1.481 | 0.007 | 4.212 | 0.032 | 1.989 | 0.071 | 2.108 | 0.054 | 3.587 | 0.029 | 1.522 | 0.001 | 2.058 | 0.056 | 2.458 | 0.116 |
| SX-SUPERUSER | 2.288 | 0.022 | 6.013 | 0.077 | 2.801 | 0.102 | 3.126 | 0.19 | 5.102 | 0.073 | 2.262 | 0.002 | 3.035 | 0.26 | 3.466 | 0.324 |
| WIKI-TALK-TEMPORAL | 12.373 | 0.391 | 29.842 | 2.129 | 14.759 | 13.015 | 15.014 | 6.032 | 24.993 | 0.973 | 12.309 | 0.009 | 17.823 | 6.574 | 20.561 | 8.103 |

# Document Retrieval Hacks

**Simon J. Puglisi**
Department of Computer Science, University of Helsinki, Finland

**Bella Zhukova**
Department of Computer Science, University of Helsinki, Finland

## Abstract

Given a collection of strings, document listing refers to the problem of finding all the strings (or *documents*) where a given query string (or *pattern*) appears. Index data structures that support efficient document listing for string collections have been the focus of intense research in the last decade, with dozens of papers published describing exotic and elegant compressed data structures. The problem is now quite well understood in theory and many of the solutions have been implemented and evaluated experimentally. A particular recent focus has been on highly repetitive document collections, which have become prevalent in many areas (such as version control systems and genomics – to name just two very different sources).

The aim of this paper is to describe simple and efficient document listing algorithms that can be used in combination with more sophisticated techniques, or as baselines against which the performance of new document listing indexes can be measured. Our approaches are based on simple combinations of scanning and hashing, which we show to combine very well with dictionary compression to achieve small space usage. Our experiments show these methods to be often much faster and less space consuming than the best specialized indexes for the problem.

**2012 ACM Subject Classification** Information systems → Data compression

**Keywords and phrases** String Processing, Pattern matching, Document listing, Document retrieval, Succinct data structures, Repetitive text collections

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.12

**Funding** This work was funded in part by the Academy of Finland via grant 319454.

**Acknowledgements** Our thanks go to Dustin Cobas for prompt help in getting his codebase to compile on our system, and to Massimiliano Rossi for assistance with datasets.

## 1 Introduction

Given a collection of strings $T_0, T_1, \ldots, T_d$, called *documents* the *document listing* problem is to preprocess the documents and build an index data structure so that later, given a previously unseen string $\mathsf{Q}$ (the *pattern* or *query*), we can report all $i \in 0..d$ for which $T_i$ contains $\mathsf{Q}$ as a substring.

The document listing problem was introduced, almost 20 years ago now, by Muthukrishnan [14] as a natural variant of the pattern matching problem. At the time, finding *all* the *occ* occurrences of a pattern in a set of texts in time proportional to *occ*, was efficiently solvable using suffix trees and arrays. The algorithmics attractiveness of the document listing problem lay in cases where *docc* – the number of documents containing the pattern – was very much smaller than *occ*, where the "brute force" solution of enumerating over the set of all *occ* occurrences to find the distinct document ids seems wasteful. Muthukrishnan gave a $\mathrm{O}(docc + |\mathsf{Q}|)$ time solution, which is optimal.

A convenient way of thinking about Muthukrishnan's approach is in terms of the suffix array [13], $\mathsf{SA}$, for the string $\mathsf{T}$ formed by concatenating the $T_i$ strings of the collection into one string $\mathsf{T} = \mathsf{T}_0\$\mathsf{T}_1\$\ldots\mathsf{T}_d\$$ where \$ is a "separator" symbol guaranteed not to be part of any query pattern. The suffix array, $\mathsf{SA}[0, n-1]$, for a string $\mathsf{T}$ of length $n$ is an array of integers containing a permutation of $(0 \ldots n-1)$, so that the suffixes of $\mathsf{T}$ starting at the consecutive positions indicated in SA are in lexicographical order: $\mathsf{T}[\mathsf{SA}[i], n] < \mathsf{T}[\mathsf{SA}[i+1], n]$. Because of the lexicographic ordering, all the suffixes starting with a given substring Q of T form an interval $\mathsf{SA}[s, e]$, which can be determined by binary search in $O(|\mathsf{Q}| \log n)$ time. Muthukrishnan's solution to document listing defines an array $\mathsf{DA}[0, n-1]$ in which $\mathsf{DA}[i] = x$ if and only if suffix $\mathsf{T}[\mathsf{SA}[i], n]$ has its starting position inside the area of T corresponding to document $\mathsf{T}_x$. We say that $\mathsf{DA}[i]$ contains the *document id* for the suffix starting at $\mathsf{SA}[i]$.

With DA in hand, document listing for pattern Q then becomes simply a matter of enumerating the distinct elements in the interval $\mathsf{DA}[s, e]$. Muthukrishnan shows this can be done in optimal $\mathrm{O}(docc)$ time via a range minimum query data structure after finding interval $[s, e]$ in $\mathrm{O}(m)$ time using the suffix tree of T. The solution requires $\mathrm{O}(n \log n)$ bits of space, and subsequent work by several authors [2, 4, 5, 7, 8, 16, 22] has aimed to reduce space in pursuit of practical solutions (we refer the reader to [15] for a survey of results prior to 2014, and [4] for more recent results). Almost all solutions make use of DA in some form.

A particular recent focus [2, 4, 5, 7, 16] has been on document retrieval indexes for *highly repetitive document collections*, which have become prevalent in many areas, such as version control systems and genomics – to name just two very different sources (see [17, 18] for recent surveys of results on highly repetitive data in general).

**Our contribution.** The main results of this paper are twofold:

1. We describe and implement a handful of algorithms for document listing that work by gathering the distinct document ids in $\mathsf{DA}[s, e]$ while scanning that interval. These algorithms are so simple as to be almost trivial, but nonetheless seem to have escaped scrutiny to date. We show experimentally that these algorithms are very fast, and represent a new baseline against which the speed of more complex document listing indexes should be gauged.
2. We show that the DA for a highly-repetitive collection can be effectively compressed via relative Lempel-Ziv (RLZ) parsing, a compression method that is known to support fast extraction of arbitrary intervals from its underlying sequence. We show that RLZ-compressed document arrays combine well with the aforementioned scan-based document listing algorithms, leading to the smallest (or near smallest) indexes we know of for that problem, which are often an order of magnitude or faster than the best competing methods.

**Roadmap.** In the next section we describe our new scan-based method for document listing and compare it experimentally to another, previously described, brute-force method for the problem. Then, in Section 3 we describe how DA is amenable to RLZ compression and combine this representation of DA with variants of our scan-based document listing algorithm. We also show that for certain types of repetitive document collection, run-length encoding can be an effective way to compress DA. Section 4 compares our new document listing indexes to state-of-the-art methods, showing them to be significantly faster and often less space consuming. We then conclude with some possible directions for future work.

## 2 Refined Brutes

We begin with a "brute-force" document listing method that scans, copies and sorts $\mathsf{DA}[s, e]$. It is the only published method we know of that explicitly inspects every element of $\mathsf{DA}[s, e]$, and is used in several papers [4, 5, 7] where it is reported to be faster than specialized document listing methods when $e - s + 1 = occ$ is small. We refer to this method as sort[1].

[1] The same method is referred to as Packed-sort in [4, 5] and SORT in [7].

The approach taken is to allocate a buffer of $occ$ integers, copy the contents of $\mathsf{DA}[s,e]$ into it, sort the buffer (bringing duplicate document ids together) and then scan the buffer removing duplicates. The final contents of the buffer are the distinct elements in $\mathsf{DA}[s,e]$.

While sort certainly gets the job done, it also looks and sounds suspiciously like a straw man: all that movement of data. The obvious alternative to sorting, of course, is something akin to the counting part of counting sort: scan $\mathsf{DA}[s,e]$ and use an array $\mathsf{B}$ of $d$ elements (initially all 0) to simply record which document ids are present in the interval – indeed, $d$ bits will do. Whenever $\mathsf{B}[\mathsf{DA}[i]] = 0$ we add document $\mathsf{DA}[i]$ to the result set (again implemented as a vector) and set $B[\mathsf{DA}[i]]$ to 1. If desirable, we can reuse $\mathsf{B}$ between queries by scanning the result set at the end of the scan of $\mathsf{DA}[s,e]$ and resetting all the set bits to 0. We call this document listing method bv (for bit vector).

A possible concern with bv is the $d$ extra bits it uses. The minimum extra space (without trying too hard) for document listing is the size of the result set itself $docc \log d$ bits. We note, however, that $d$ bits can easily be less than the $\mathrm{O}(occ \log n)$ bits that sort uses when it copies $\mathsf{DA}[s,e]$ to its buffer for sorting. In any case, we can reduce the working space in at least two ways.

For cases when $\mathrm{O}(occ \log n)$ extra space is preferable, hashing is a simple option. To this end, we implemented a simple linear probing hash table that indicates on inserting $\mathsf{DA}[i]$ whether that element is already present in the table. If not, it is inserted into the table and added to the result set. The hash table allocates space for $2\lceil \log occ \rceil$ elements, ensuring a low load factor (maximum 0.5) and therefore fast insert time. We call this variant hash and implemented a second version of it, uset, that uses a C++ `std::unordered_set` as a sanity check to our hand-rolled linear-probing hash table. Working space usage can be further reduced to $\mathrm{O}(docc \log n)$ by using a search tree to accumulate distinct elements instead of a hash table. We implemented such a variant, which we denote tree, variant using a C++ `std::set`, which is backed by a red-black tree.

We measured the performance of the above scan-based document listing algorithms (including the previously described sort-based method) on three document collections. These data sets are described in Section 4 and Table 1. We tried all variants on two trivial encodings of DA. The first, Plain stores DA as an array of 32-bit unsigned integers. The second, Packed, packs elements of DA into $\lceil \log d \rceil$ bits each, with each element still being accessible in $\mathrm{O}(1)$ time, albeit with a higher constant of proportionality than element access with the Plain encoding.

Results for query patterns with 4-mers of high frequencies are shown in Figure 1. The bitvector-based scanning approach (bv) is a clear winner on all data sets. It is on average an order of magnitude faster than the sort-based baseline (sort) on the Page data set, and two order of magnitude faster on Revision and Influenza. The linear probing hash-based method (hash) is also significantly faster than sort on all data sets (1.32-2.27$\times$). Somewhat unsurprisingly, Plain document arrays always led to faster queries than did Packed document arrays for all methods. We did not include Packed-tree and Packed-uset to save space in the plots because they are dominated by other methods. Figure 1 also shows that the differences between the two variants Plain-bv and Packed-bv are bigger than the differences for other variants. Taking into account that the number of DA access must be the same in all scanning algorithms, we do not yet have a good explanation for it, but it could be due to the overhead of the function call required with Packed-bv that makes the loop more difficult to unroll for the compiler.

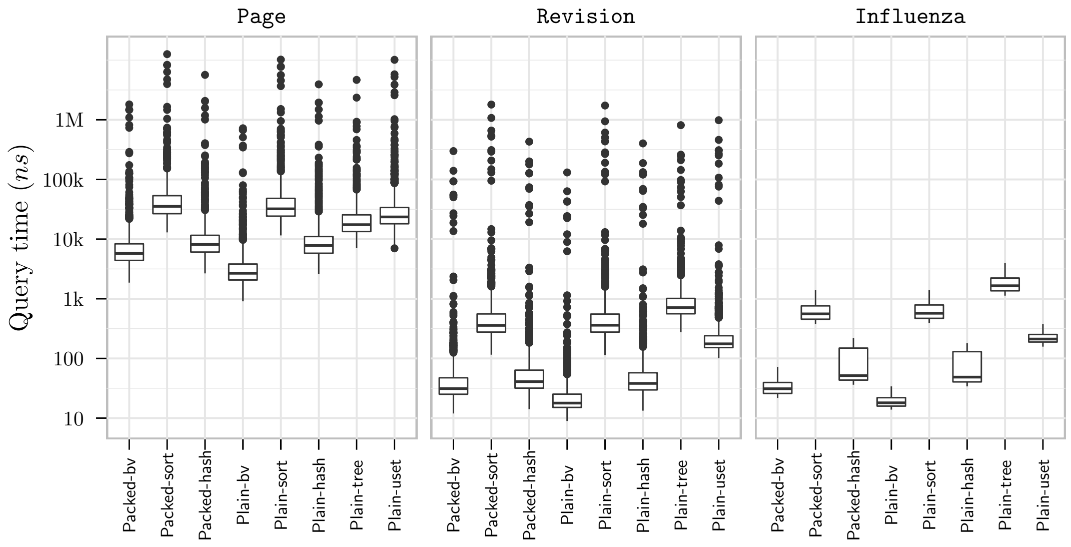


**Figure 1** Boxplots showing time for Packed-sort versus our variants. These results are obtained on query patterns with $k$-mers of length 4 with high frequencies (see Section 4 and Table 1 for details of data sets and patterns). Note that the vertical axis is logarithmic.

## 3 Petite Brutes

On highly repetitive collections, the size of the compressed index used to find the interval of DA containing the document occurrences for the query pattern is dwarfed by the space used by DA. Even when DA is stored bit-packed in $n\lceil\log d\rceil$ bits, it is still almost 70 times bigger than the interval finding component on all our data sets. Reducing the space used by DA is therefore important.

As several authors have now observed [4,19], repeated substrings in $T$ give rise to repeated intervals in DA. To see this, consider two lexicographically adjacent suffixes of $T[\mathsf{SA}[i], n]$ and $T[\mathsf{SA}[i+1], n]$. If SA[i] and SA[i+1] are preceded by an identical symbol $c \neq \$$ (i.e. $\mathsf{BWT}[i] = \mathsf{BWT}[i+1] = c \neq \$$) then the lexicographical ordering of suffixes in SA dictates that suffixes $T[\mathsf{SA}[i]-1, n]$ and $T[\mathsf{SA}[i+1]-1, n]$ will be adjacent too, and so $\mathsf{DA}[i, i+1]$ will be repeated. More generally, a run of $x$ suffixes $\mathsf{SA}[i, i+x]$ having identical preceding symbols implies the sequence $\mathsf{DA}[i, i+x]$ is repeated in DA.

In [4], Cobas and Navarro employ grammar compression [1] to capture such repetitions and so reduce the size of DA. They preprocess the resulting grammar so that the compressed representation of DA supports the extraction of arbitrary intervals, and use this to implement sort-based document listing: the relevant interval is extracted and copied to a buffer, which is then sorted so that distinct elements can be reported.

In this section we explore compression of DA using relative Lempel-Ziv (RLZ) dictionary compression, as an alternative to grammar compression of DA. Our rational for RLZ is twofold. Firstly, it is known to support fast random-access decompression of arbitrary substrings (in our context, intervals of DA), and, secondly, it has been recently shown to be effective at compressing suffix arrays. Given the tight relationship between SA and DA, it is reasonable to expect RLZ to compress DA well too.

## 3.1 RLZ-Compressed Document Array

RLZ parsing [9, 10] is a variant of the classic LZ77 parsing [24], in which a sequence X is compressed relative to a second sequence R (the reference) by encoding X as a sequence of $z$ substrings, or *phrases*, that occur in R. In our context X = DA.

Intuitively, if the substrings of sequence R are "similar" to those in X, then the parsing will produce a small number of phrases. There are a number of ways to determine a good R for a given X. Random sampling substrings from X has been shown to give good results in practice [9] and also in theory [6]. Several authors have advocated more judicious selection of substrings [11] and reference pruning methods to eliminate sparsely used parts of the reference [23]. We return to the problem of reference selection for DA below.

**Data Structure.** We now describe how, given an reference R, the resulting RLZ parsing of DA is encoded to facilitate fast random access to arbitrary intervals of DA. The approach is essentially the way in which random access is supported in RLZ-compressed text [10], but we include the description here for completeness.

The RLZ parsing of DA is stored in two arrays, $S$ and $P$, both of length $z$. $S$ contains the starting position in DA of each phrase in ascending order. Elements of $S$ are kept in a predecessor data structure. $P$ contains either literal DA values or positions in $R$ as output by the parsing algorithm (the second components of each pair). The type of the $i$th (literal or repeat) is determined from the phrase length, which is in turn computed from the phrase starting positions: $\ell_i = S[i+1] - S[i]$. If $\ell_i = 1$ then $P[i]$ should be interpreted as containing the value of a literal phrase, and otherwise $P[i]$ is the position in $R$ at which the $\ell_i$ symbols constituting the $i$th phrase begin.

Scanning (i.e. decoding) an arbitrary interval DA$[s, e]$ is performed as follows. An output buffer $B$ of size $e - s + 1$ will contain the decoded elements. At a high level, the phrases covering DA$[s, e]$ are decoded and copied to $B$ (some parts of the first and last phrase may not be) until $B$ is full, at which point we are done. To this end, begin by finding the index in $S$ of the predecessor of $s$. Let $x$ denote this index, and so $S[x] \leq s$. If $P[x]$ is a literal phrase, copy its value to the output buffer. Otherwise, ($P[x]$ is non-literal) find the position where the interval in this phrase begins, which is $s - S[x]$ elements from the start of the phrase. The length of the phrase is $\ell = S[x+1] - S[x]$. So we need to decode from this phase $\min(l - (s - S[x]), e - s + 1)$ elements. If $S[x] = s$, to decode phrase $x$ we access $R[P[x]]$, copy $R[P[x]]$ to the output buffer, continuing then to copy $(R[P[x] + 1])$ to $B$, and so on until either the whole phrase has been decoded, or the output buffer is full. And if $S[x] < s$, the copying starts from $(R[P[x]] + (s - S[x]))$. After decoding phrase $x$, if the output buffer is not full, then phrase $x + 1$ is decoded, and so on, until all $e - s + 1$ values have been decoded.

The time to decode the desired interval from the RLZ-compressed DA is $O(e-s+\log\log n)$, where $\log\log n$ is the time need for the initial predecessor query.

**Reference Selection.** In earlier work [20, 21] we explored compression of SA using a combination of differential encoding and RLZ compression, with [20] using random sampling of substrings, and [21] obtaining superior compression performance via a more sophisticated algorithm (adapted from [11]), in which substrings are selected for inclusion in the reference according to the abundance of smaller substrings of length $k$ ($k$-mers) contained therein.

We adopt a similar approach to derive a good reference for DA, which we now describe. DA is logically divided into $n/s$ substrings of equal length $s$ called *segments* (the last segment may have length $< s$). The frequency of each distinct $k$-mer in DA is then computed. Each segment is assigned an initial score according to the $k$-mers it contains. In particular, let $f(x)$, for $k$-mer $x$ be the frequency of $x$ in DA. Let $x \in X_i$ denote that $k$-mer $x$ has at least one occurrence in segment $X_i$. Then the initial score for segment $X_i$, $i \in [0..n/s + 1]$ is the $\ell_p$ norm of the vector of its constituent $k$-mers, calculated as:

$$\text{score}(X_i) = (\Sigma_{x \in X_i} f(x)^p)^{1/p}.$$

The highest scoring $X_i$ is then selected for inclusion in the reference, and frequencies $f(.)$ are then reduced for every $k$-mer $x \in X_i$, in particular $f(x)$ is reduced by the frequency of $k$-mer of $x$ in $X_i$. This process of segment selection and subsequent score adjustment is repeated until the sum of the lengths of the selected segments has reached the target reference length (an input parameter).

Apart from the target reference length, there are three parameters to this process: $s$, the segment size; $k$, the $k$-mer length; and $p$, which affects the way in which $k$-mer frequencies affect segment scores (we set $p = 0.5$ in all experiments in this paper). Figure 2 shows the effect of parameters $s$ and $k$ on the index size for the Influenza dataset for a fixed target reference size. The figure shows that the choice of $k$ and $s$ can significantly influence the size of the index and should be chosen with care. Having said this, the range illustrated in the figure across all tested $(k, s)$ settings is from just under 130MB to just over 270MB for the 336,798,466-entry long DA sequence. This corresponds to a range of 3.2 to 6.7 bits per symbol, which is significantly less than the $\log d = \log(227, 356) = 18$ bits per symbol required to store DA uncompressed. Our point here is that while finding the minimum index size may be difficult, finding a "good" one seems not to be.

We also found that not every position in our constructed references was actually covered by any phrase, and removing the symbols at those positions led to around a small reduction in reference length: 1.5% for Page, 9.35% for Revision, and 2.64% for Influenza.

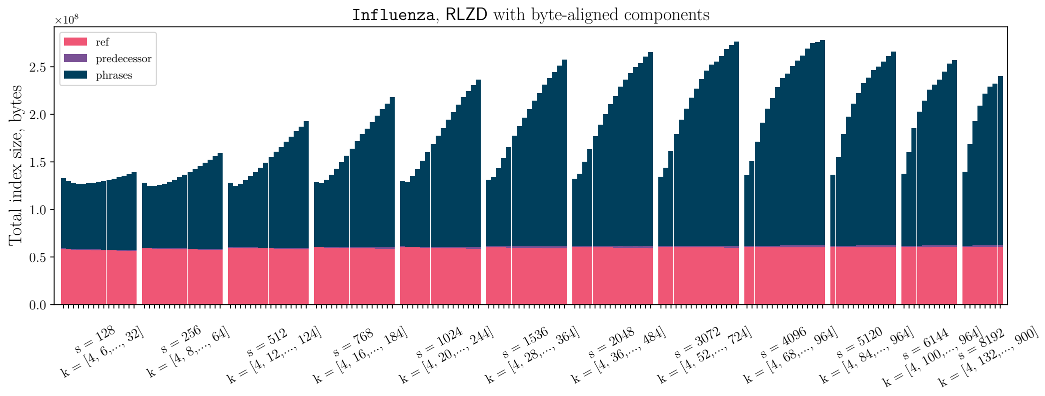


**Figure 2** Effect of segment size $s$ and $k$-mer length on overall index size (reference and phrases), in bytes. Target reference size requested for every case is the same, and is equal to 61.24MB. Actual reference size (ref) in the index varies a little due to removal of symbols that were not referred to during the RLZ compression (see text). Predecessor size (pred) grows with the number of phrases, though it is difficult to see on this plot.

## 3.2 Run-Length Compressed Document Array

To our initial surprise, we found that DA for the Page collection ($n = 1,036,000,000$) consisted of just $17,224,529$ runs of equal document ids. We believe this phenomenon can be explained as follows. Recall that in Page all versions of a given Wikipedia entry are treated as a single document, sharing the same document id. For a given entry with, say, document id $x$, substrings corresponding to terminology (or, e.g., dates) specific to that entry are likely to be shared across versions, and suffixes prefixed with such patterns will be concentrated together in SA, with a corresponding concentration of id $x$ in DA.

With this in mind, we implemented a run-length encoded version of DA that supports random access to intervals via a predecessor data structure holding the starting positions of runs in DA. We call this data structure RLED, and include it in experiments only for the Page collection (Revision and Influenza both had average run length around 1 in their DAs – far too low for RLED to acheive any compression).

## 3.3 Experiments Results

We implemented the RLZ- and RLE-based encodings of DA and combined them with the scan-based "brute-force" methods for document listing described in Section 2. Results of experiments on our three test collections are shown in Figure 3. Plain-bv, the fastest uncompressed method from Section 2 is included as a reference point. On the Page dataset, the methods using RLE are clearly fastest, with RLED-bv being the fastest overall. However the RLZ-based methods also show good performance, with RLZD-bv being only marginally slower than the uncompressed Plain-bv baseline (0.07 vs. 0.06 nanoseconds per query). On the Revision and Influenza datasets, where the RLE method does not apply, we observe a similar pattern in the RLZ-based methods as was observed in Section 2 for the uncompressed methods: RLZD-bv is fastest, followed by RLZD-hash.

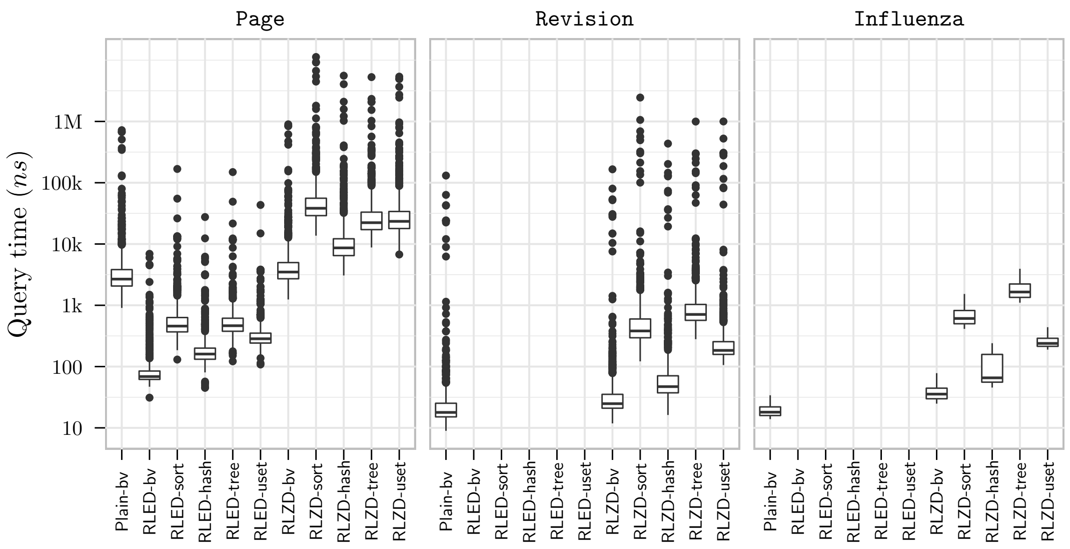


**Figure 3** Boxplots showing query times for our RLZ- and RLE-compressed DA variants. We include the fastest among uncompressed methods, Plain-bv, as a reference point. Note that the vertical axis is logarithmic. These results were obtained on patterns with 4-mers of high frequency (see Section 4 and Table 1 for details of data sets and patterns). The RLZD index was constructed with the following parameters: for Page $k = 312$, $s = 2048$, reference size 7.95MB, for Revision $k = 8$, $s = 1024$, 36.38MB reference, and for Influenza $k = 12$, $s = 256$, 61.24MB reference. Reference sizes were computed experimentally. For Revision and Influenza there is no data for RLED because on these datasets it does not achieve any compression (see text, Section 3.2).

**Table 1** Statistics for document collections: *Collection* name; *Size* in megabytes; *Docs*, number of documents; *Doc size*, average document length; *k-mer*, length of k-mers in the patterns; *Frequency*, frequency of k-mers chosen for patterns; number of *Patterns*; *Occs*, average number of occurrences; *Doc occ*, average number of document occurrences; *Occs/doc*, average ratio of occurrences to document occurrences.

| *Collection* | *Size*, MB | *Docs* | *Doc size* | *k-mer* | *Frequency* | *Patterns* | *Occs* | *Doc occs* | *Occs/doc* |
|---|---|---|---|---|---|---|---|---|---|
| | $(n)$ | $(D)$ | $(n/D)$ | | | | $(occ)$ | $(docc)$ | $(\frac{occ}{docc})$ |
| `Page` | 1 037 | 280 | 38 883 145 | 4 | high | 1 000 | 318 136.00 | 153.81 | 2 068.32 |
| | | | | | mid | | 11 994.35 | 34.23 | 350.43 |
| | | | | | low | | 500.50 | 4.80 | 104.16 |
| | | | | 8 | high | | 87 081.25 | 55.31 | 1 574.51 |
| | | | | | mid | | 7 148.71 | 19.90 | 359.25 |
| | | | | | low | | 500.50 | 6.54 | 76.56 |
| `Revision` | 1 035 | 65 565 | 16 552 | 4 | high | | 317 512.93 | 31 850.79 | 9.97 |
| | | | | | mid | | 12 017.48 | 5 937.12 | 2.02 |
| | | | | | low | | 500.50 | 389.86 | 1.28 |
| | | | | 8 | high | | 86 895.30 | 11 627.18 | 7.47 |
| | | | | | mid | | 7 147.44 | 3 814.97 | 1.87 |
| | | | | | low | | 500.50 | 390.65 | 1.28 |
| `Influenza` | 321 | 227 356 | 1 480 | 4 | high | 141 | 1 819 407.80 | 224 449.40 | 8.11 |
| | | | | | mid | | 562 380.65 | 161 793.77 | 3.48 |
| | | | | | low | 142 | 75.18 | 70.52 | 1.07 |
| | | | | 8 | high | 1 000 | 41 721.84 | 37 822.00 | 1.10 |
| | | | | | mid | | 9 784.96 | 9 665.94 | 1.01 |
| | | | | | low | | 500.50 | 498.79 | 1.00 |

## 4 Performance Comparison

In this section we report on the practical performance of our document listing approaches to other state-of-the art solutions.

### 4.1 Experimental Setup

**Test Machine.** All our experiments[2] were conducted on a 2.10 GHz Intel Xeon E7-4830 v3 CPU equipped with 30 MiB L3 cache and 1.5 TiB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS was Linux (Ubuntu 18.04.5 LTS) running kernel 5.4.0-58-generic. Programs were compiled using `g++` version 7.5.0. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

**Datasets.** Table 1 summarizes the collections and patterns used. Page and Revision are repetitive collections generated from a Finnish-language Wikipedia archive with full version history: 280 pages with a total of 65, 565 revisions. In Page, all the revisions of a page form a single document. In the case of Revision, each page revision becomes a separate document. Influenza is composed of 227, 356 sequences of the H. influenzae virus genomes[3].

**Query Patterns.** To form the query patterns for each dataset (see statistics in Table 1) we first computed the set of all distinct substrings of printable characters of lengths 4 and 8, and their number of occurrences. Each such set of these substrings were then sorted based

[2] Code is available at `https://www2.helsinki.fi/en/researchgroups/algorithmic-bioinformatics/compressed-data-structures`.
[3] All data sets are available at `https://jltsiren.kapsi.fi/rlcsa`.

on their occurrences, and divided into three equal subsets based on their frequencies: high, middle, and low. In the subset if there were less than a 1000 patterns, then all the patterns in this subset form a corresponding file with queries, and if there were more than 1000, we choose a 1000 out of them that represent the subset: for high – 1000 most frequent, for mid – 1000 that are exactly at the middle of the list with the sorted frequencies, and for the low – 1000 least frequent. Patterns with high *occ* represent hard cases for our scan-based methods.

**Indexes Measured.** We used the recent study of Cobas and Navarro [4] as a guide for selecting the best-performing document listing methods to include for each data set. In particular, we measured the following indexes.

- Brute Force. Apart from sort, the method from Section 2, which is referred to as Packed-sort in [4], we also include Brute-C a variant from [4] that uses a grammar-compressed DA, augmented with the length of the expansion of each nonterminal. If the resulting grammar tree has height $h$ the interval DA$[s, e]$ can be extracted in time $\mathrm{O}(h + e - s)$.
- Grammar-Compressed Document Array (GCDA), the proposal of Cobas and Navarro [4], which precomputes answers to document listing queries and stores them grammar compressed to reduce space. We set parameters $b$ and $\beta$ as described in [4].
- Sadakane (Sada). Sada-D is the index of Sadakane [22] in which the query time was sped up by explicitly storing DA, as first suggested in [5].
- Interleaved Longest Common Prefix (ILCP). ILCP-D is a variant of the ILCP index of Gagie et al. [5] that uses DA. ILCP-C uses, instead, Cobas and Navarro's [4] grammar-compressed DA, which can access any cell of DA in $O(h)$ time.

All indexes tested (including our own) use the same RLCSA implementation [12] to find the relevant interval $[s, e]$ of DA for each query pattern. This RLCSA uses $(r \log \sigma + 2r \log(n/r))(1 + o(1))$ bits of space, where $r$ is the number of runs in the BWT of T, and finds the interval in $O(m \log r)$ time. RLCSA for Page and Revision required 0.13 bits per symbol, and 0.26 bits per symbol for Influenza.

## 4.2 Results

Figure 4 shows the tradeoff between time and space for the indexes that were shown in [4] to be among fastest and smallest, and the ones described in this paper: Packed-bv, which was the second fastest among uncompressed versions (Figure 1) after Plain-bv (but uses much less space); and the two fastest variants for RLZD and RLED: -bv and -hash (Figure 3). RLZD on these datasets has shown the best compression on Page (0.341 for RLZD and 0.412 for Brute-C, which is the closest) and second best after Brute-C on Revision (0.812 for RLZD and 0.595 for Brute-C) and Influenza (2.958 for RLZD and 2.541 for Brute-C). These space results for RLZD were received with the following parameters to generate the reference: for Page $k = 312$, $s = 2048$, ref $= 1,987,456$ integers requested, for Revision $k = 8$, $s = 1024$, ref $= 9,096,048$, and for Influenza $k = 12$, $s = 256$, ref $= 15,310,000$. Ref values were computed experimentally. In terms of time RLZD-bv yields only to Packed-bv (and in half of the cases outperforms it), but the time difference is marginal and Packed-bv requires 6-20$\times$ more space.

RLED dominates on Page (where there are few and large documents), especially on high frequent patterns, where the closest competitor GCDA is 6 times slower. RLZD-bv in this case is 41 times slower than RLED-bv, but requires 3 times less space; compared to GCDA, RLZD-bv is almost 5 times slower, but needs almost half the space (GCDA takes 0.584 bits per symbol, RLZD 0.341). On the moderately frequent pattern set the time difference between RLZD-bv and GCDA becomes negligible, and RLZD-bv is starting to win on longer patterns.

On low frequent queries RLZD-bv becomes 3 times faster than GCDA, only 1.6 times slower than RLED-bv and is indeed only marginally slower than Packed-bv.

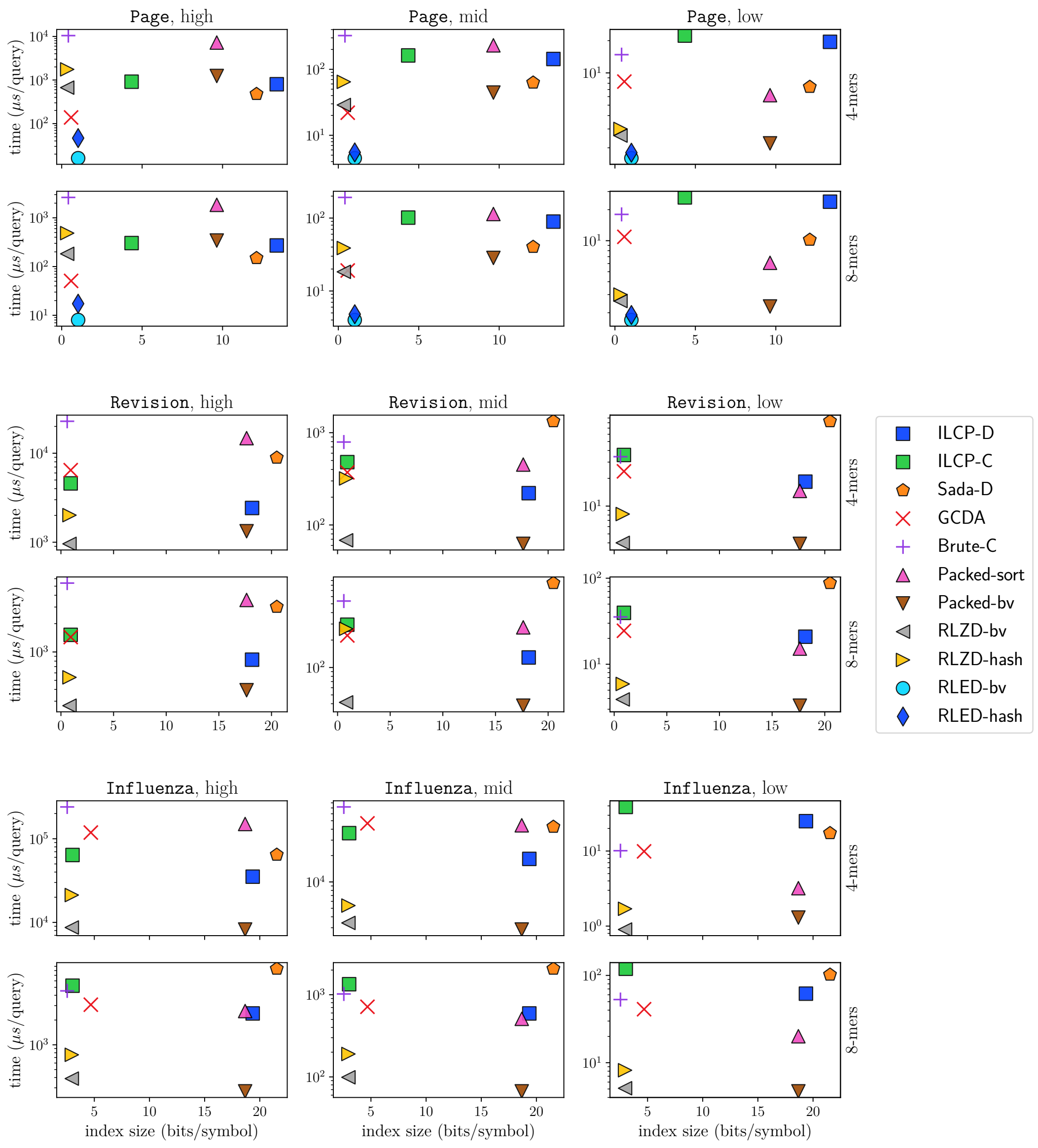


**Figure 4** Document listing indexes on real repetitive collections. The $x$ axis shows the total size of the index in bits per symbol. The $y$ axis shows the average time per query in microseconds. Note that the vertical axis is logarithmic.

In the case of Revision, where are more and smaller documents, RLZD-bv is about 1.4-1.5$\times$ faster than Packed-bv on frequent patterns, showing almost the same time results on middle and low frequent queries. ILCP-D is 2.5-3$\times$ slower on high frequent patterns (taking more than 22 times more space, requiring 18.155 bit per symbol), slowing down to 4.5-5$\times$ on low frequent. GCDA is 5.3-6.7$\times$ slower than RLZD-bv here. Brute-C with the index size of 1.36$\times$ smaller than RLZD, takes $8.5 - 23.8\times$ more time compared to RLZD-bv. Here RLZD-hash is 9.7-14$\times$ slower than RLZD-bv variant.

Influenza, with many small documents, is the worst case for many indexes. Here the smallest index, Brute-C, takes 1.16× less space than the nearest competitor, RLZD, and from 10.3 (on 8-mers with low frequency) upto 27.6 (on 4-mers with high frequency) more time than RLZD-bv. Packed-bv is 1.06 upto 1.42 times faster (271 microseconds for Packed-bv versus 388 for RLZD-bv on 8-mers with high frequency) and requires 6.3× more space. The other three nearest competitors: ILCP-D is 4-28× slower and as big as Packed-bv, GCDA is 8-13.6× slower, requiring 1.6 more space, and ILCP-C, which is third best space-wise, requires 1.14 more space (0.923 bits per symbol) and 4.7-10.2 more time.

In most cases the best previous indexes (from [4]) are either much slower, much larger, or both, compared to the ones we describe.

## 5 Conclusions and Future Work

We have shown that very simple algorithms based on scanning intervals of the document array lead to very fast document listing times on three highly repetitive data sets of versioned documents and genome collections. We have further shown these approaches to work well with new compressed representations of the document array based on relative Lempel-Ziv parsing and run-length encoding. We speculate that there are many ways to further engineer and improve the approaches we have described, however, our experiments here strongly indicate that they already exhibit significant performance improvements over existing methods. Our indexes are among the smallest document listing approaches known (at least on the data sets tested) and are an order of magnitude or faster.

There are numerous avenues for future work. Firstly, the results of this paper should lead directly to greatly improved performance for the metagenomics applications of document listing investigated in [3]. Extending our results to more complex versions of the document listing problem, such as document listing with frequencies and top-$k$ document retrieval should also be possible. It may also be fruitful to apply RLZ compression to improve other document listing methods, as was done with grammar compression by Cobas and Navarro [4].

Finally, it would seem important to determine the interval size at which specialized document listing methods begin to overhaul scanning-based methods. Such an investigation would necessarily involve indexing large test collections, which bring their own different challenges for the more intricate indexes, such as index construction. Our results here show that scanning-methods still have the edge for interval sizes into the millions.

## References

1 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

2 F. Claude and I. Munro. Document listing on versioned documents. In *Proc. SPIRE*, LNCS 8214, pages 72–83, 2013.

3 D. Cobas, V. Mäkinen, and M. Rossi. Tailoring r-index for document listing towards metagenomics applications. In *Proc. SPIRE*, LNCS 12303, pages 291–306. Springer, 2020.

4 D. Cobas and G. Navarro. Fast, small, and simple document listing on repetitive text collections. In *Proc. SPIRE*, LNCS 11811, pages 482–498, 2019.

5 T. Gagie, A. Hartikainen, K. Karhu, J. Kärkkäinen, G. Navarro, S. J. Puglisi, and J. Sirén. Document retrieval on repetitive collections. *Information Retrieval*, 20:253–291, 2017.

6 T. Gagie, S. J. Puglisi, and D. Valenzuela. Analyzing relative Lempel-Ziv reference construction. In *Proc. SPIRE*, LNCS 9954, pages 160–165, 2016.

7 S. Gog, R. Konow, and G. Navarro. Practical compact indexes for top-k document retrieval. *ACM Journal of Experimental Algorithmics*, 22(1):article 1.2, 2017.

8 W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-$k$ string retrieval problems. In *Proc. FOCS*, pages 713–722. IEEE, 2009.

9 C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment*, 5(3):265–273, 2011.

10 S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. SPIRE*, LNCS 6393, pages 201–206, 2010.

11 K. Liao, M. Petri, A. Moffat, and A. Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proc. WWW*, pages 807–816. ACM, 2016.

12 V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

13 U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

14 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

15 G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.

16 G. Navarro. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science*, 777:58–72, 2019.

17 G. Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 2020. To appear.

18 G. Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

19 G. Navarro, S. J. Puglisi, and D. Valenzuela. General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, 19(2):article 3, 2014.

20 S. J. Puglisi and B. Zhukova. Relative Lempel-Ziv compression of suffix arrays. In *Proc. SPIRE*, LNCS 12303, pages 89–96. Springer, 2020.

21 S. J. Puglisi and B. Zhukova. Smaller RLZ-compressed suffix arrays. In *Proc. Data Compression Conference*, pages 213–222. IEEE Computer Society, 2021.

22 K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5:12–22, 2007.

23 J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proc. SIGIR*, pages 283–292. ACM, 2014.

24 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

# O'Reach:
# Even Faster Reachability in Large Graphs

**Kathrin Hanauer**
University of Vienna, Faculty of Computer Science, Austria

**Christian Schulz**
Heidelberg University, Germany

**Jonathan Trummer**
University of Vienna, Faculty of Computer Science, Austria

**Abstract**

One of the most fundamental problems in computer science is the *reachability problem*: Given a directed graph and two vertices $s$ and $t$, can $s$ *reach* $t$ via a path? We revisit existing techniques and combine them with new approaches to support a large portion of *reachability queries* in constant time using a linear-sized *reachability index*. Our new algorithm `O'Reach` can be easily combined with previously developed solutions for the problem or run standalone.

In a detailed experimental study, we compare a variety of algorithms with respect to their index-building and query times as well as their memory footprint on a diverse set of instances. Our experiments indicate that the query performance often depends strongly not only on the type of graph, but also on the result, i.e., *reachable* or *unreachable*. Furthermore, we show that previous algorithms are significantly sped up when combined with our new approach in almost all scenarios. Surprisingly, due to cache effects, a higher investment in space doesn't necessarily pay off: *Reachability queries* can often be answered even faster than single memory accesses in a precomputed full reachability matrix.

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

**Keywords and phrases** Reachability, Static Graphs, Graph Algorithms, Reachability Index, Algorithm Engineering

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.13

**Related Version** *Full Version*: `https://arxiv.org/abs/2008.10932` [12]

**Supplementary Material** *Software (Code Repository)*: `https://github.com/o-reach/O-Reach`
archived at `swh:1:dir:55d23a5b940f1ead285729c8dbd82c71e28d504a`

**Funding** The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

## 1 Introduction

Graphs are used to model problem settings of various different disciplines. A natural question that arises frequently is whether one vertex of the graph can *reach* another vertex via a path of directed edges. *Reachability* finds application in a wide variety of fields, such as program and dataflow analysis [24, 25], user-input dependence analysis [27], XML query processing [34], and more [40]. Another prominent example is the Semantic Web which is composed of RDF/OWL data. These are often very huge graphs with rich content. Here, reachability queries are often necessary to deduce relationships among the objects.

There are two straightforward solutions to the reachability problem: The first is to answer each query individually with a graph traversal algorithm, such as breadth-first search (BFS) or depth-first search (DFS), in worst-case $\mathcal{O}(m + n)$ time and $\mathcal{O}(n)$ space. Secondly, we can precompute a full all-pairs reachability matrix in an initialization step and answer all ensuing queries in worst-case constant time. In return, this approach suffers from a space complexity of $\mathcal{O}(n^2)$ and an initialization time of $\mathcal{O}(n \cdot m)$ using the Floyd-Warshall algorithm [7, 35, 6] or starting a graph traversal at each vertex in turn. Alternatively, the initialization step can be performed in $\mathcal{O}(n^\omega)$ via fast matrix multiplication, where $\mathcal{O}(n^\omega)$ is the time required to multiply two $n \times n$ matrices ($2 \leq \omega < 2.38$ [20]). With increasing graph size, however, both the initialization time and space complexity of this approach become impractical. We therefore strive for alternative algorithms which decrease these complexities whilst still providing fast query lookups.

**Contribution.** In this paper, we study a variety of approaches that are able to support fast *reachability queries*. All of these algorithms perform some kind of preprocessing on the graph and then use the collected data to answer reachability queries in a timely manner. Based on simple observations, we provide a new algorithm, `O'Reach`, that can improve the query time for a wide range of cases over state-of-the-art reachability algorithms at the expense of some additional precomputation time and space or be run standalone. Furthermore, we show that previous algorithms are significantly sped up when combined with our new approach in almost all scenarios. In addition, we show that the expected query performance of various algorithms does not only depend on the type of graph, but also on the ratio of successful queries, i.e., with result *reachable*. Surprisingly, through cache effects and a significantly smaller memory footprint, especially unsuccessful *reachability queries* can be answered faster than single memory accesses in a precomputed reachability matrix.

## 2 Preliminaries

**Terms and Definitions.** Let $G = (V, E)$ be a simple directed graph with vertex set $V$ and edge set $E \subseteq V \times V$. As usual, $n = |V|$ and $m = |E|$. An edge $(u, v)$ is said to be *outgoing* at $u$ and *incoming* at $v$, and $u$ and $v$ are called *adjacent*. The *out-degree* $\deg^+(u)$ (*in-degree* $\deg^-(u)$) of a vertex $u$ is its number of outgoing (incoming) edges. A vertex without incoming (outgoing) edges is called a *source* (*sink*). The *out-neighborhood* $\mathsf{N}^+(v)$ (*in-neighborhood* $\mathsf{N}^-(v)$) of a vertex $u$ is the set of all vertices $v$ such that $(u, v) \in E$ ($(v, u) \in E$). The *reverse* of an edge $(u, v)$ is an edge $(v, u) = (u, v)^{\mathrm{R}}$. The *reverse* $G^{\mathrm{R}}$ of a graph $G$ is obtained by keeping the vertices of $G$, but substituting each edge $(u, v) \in E$ by its reverse, i.e., $G^{\mathrm{R}} = (V, E^{\mathrm{R}})$.

A sequence of vertices $s = v_0 \to \cdots \to v_k = t$, $k \geq 0$, such that for each pair of consecutive vertices $v_i \to v_{i+1}$, $(v_i, v_{i+1}) \in E$, is called an *s-t path*. If such a path exists, $s$ is said to *reach* $t$ and we write $s \to^* t$ for short, and $s \not\to^* t$ otherwise. The *out-reachability* $\mathsf{R}^+(u) = \{v \mid u \to^* v\}$ (*in-reachability* $\mathsf{R}^-(u) = \{v \mid v \to^* u\}$) of a vertex $u \in V$ is the set of all vertices that $u$ can reach (that can reach $u$).

A *weakly connected component (WCC)* of $G$ is a maximal set of vertices $C \subseteq V$ such that $\forall u, v \in C : u \to^* v$ in $G = (V, E \cup E^{\mathrm{R}})$, i.e., also using the reverse of edges. Note that if two vertices $u, v$ reside in different WCCs, then $u \not\to^* v$ and $v \not\to^* u$. A *strongly connected component (SCC)* of $G$ denotes a maximal set of vertices $S \subseteq V$ such that $\forall u, v \in S : u \to^* v \wedge v \to^* u$ in $G$. Contracting each SCC $S$ of $G$ to a single vertex $v_S$, called its *representative*, while preserving edges between different SCCs as edges between their corresponding representatives, yields the *condensation* $G^{\mathrm{C}}$ of $G$. We denote the SCC a vertex $v \in V$ belongs to by $\mathcal{S}(v)$. A directed graph $G$ is *strongly connected* if it only has a single SCC and *acyclic* if each SCC is a singleton, i.e., if $G$ has $n$ SCCs. Observe that $G$ and

$G^{\mathrm{R}}$ have exactly the same WCCs and SCCs and that $G^{\mathrm{C}}$ is a directed acyclic graph (DAG). Weakly connected components of a graph can be computed in $\mathcal{O}(n+m)$ time, e.g., via a breadth-first search that ignores edge directions. The strongly connected components of a graph can be computed in linear time [29] as well.

A *topological ordering* $\tau : V \to \mathbb{N}_0$ of a DAG $G$ is a total ordering of its vertices such that $\forall(u,v) \in E : \tau(u) < \tau(v)$. Note that the topological ordering of $G$ isn't necessarily unique, i.e., there can be multiple different topological orderings. For a vertex $u \in V$, the *forward topological level* $\mathcal{F}(u) = \min_\tau \tau(u)$, i.e., the minimum value of $\tau(u)$ among all topological orderings $\tau$ of $G$. Consequently, $\mathcal{F}(u) = 0$ if and only if $u$ is a source. The *backward topological level* $\mathcal{B}(u)$ of $u \in V$ is the topological level of $u$ with respect to $G^{\mathrm{R}}$ and $\mathcal{B}(u) = 0$ if and only if $u$ is a sink. A topological ordering as well as the forward and backward topological levels can be computed in linear time [19, 30, 6], see also Sect. 4.

A *reachability query* Query$(s,t)$ for a pair of vertices $s,t \in V$ is called *positive* and answered with `true` if $s \to^* t$, and otherwise *negative* and answered with `false`. Trivially, Query$(v,v)$ is always `true`, which is why we only consider *non-trivial* queries between distinct vertices $s \neq t \in V$ from here on. Let $\mathcal{P}$ ($\mathcal{N}$) denote the set of all positive (negative) non-trivial queries of $G$, i.e., the set of all $(s,t) \in V \times V$, $s \neq t$, such that Query$(s,t)$ is positive (negative). The *reachability* $\rho$ in $G$ is the ratio of positive queries among all non-trivial queries, i.e., $\rho = \frac{|\mathcal{P}|}{n(n-1)}$. Note, that due to the restriction to non-trivial queries[1], $0 \leq \rho \leq 1$. The *Reachability problem*, studied in this paper, consists in answering a sequence of reachability queries for arbitrary pairs of vertices on a given input graph $G$.

**Basic Observations.** With respect to processing a reachability Query$(s,t)$ in a graph $G$ for an arbitrary pair of vertices $s \neq t \in V$, the following basic observations are immediate and have partially also been noted elsewhere [22]:

**(B1)** If $s$ is a sink or $t$ is a source, then $s \not\to^* t$.
**(B2)** If $s$ and $t$ belong to different WCCs of $G$, then $s \not\to^* t$.
**(B3)** If $s$ and $t$ belong to the same SCC of $G$, then $s \to^* t$.
**(B4)** If $\tau(\mathcal{S}(t)) < \tau(\mathcal{S}(s))$ for any topological ordering $\tau$ of $G^{\mathrm{C}}$, then $s \not\to^* t$.

As mentioned above, the precomputations necessary for Observations (B2) and (B3) can be performed in $\mathcal{O}(n+m)$ time. Note, however, that Observations (B3) and (B4) together are *equivalent* to asking whether $s \to^* t$: If $s \to^* t$ and $\mathcal{S}(s) \neq \mathcal{S}(t)$, then for every topological ordering $\tau$, $\tau(\mathcal{S}(s)) < \tau(\mathcal{S}(t))$. Otherwise, if $s \not\to^* t$, a topological ordering $\tau$ with $\tau(\mathcal{S}(t)) < \tau(\mathcal{S}(s))$ can be computed by topologically sorting $G^{\mathrm{C}} \cup \{(\mathcal{S}(t), \mathcal{S}(s))\}$. Hence, the precomputations necessary for Observation (B4) would require solving the *Reachability* problem for all pairs of vertices already. Furthermore, a DAG can have exponentially many different topological orderings. In consequence, weaker forms are employed, such as the following [38, 39, 22] (see also Sect. 4):

**(B5)** If $\mathcal{F}(\mathcal{S}(t)) < \mathcal{F}(\mathcal{S}(s))$ w.r.t. $G^{\mathrm{C}}$, then $s \not\to^* t$.
**(B6)** If $\mathcal{B}(\mathcal{S}(s) < \mathcal{B}(\mathcal{S}(t))$ w.r.t. $G^{\mathrm{C}}$, then $s \not\to^* t$.

**Assumptions.** Following the convention introduced in preceding work [38, 39, 3, 22] (cf. Sect. 3), we only consider *Reachability* on DAGs from here on and implicitly assume that the condensation, if necessary, has already been computed and Observation (B3) has been applied. For better readability, we also drop the use of $\mathcal{S}(\cdot)$.

[1] Otherwise, $\frac{1}{n} \leq \rho$.

**Table 1** Time and space complexity of reachability algorithms. Parameters: $k_{\texttt{IP}}$: #permutations, $h_{\texttt{IP}}$: #vertices with precomputed $\mathsf{R}^+(\cdot)$, $s_{\texttt{BFL}}$: size of Bloom filter (bits), $\rho$: reachability in $G$, $d$: #topological orderings, $k$: #supportive vertices, $p$: #candidates per supportive vertex.

| Algorithm | Initialization Time | Index Size (Byte) | Queries: Time | Space |
|---|---|---|---|---|
| BFS/DFS | $\mathcal{O}(1)$ | 0 | $\mathcal{O}(n+m)$ | $\mathcal{O}(n)$ |
| Full matrix | $\mathcal{O}(n \cdot (n+m))$ | $n^2/8$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| `PPL` [37] | $\mathcal{O}(n \log n + m)$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| `PReaCH` [22] | $\mathcal{O}(m + n \log n)$ | $56n$ | $\mathcal{O}(1)$ / $\mathcal{O}(n+m)$ | $\mathcal{O}(n)$ |
| `IP`($k_{\texttt{IP}}$, $h_{\texttt{IP}}$) [36] | $\mathcal{O}((k_{\texttt{IP}} + h_{\texttt{IP}})(n+m))$ | $\mathcal{O}((k_{\texttt{IP}} + h_{\texttt{IP}})n)$ | $\mathcal{O}(k_{\texttt{IP}})$ / $\mathcal{O}(k_{\texttt{IP}} \cdot n \cdot \rho^2)$ | $\mathcal{O}(n)$ |
| `BFL`($s_{\texttt{BFL}}$) [28] | $\mathcal{O}(s_{\texttt{BFL}} \cdot (n+m))$ | $2\lceil \frac{s_{\texttt{BFL}}}{8} \rceil n$ | $\mathcal{O}(s_{\texttt{BFL}})$ / $\mathcal{O}(s_{\texttt{BFL}} \cdot n + m)$ | $\mathcal{O}(n)$ |
| `O'Reach`($d, k, p$) (Sect. 4) | $\mathcal{O}((d+kp)(n+m))$ | $(12 + 12d + 2\lceil \frac{k}{8} \rceil)n$ | $\mathcal{O}(k+d+1)$ / $\mathcal{O}(n+m)$ | $\mathcal{O}(n)$ |

## 3 Related Work

A large amount of research on reachability indices has been conducted. Existing approaches can roughly be put into three categories: compression of transitive closure [14, 13, 2, 34, 15, 32], hop-labeling-based algorithms [5, 4, 26, 37, 16], as well as pruned search [18, 31, 38, 39, 22, 33, 36, 28]. As Merz and Sanders [22] noted, the first category gives very good query times for small networks, but doesn't scale very well to large networks (which is the focus of this work). Therefore, we do not consider approaches based on this technique more closely. Hop labeling algorithms typically build paths from labels that are stored for each vertex. For example in 2-hop labeling, each vertex stores two sets containing vertices it can reach in the given graph as well as in the reverse graph. A query can then be reduced to the set intersection problem. Pruned-search-based approaches precompute information to speed up queries by pruning the search.

Due to its volume, it is impossible to compare against all previous work. We mostly follow the methodology of Merz and Sanders [22] and focus on five recent techniques. The two most recent hop-labeling-based approaches are `TF` [3] and `PPL` [37]. In the pruned search category, the three most recent approaches are `PReaCH` [22], `IP` [36], and `BFL` [28]. We now go into more detail:

**`TF`.** The work by Cheng et al. [3] uses a data structure called topological folding. On the condensation DAG, the authors define a topological structure that is obtained by recursively folding the structure in half each time. Using this topological structure, the authors create labels that help to quickly answer reachability queries.

**`PPL`.** Yano et al. [37] use pruned landmark labeling and pruned path labeling as labels for their reachability queries. In general, the method follows the 2-hop labeling technique mentioned above, which stores sets of vertices for each vertex $v$ and reduces queries to the set intersection problem. Their techniques are able to reduce the size of the stored labels and hence to improve query time and space consumption.

**`PReaCH`.** Merz and Sanders [22] apply the approach of *contraction hierarchies* (CHs) [9, 10] known from shortest-path queries to the reachability problem. The method first tries to answer queries by using pruning and precomputed information such as topological levels (Observation (B5) and (B6)). It adopts and improves techniques from `GRAIL` [38, 39] for that task, which is distinctly outperformed by `PReaCH` in the subsequent experiments. Should these techniques not answer the query, `PReaCH` instead performs a bidirectional breadth-first search (BFS) using the computed hierarchy, i.e., for a QUERY$(s,t)$ the BFS only considers neighboring vertices with larger topological level and along the CH. The overall approach is simple and guarantees linear space and near linear preprocessing time.

**IP.** Wei et al. [36] use a randomized labeling approach by applying independent permutations on the labels. Contrary to other labeling approaches, IP checks for set-containment instead of set-intersection. Therefore, IP tries to answer negative queries by checking for at least one vertex that it is contained in only one of the two sets, where each set can consist of at most $k_{\mathtt{IP}}$ vertices. If this test fails, IP checks another label, which contains precomputed reachability information from the $h_{\mathtt{IP}}$ vertices with largest out-degree, and otherwise falls back to depth-first-search (DFS).

**BFL.** Su et al. [28] propose a labeling method which is based on IP, but additionally uses Bloom filters for storing and comparing labels, which are then used to answer negative queries. As parameters, BFL accepts $s_{\mathtt{BFL}}$ and $d_{\mathtt{BFL}}$, where $s_{\mathtt{BFL}}$ denotes the length of the Bloom filters stored for each vertex and $d_{\mathtt{BFL}}$ controls the false positive rate. By default, $d_{\mathtt{BFL}} = 10 \cdot s_{\mathtt{BFL}}$.

Table 1 subsumes the time and space complexities of the new algorithm O'Reach that we introduce in Sect. 4 as well as all algorithms mentioned in this paper except for TF, where the expressions describing the theoretical complexities are bulky and quite complex themselves.

## 4 O'Reach: Faster Reachability via Observations

In this section we propose our new algorithm O'Reach, which is based on a set of simple, yet powerful observations that enable us to answer a large proportion of reachability queries in constant time and brings together techniques from both hop labeling and pruned search. Unlike regular hop-labeling-approaches, however, its initialization time is linear. As a further plus, our algorithm is configurable via multiple parameters and extremely space-efficient with an index of only 38n Byte in the most space-saving configuration that could handle all instances used in Sect. 5 and uses all features.

**Overview.** The hop labeling technique used in our algorithm is inspired by a recent result for experimentally faster reachability queries in a dynamic graph by Hanauer et al. [11]. The idea here is to speed up reachability queries based on a selected set of so-called *supportive vertices*, for which complete out- and in-reachability is maintained explicitly. This information is used in three simple observations, which allow to answer matching queries in constant time. In our algorithm, we transfer this idea to the static setting. We further increase the ratio of queries answerable in constant time by a new perspective on topological orderings and their conflation with depth-first search, which provides additional reachability information and further increases the ratio of queries answerable in constant time. In case that we cannot answer a query via an observation, we fall back to either a pruning bidirectional breadth-first search or one of the existing algorithms.

In the following, we switch the order and first discuss topological orderings in depth, followed by our adaptation of supportive vertices. For both parts, consider a reachability $\textsc{Query}(s, t)$ for two vertices $s, t \in V$ with $s \neq t$.

### 4.1 Extended Topological Orderings

Taking up on the observation that topological orderings can be used to answer a reachability query decisively negative, we first investigate how Observation (B4) can be used most effectively in practice. Before we dive deeper into this subject, let us briefly review some facts concerning topological orderings and reachability in general.

▶ **Theorem 1.** *Let $\mathcal{N}(\tau) \subseteq \mathcal{N}$ denote the set of negative queries a topological ordering $\tau$ can answer, i.e., the set of all $(s,t) \in \mathcal{N}$ such that $\tau(t) < \tau(s)$, and let $\rho^-(\tau) = \mathcal{N}(\tau)/\mathcal{N}$ be the answerable negative query ratio.*

**(i)** *The reachability in any DAG is at most 50%. In this case, the topological ordering is unique.*

**(ii)** *Any topological ordering $\tau$ witnesses the non-reachability between exactly 50% of all pairs of distinct vertices. Therefore, $\rho^-(\tau) \geq 50\%$.*

**(iii)** *Every topological ordering of the same DAG can answer the same ratio of all negative queries via Observation (B4), i.e., for two topological orderings $\tau$, $\tau'$: $\rho^-(\tau) = \rho^-(\tau')$.*

**(iv)** *For two different topological orderings $\tau \neq \tau'$ of a DAG, $\mathcal{N}(\tau) \neq \mathcal{N}(\tau')$.*

**Proof.** Let $G$ be a directed acyclic graph (DAG).

(i) As $G$ is acyclic, there is at least one topological ordering $\tau$ of $G$. Then, for every edge $(u,v)$ of $G$, $\tau(u) < \tau(v)$, which implies that each vertex $u$ can reach at most all those vertices $w \neq u$ with $\tau(u) < \tau(w)$. Consequently, a vertex $u$ with $\tau(u) = i$ can reach at most $n - i - 1$ *other* vertices (note that $i \geq 0$). Thus, the reachability in $G$ is at most $\frac{1}{n(n-1)} \sum_{i=0}^{n-1} (n - i - 1) = \frac{1}{n(n-1)} \sum_{j=0}^{n-1} j = \frac{n(n-1)}{n(n-1)\cdot 2} = \frac{1}{2}$. Conversely, assume that the reachability in $G$ is $\frac{1}{2}$. Then, each vertex $u$ with $\tau(u) = i$ reaches exactly all $n-i-1$ other vertices ordered after it, which implies that there exists no other topological ordering $\tau'$ with $\tau'(u) > \tau(u)$. By induction on $i$, the topological ordering of $G$ is unique.

(ii) Let $\tau$ be an arbitrary topological ordering of $G$. Then, each vertex $u$ with $\tau(u) = i$ can certainly *reach* those vertices $v$ with $\tau(v) < \tau(u)$. Hence, $\tau$ witnesses the non-reachability of exactly $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ pairs of distinct vertices.

(iii) As Observation (B4) corresponds exactly to the non-reachability between those pairs of vertices witnessed by the topological ordering, the claim follows directly from (ii).

(iv) As $\tau \neq \tau'$, there is at least one $i \in \mathbb{N}_0$ such that $\tau(u) = i = \tau'(v)$ and $u \neq v$. Let $j = \tau(v)$. If $j > i$, the number of non-reachabilities from $v$ to another vertex witnessed by $\tau$ exceeds the number of those witnessed by $\tau'$, and falls behind it otherwise. In both cases, the difference in numbers immediately implies a difference in the set of vertex pairs, which proves the claim. ◀

In consequence, it is pointless to look for one particularly good topological ordering. Instead, to get the most out of Observation (B4), we need topological orderings whose sets of answerable negative queries differ greatly, such that their union covers a large fraction of $\mathcal{N}$. Note that both forward and backward topological levels each represent the set of topological orderings that can be obtained by ordering the vertices in blocks grouped by their level and arbitrarily permuting the vertices in each block. Different algorithms [19, 29, 6] for computing a topological ordering in linear time have been proposed over the years, with Kahn's algorithm [19] in combination with a queue being one that always yields a topological ordering represented by forward topological levels. We therefore complement the forward and backward topological levels by stack-based approaches, as in Kahn's algorithm [19] in combination with a stack or Tarjan's DFS-based algorithm [29] for computing the SCCs of a graph, which as a by-product also yields a topological ordering of the condensation. To diversify the set of answerable negative queries further, we additionally randomize the order in which vertices are processed in case of ties and also compute topological orderings on the reverse graph, in analogy to backward topological levels.

We next show how, with a small extension, the stack-based topological orderings mentioned above can be used to additionally answer positive queries. To keep the description concise, we concentrate on Tarjan's algorithm [29] in the following and reduce it to the part relevant for obtaining a topological ordering of a DAG. In short, the algorithm starts a depth-first

(a)

```
procedure ExtendedTopSort($G = (V, E), S$)
    for all $v \in V$ do $v$.visited ← false
    $i \leftarrow n - 1$ ;   initialize $\tau$, $\tau_H$, $\tau_X$ empty
    for all $s \in S$ in random order do Visit($s$)
    procedure Visit($v$)
        if $v$.visited then return
        $v$.visited ← true;   $\tau_H(v) \leftarrow i$;   $\tau_X(v) \leftarrow i$
        for $u \in \mathsf{N}^+(v)$ in random order do
            Visit($u$);   $\tau_X(v) \leftarrow \max(\tau_X(v), \tau_X(u))$
        $\tau(v) \leftarrow i$ ;   $i \leftarrow i - 1$
    return $\tau$, $\tau_H$, $\tau_X$
```

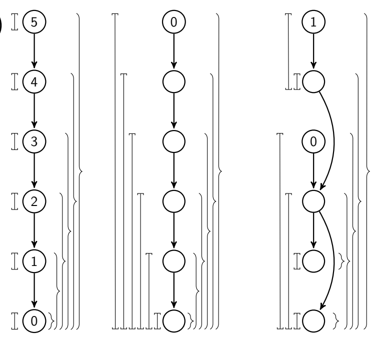
(b) 

**Figure 1** (a): Extended Topological Sorting. (b): Three extended topological orderings of two graphs: The labels correspond to the order in the start set $S$. If the label is empty, the vertex need not be in $S$ or can have any larger number. The brackets to the left show the range $[\tau(v), \tau_H(v)]$, the braces to the right the range $[\tau(v), \tau_X(v)]$.

search at an arbitrary vertex $s \in S$, where $S \subseteq V$ is a given set of vertices to start from. Whenever it visits a vertex $v$, it marks $v$ as visited and recursively visits all unvisited vertices in its out-neighborhood. On return, it *prepends* $v$ to the topological ordering. A loop over $S = V$ ensures that all vertices are visited. Note that although the vertices are visited in DFS order, the topological ordering is different from a DFS numbering as it is constructed "from back to front" and corresponds to a reverse sorting according to what is also called *finishing time* of each vertex.

To answer positive queries, we exploit the invariant that when visiting a vertex $v$, all yet unvisited vertices reachable from $v$ will be prepended to the topological ordering prior to $v$ being prepended. Consequently, $v$ can *certainly* reach all vertices in the topological ordering between $v$ and, exclusively, the vertex $w$ that was at the front of the topological ordering when $v$ was visited. Let $x$ denote the vertex preceding $w$ in the final topological ordering, i.e., the vertex with the largest index that was reached recursively from $v$. For a topological ordering $\tau$ constructed in this way, we call $\tau(x)$ the *high index* of $v$ and denote it with $\tau_H(v)$. Furthermore, $v$ *may* be able to also reach $w$ and vertices beyond, which occurs if $v \rightarrow^* y$ for some vertex $y$, but $y$ had already been visited earlier. We therefore additionally track the *max index*, the largest index of any vertex that $v$ can reach, and denote it with $\tau_X(v)$. Figure 1a shows how to compute an extended topological ordering with both high and max indices in pseudo code and highlights our extensions. Compared to Tarjan's original version [29], the running time remains unaffected by our modifications and is still in $\mathcal{O}(n + m)$.

Note that neither max nor high indices yield an ordering of $V$: Every vertex that is visited recursively starting from $v$ and before vertex $x$ with $\tau(x) = \tau_H(v)$, inclusively, has the same high index as $v$, and the high index of each vertex in a graph consisting of a single path, e.g., would be $n - 1$. In particular, neither max nor high index form a DFS numbering and also differ in definition and use from the DFS finishing times $\hat{\phi}$ used in `PReaCH`, where a vertex $v$ can *certainly* reach vertices with DFS number up to $\hat{\phi}$ and *certainly none* beyond. Conversely, $v$ may be able to also reach vertices with smaller DFS number than its own, which cannot occur in a topological ordering.

If ExtendedTopSort is run on the reverse graph, it yields a topological ordering $\tau'$ and high and max indices $\tau'_H$ and $\tau'_X$, such that reversing $\tau'$ yields again a topological ordering $\tau$ of the original graph. Furthermore, $\tau_L(v) := n - 1 - \tau'_H(v)$ is a *low index* for each vertex $v$, which denotes the smallest index of a vertex in $\tau$ that can certainly reach $v$, i.e., the out-reachability of $v$ is replaced by in-reachability. Analogously, $\tau_N(v) := n - 1 - \tau'_X(v)$ is a *min index* in $\tau$ and no vertex $u$ with $\tau(u) < \tau_N(v)$ can reach $v$.

The following observations show how such an extended topological ordering $\tau$ can be used to answer both positive and negative reachability queries:

**(T1)** If $\tau(s) \leq \tau(t) \leq \tau_H(s)$, then $s \to^* t$. **(T4)** If $\tau_L(t) \leq \tau(s) \leq \tau(t)$, then $s \to^* t$.
**(T2)** If $\tau(t) > \tau_X(s)$, then $s \not\to^* t$. **(T5)** If $\tau(s) < \tau_N(t)$, then $s \not\to^* t$.
**(T3)** If $\tau(t) = \tau_X(s)$, then $s \to^* t$. **(T6)** If $\tau(s) = \tau_N(t)$, then $s \to^* t$.

Recall that by definition, $\tau(s) \leq \tau_H(s) \leq \tau_X(s)$ and $\tau_N(t) \leq \tau_L(t) \leq \tau(t)$. Figure 1b depicts three examples for extended topological orderings. In contrast to negative queries, not every extended topological ordering is equally effective in answering positive queries, and it can be arbitrarily bad, as shown in the extremes on the left (worst) and at the center (best) of Figure 1b:

▶ **Theorem 2.** *Let $\mathcal{P}(\tau) \subseteq \mathcal{P}$ be the set of positive queries an extended topological ordering $\tau$ can answer and let $\rho^+(\tau) = \mathcal{P}(\tau)/\mathcal{P}$ be the answerable positive query ratio. Then,* $0 \leq \rho^+(\tau) \leq 1$.

Instead, the effectiveness of an extended topological ordering depends positively on the size of the ranges $[\tau(v), \tau_H(v)]$ and $[\tau_L(v), \tau(v)]$, and negatively on $[\tau_H(v), \tau_X(v)]$ and $[\tau_N(v), \tau_L(v)]$ which in turn depend on the recursion depths during construction and the order of recursive calls. The former two can be maximized if the first, non-recursive call to Visit in line 4 in ExtendedTopSort always has a source as its argument, i.e., if the algorithm's parameter $S$ corresponds to the set of all sources. Clearly, this still guarantees that every vertex is visited.

In addition to the forward and backward topological levels, `O'Reach` thus computes a set of $d$ extended topological orderings starting from sources, where $d$ is a tuning parameter, and $d/2$ of them are obtained via the reverse graph. It then applies Observation (B4) as well as Observations (T1)–(T6) to all extended topological orderings.

## 4.2 Supportive Vertices

We now show how to apply and improve the idea of supportive vertices in the static setting. A vertex $v$ is *supportive* if the set of vertices that $v$ can reach and that can reach $v$, $R^+(v)$ and $R^-(v)$, respectively, have been precomputed and membership queries can be performed in sublinear time. We can then answer reachability queries using the following simple observations [11]:

**(S1)** If $s \in R^-(v)$ and $t \in R^+(v)$ for *any* $v \in V$, then $s \to^* t$.
**(S2)** If $s \in R^+(v)$ and $t \notin R^+(v)$ for *any* $v \in V$, then $s \not\to^* t$.
**(S3)** If $s \notin R^-(v)$ and $t \in R^-(v)$ for *any* $v \in V$, then $s \not\to^* t$.

To apply these observations, our algorithm selects a set of $k$ supportive vertices during the initialization phase. In contrast to the original use scenario in the dynamic setting, where the graph changes over time and it is difficult to choose "good" supportive vertices that can help to answer many queries, the static setting leaves room for further optimizations here: With respect to Observation (S1), we consider a supportive vertex $v$ "good" if $|R^+(v)| \cdot |R^-(v)|$ is large as it maximizes the possibility that $s \in R^-(v) \wedge t \in R^+(v)$. With respect to Observation (S2) and (S3), we expect a "good" supportive vertex to have out- or in-reachability sets, respectively, of size close to $\frac{n}{2}$, i.e., when $|R^+(v)| \cdot |V \setminus R^+(v)|$ or $|R^-(v)| \cdot |V \setminus R^-(v)|$, respectively, are maximal. Furthermore, to increase total coverage and avoid redundancy, the set of queries Query$(s,t)$ covered by two different supportive vertices should ideally overlap as little as possible.

`O'Reach` takes a parameter $k$ specifying the number of supportive vertices to pick. Intuitively speaking, we expect vertices in the topological "mid-levels" to be better candidates than those at the ends, as their out- and in-reachabilities (or non-reachabilities) are likely to be more balanced. Furthermore, if *all* vertices on one forward (backward) level $i$ were supportive, then *every* $\textsc{Query}(s,t)$ with $\mathcal{F}(s) < i < \mathcal{F}(t)$ ($\mathcal{B}(t) < i < \mathcal{B}(s)$) could be answered using only Observation (S1). As finding a "perfect" set of supportive vertices is computationally expensive and we strive for linear preprocessing time, we experimentally evaluated different strategies for the selection process. Due to page limits, we only describe the most successful one: A forward (backward) level $i$ is called *central*, if $\frac{1}{5}L_{\max} \leq i \leq \frac{4}{5}L_{\max}$, where $L_{\max}$ is the maximum topological level. A level $i$ is called *slim* if there at most $h$ vertices having this level, where $h$ is a parameter to `O'Reach`. We first compute a set of candidates of size at most $k \cdot p$ that contains all vertices on slim forward or backward levels, arbitrarily discarding vertices as soon as the threshold $k \cdot p$ is reached. $p$ is another parameter to `O'Reach` and together with $k$ controls the size of the candidate set. If the threshold is not reached, we fill up the set of candidates by picking the missing number of vertices uniformly at random from all other vertices whose forward level is central. In the next step, the out- and in-reachabilities of all candidates are obtained and the $k$ vertices $v$ with largest $|R^{+}(v)| \cdot |R^{-}(v)|$ are chosen as supportive vertices. This strategy primarily optimizes for Observation (S1), but worked better in experiments than strategies that additionally tried to optimize for Observation (S2) and (S3). The time complexity of this process is in $\mathcal{O}(kp(n+m) + kp\log(kp))$.

We remark that this is a general-purpose approach that has shown to work well across different types of instance, albeit possibly at the expense of an increased initialization time. It seems natural that more specialized routines for different graph classes can improve both running time and coverage.

## 4.3 The Complete Algorithm

Given a graph $G$ and a sequence of queries $Q$, we summarize in the following how `O'Reach` proceeds. During initialization, it performs the following steps:

**Step 1:** Compute the WCCs
**Step 2:** Compute forward/backward topological levels
**Step 3:** Obtain $d$ random extended topological orderings
**Step 4:** Pick $k$ supportive vertices, compute $\mathsf{R}^{+}(\cdot)$ and $\mathsf{R}^{-}(\cdot)$

Steps 1 and 2 run in linear time. As shown in Sect. 4.1 and Sect. 4.2, the same applies to Steps 3 and 4, assuming that all parameters are constants. The required space is linear for all steps. The reachability index consists of the following information for each vertex $v$: one integer for the WCC, one integer each for $\mathcal{F}(v)$ and $\mathcal{B}(v)$, three integers for each of the $d$ extended topological orderings $\tau$ ($\tau(v), \tau_H(v)/\tau_L(v), \tau_X(v)/\tau_N(v)$), two bits for each of the $k$ supportive vertices, indicating its reachability to/from $v$. For graphs with $n \leq 2^{32}$, 4 Byte per integer suffice. Furthermore, we group the bits encoding the reachabilities to and from the supportive vertices, respectively, and represent them each by one suitably sized integer, e.g., using `uint8_t` (8 bit), for $k \leq 8$ supportive vertices. As the smallest integer has at least 8 bit on most architectures, we store $12 + 12\mathrm{d} + 2 \cdot \lceil \frac{\mathrm{k}}{8} \rceil$ Byte per vertex.

For each query $\textsc{Query}(s,t)$, `O'Reach` tries to answer it using one of the observations in the order given below, which on the one hand has been optimized by some preliminary experiments on a small subset of benchmark instances (see Sect. 5 for details) and on the other hand strives for a fair alternation between "positive" and "negative" observations to avoid overfitting. Note that all observation-based tests run in constant time. As soon as one of them can answer the query affirmatively, the result is returned immediately. A test leading to a positive or negative answer is marked as ○ or ●, respectively.

**Test 1:** ○ $s = t$?

**Test 2:** ● ● topological levels (B5), (B6)

**Test 3:** ○ $k$ supportive vertices, positive (S1)

**Test 4:** ● ○ ● ○ first topological ordering (B4), (T1), (T2), (T3)

**Test 5:** ● ● $k$ supportive vertices, negative (S2), (S3)

**Test 6:** ● ○ ● ○ remaining $d-1$ topological orderings (B4), (T1)/(T4), (T2)/(T5), (T3)/(T6)

**Test 7:** ● different WCCs (B2)

Observe that the tests for Observation (S1), (S2), and (S3) can each be implemented easily using boolean logic, which allows for a concurrent test of all supports whose reachability information is encoded in one accordingly-sized integer: For Observation (S1), it suffices to test whether $r^-(s) \wedge r^+(t) > 0$, and $r^+(s) \wedge \neg r^+(t) > 0$ and $\neg r^-(s) \wedge r^-(t) > 0$ for Observations (S2) and (S3), where $r^+$ and $r^-$ hold the respective forward and backward reachability information in the same order for all supports. Each test hence requires at most one comparison of two integers plus at most two elementary bit operations. Also note that Observation (B1) is implicitly tested by Observations (B5) and (B6). Using the data structure described above, our algorithm requires at most one memory transfer for $s$ and one for $t$ for each QUERY$(s, t)$ that is answerable by one of the observations. Note that there are more observations that allow to identify a negative query than a positive query, which is why we expect a more pronounced speedup for the former. However, as stated in Theorem 1, the reachability in DAGs is always less than 50 %, which justifies a bias towards an optimization for negative queries.

If the query can not be answered using any of these tests, we instead fall back to either another algorithm or a bidirectional BFS with pruning, which uses these tests for each newly encountered vertex $v$ in a subquery QUERY$(v, t)$ (forward step) or QUERY$(s, v)$ (backward step). If a subquery can be answered decisively positive by a test, the bidirectional BFS can immediately answer QUERY$(s, t)$ positively. Otherwise, if a subquery is answered decisively negative by a test, the encountered vertex $v$ is no longer considered (pruning step). If the subquery could not be answered by a test, the vertex $v$ is added to the queue as in a regular (bidirectional) BFS.

## 5 Experimental Evaluation

We evaluated our new algorithm `O'Reach` as a preprocessor to various recent state-of-the-art algorithms listed below against running these algorithms on their own. Furthermore, we use as an additional fallback solution the pruned bidirectional BFS (`pBiBFS`). Our experimental study follows the methodology in [22] and comprises the algorithms `PPL` [37], `TF` [3], `PReaCH` [22], `IP` [36], and `BFL` [28]. Moreover, our evaluation is the first that directly relates `IP` and `BFL` to `PReaCH` and studies the performance of `IP` and `BFL` separately for successful (*positive*) and unsuccessful (*negative*) reachability queries. For reasons of comparison, we also assess the query performance of a full reachability matrix by computing the transitive closure of the input graph entirely during initialization, storing it in a matrix using 1 bit per pair of vertices, and answering each query by a single memory lookup. We refer to this algorithm simply as `Matrix`. As the reachability in DAGs is small and cache locality can influence lookup times, we also experimented with various hash set implementations. However, none was faster or more memory-efficient than `Matrix`.

**Setup and Methodology.** We implemented `O'Reach` in C++14[2] with `pBiBFS` as built-in fallback strategy. For `PPL`[3], `TF`[3], `PReaCH`[4], `IP`[5], and `BFL`[6] we used the original C++ implementation in each case. All source code was compiled with GCC 7.5.0 and full optimization (`-O3`). The experiments were run on a Linux machine under Ubuntu 18.04 with kernel 4.15 on four AMD Opteron 6174 CPUs clocked at 2.2 GHz with 512 kB and 6 MB L2 and L3 cache, respectively and 12 cores per CPU. Overall, the machine has 48 cores and a total of 256 GB of RAM. Unless indicated otherwise, each experiment was run sequentially and exclusively on one processor and its local memory. As non-local memory accesses incur a much higher cost, an exception to this rule was only made for `Matrix`, where we would otherwise have been able to only run twelve instead of 29 instances. We also parallelized the initialization phase for `Matrix`, where the transitive closure is computed, using 48 threads. However, all queries were processed sequentially.

To counteract artifacts of measurement and accuracy, we ran each algorithm five times on each instance and in general use the median for the evaluation. As `O'Reach` uses randomization during initialization, we instead report the average running time over five different seeds. For `IP` and `BFL`, which are randomized in the same way, but don't accept a seed, we just give the average over five repetitions. We note that also taking the median instead or increasing the number of repetitions or seeds does not change the overall picture.

**Instances.** To facilitate comparability, we adopt the instances used in the papers introducing `PReaCH` [22] and `TF` [3], which overlap with those used to evaluate `IP` [36] and `BFL` [28], and which are available either from the `GRAIL` code repository[7] or the Stanford Network Analysis Platform SNAP [21]. Furthermore, we extended the set of benchmark graphs by further instance sizes and Delaunay graphs. Table 2 provides a short overview on the left side, more details are available in the full version [12]. As we only consider DAGs, all instances are condensations of their respective originals, if they were not acyclic already. We also adopt the grouping of the instances as in [39, 22] and provide only a short description of the different sets in the following.

*Kronecker:* These instances were generated by the RMAT generator for the Graph500 benchmark [23] and oriented acyclically from smaller to larger node ID. The name encodes the number of vertices $2^i$ as *kron_logni*. *Random:* Graphs generated according to the Erdős-Renyí model $G(n, m)$ and oriented acyclically from smaller to larger node ID. The name encodes $n = 2^i$ and $m = 2^j$ as *randni-j*. *Delaunay:* Delaunay graphs from the 10th DIMACS Challenge [1, 8]. *delaunay_ni* is a Delaunay triangulation of $2^i$ random points in the unit square. *Large real:* Introduced in [39], these instances represent citation networks (*citeseer.scc*, *citeseerx*, *cit-Patents*), a taxonomy graph (*go-uniprot*), as well as excerpts from the RDF graph of a protein database (*uniprotm22*, *uniprotm100*, *uniprotm150*). *Small real dense:* Among these instances, introduced in [17], are again citation networks (*arXiv*, *pubmed_sub*, *citeseer_sub*), a taxonomy graph (*go_sub*), as well as one obtained from a semantic knowledge database (*yago_sub*). *Small real sparse:* These instances were introduced in [18] and represent XML documents (*xmark*, *nasa*), metabolic networks (*amaze*, *kegg*) or originate from pathway and genome databases (all others). *SNAP:* The e-mail network graph

[2] Source code and instances are available from `https://oreach.taa.univie.ac.at`.
[3] Provided directly by the authors.
[4] `https://github.com/fiji-flo/preach2014/tree/master/original_code`
[5] `https://github.com/datourat/IP-label-for-graph-reachability`
[6] `https://github.com/BoleynSu/bfl`
[7] `https://code.google.com/archive/p/grail/`

**Table 2** Left: Instance sizes (read $/10^3$: in thousands), density, and reachability. Right: Median initialization time in ms over five repetitions. Highlighted results are the overall best.

| Instance | $n/10^3$ | $m/10^3$ | $\frac{m}{n}$ | $\rho\%$ | `O'Reach` | `PReaCH` | `PPL` | `TF` | `IP(s)` | `IP(d)` | `BFL(s)` | `BFL(d)` |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| kron_logn12 | 4.1 | 117.0 | 28.55 | 27.4760 | 451.0 | 13.5 | 56.5 | 46 555.2 | 22.6 | 53.0 | **2.0** | 4.0 |
| kron_logn16 | 65.5 | 2 456.1 | 37.48 | 21.2187 | 13 045.7 | 602.4 | 1 869.5 | | 685.5 | 1 283.0 | **88.8** | 118.3 |
| kron_logn17 | 131.1 | 5 114.0 | 39.02 | 19.4544 | 31 835.0 | 1 425.8 | 4 268.9 | | 1 611.7 | 2 897.9 | **228.1** | 288.1 |
| kron_logn20 | 1 048.6 | 44 619.4 | 42.55 | 5.8195 | 380 698.0 | 20 791.9 | 62 836.0 | | 22 788.2 | 37 103.7 | **3 301.1** | 3 999.0 |
| kron_logn21 | 2 097.2 | 91 040.9 | 43.41 | 1.2150 | 812 416.0 | 46 559.0 | 151 870.0 | | 49 988.1 | 79 226.0 | **7 513.1** | 9 014.9 |
| randn20-21 | 1 048.6 | 2 097.2 | 2.00 | 0.0012 | 4 272.7 | 2 878.3 | 11 579.3 | 11 615.8 | 2 434.7 | 2 635.1 | **626.1** | 677.2 |
| randn20-22 | 1 048.6 | 4 194.3 | 4.00 | 0.0352 | 5 706.9 | 4 459.6 | 43 761.5 | 47 679.2 | 3 364.6 | 3 704.3 | **892.0** | 976.9 |
| randn20-23 | 1 048.6 | 8 388.6 | 8.00 | 1.9067 | 13 724.7 | 7 128.3 | 9 348 510.0 | | 4 830.2 | 5 311.5 | **1 287.7** | 1 449.3 |
| randn23-24 | 8 388.6 | 16 777.2 | 2.00 | 0.0001 | 46 043.5 | 28 959.1 | 132 570.0 | 122 270.0 | 24 566.7 | 25 906.9 | **6 094.8** | 6 580.6 |
| randn23-25 | 8 388.6 | 33 554.4 | 4.00 | 0.0044 | 61 206.2 | 45 573.7 | 413 684.0 | 465 300.0 | 34 145.7 | 36 815.0 | **8 964.7** | 9 715.1 |
| delaunay_n15 | 32.8 | 98.3 | 3.00 | 0.4380 | 104.4 | 38.9 | 174.2 | 602.1 | 42.5 | 55.3 | **7.0** | 9.0 |
| delaunay_n20 | 1 048.6 | 3 145.7 | 3.00 | 0.0093 | 2 816.5 | 1 788.4 | 9 350.5 | 24 563.9 | 2 339.1 | 2 785.1 | **299.8** | 351.5 |
| delaunay_n22 | 4 194.3 | 12 582.9 | 3.00 | 0.0020 | 11 402.7 | 7 363.9 | 38 674.1 | 108 297.0 | 10 106.6 | 11 911.6 | **1 203.1** | 1 394.5 |
| citeseer.scc | 693.9 | 312.3 | 0.45 | 0.0002 | 865.9 | 503.4 | 1 185.3 | 1 579.7 | 602.5 | 613.4 | **107.0** | 122.5 |
| citeseerx | 6 540.4 | 15 011.3 | 2.30 | 0.1367 | 90 695.8 | 12 545.7 | 73 061.0 | 145 773.0 | 11 208.0 | 11 807.4 | **2 349.2** | 2 700.0 |
| cit-Patents | 3 774.8 | 16 518.9 | 4.38 | 0.0409 | 22 358.6 | 15 989.7 | 393 412.0 | 342 680.0 | 13 098.4 | 14 384.0 | **2 905.4** | 3 210.1 |
| go_uniprot | 6 968.0 | 34 769.3 | 4.99 | 0.0004 | 28 270.0 | 11 858.8 | 34 660.6 | 90 942.4 | 11 935.8 | 13 381.6 | **3 137.0** | 3 701.2 |
| uniprotenc_22m | 1 595.4 | 1 595.4 | 1.00 | 0.0001 | 2 802.5 | 714.8 | 2 762.0 | 3 446.0 | 1 322.6 | 1 313.7 | **147.8** | 189.3 |
| uniprotenc_100m | 16 087.3 | 16 087.3 | 1.00 | 0.0000 | 39 539.9 | 10 420.6 | 30 967.4 | 59 660.2 | 16 089.1 | 16 194.7 | **2 169.6** | 2 639.2 |
| uniprotenc_150m | 25 037.6 | 25 037.6 | 1.00 | 0.0000 | 65 983.9 | 17 612.9 | 50 254.7 | 86 052.0 | 26 453.4 | 26 730.9 | **3 830.4** | 4 548.6 |
| go_sub | 6.8 | 13.4 | 1.97 | 0.2258 | 10.4 | 4.0 | 16.6 | 37.6 | 5.0 | 6.2 | **1.0** | **1.0** |
| pubmed_sub | 9.0 | 40.0 | 4.45 | 0.6458 | 19.4 | 9.1 | 31.3 | 101.5 | 8.9 | 10.8 | **2.0** | 3.0 |
| yago_sub | 6.6 | 42.4 | 6.38 | 0.1506 | 12.5 | 6.0 | 18.9 | 61.5 | 7.5 | 10.4 | **1.1** | 2.0 |
| citeseer_sub | 10.7 | 44.3 | 4.13 | 0.3672 | 25.3 | 11.3 | 48.4 | 131.9 | 11.8 | 15.3 | **2.3** | 3.0 |
| arXiv | 6.0 | 66.7 | 11.12 | 15.4643 | 223.2 | 9.7 | 60.8 | 10 008.7 | 14.9 | 26.3 | **2.0** | 3.0 |
| amaze | 3.7 | 3.6 | 0.97 | 17.2337 | 12.0 | 1.2 | 5.3 | 25.9 | 2.2 | 2.4 | **0.0** | 0.4 |
| kegg | 3.6 | 4.4 | 1.22 | 20.1636 | 16.3 | 1.4 | 6.8 | 18.3 | 2.7 | 2.8 | **0.3** | 0.5 |
| nasa | 5.6 | 6.5 | 1.17 | 0.5284 | 7.0 | 2.4 | 11.6 | 27.3 | 3.3 | 3.8 | **1.0** | **1.0** |
| xmark | 6.1 | 7.1 | 1.16 | 1.4513 | 10.7 | 2.3 | 12.9 | 24.2 | 3.9 | 4.3 | **1.0** | **1.0** |
| vchocyc | 9.5 | 10.3 | 1.09 | 0.1517 | 12.0 | 2.9 | 13.4 | 53.7 | 5.4 | 5.9 | **1.0** | **1.0** |
| mtbrv | 9.6 | 10.4 | 1.09 | 0.1511 | 11.1 | 3.0 | 13.7 | 24.0 | 5.4 | 6.0 | **1.0** | **1.0** |
| anthra | 12.5 | 13.1 | 1.05 | 0.0951 | 15.4 | 3.8 | 18.3 | 62.5 | 7.1 | 7.8 | **1.0** | **1.0** |
| ecoo | 12.6 | 13.4 | 1.06 | 0.1088 | 15.9 | 3.9 | 18.8 | 41.4 | 7.4 | 8.0 | **1.0** | **1.0** |
| agrocyc | 12.7 | 13.4 | 1.06 | 0.1060 | 16.1 | 3.9 | 19.1 | 48.1 | 7.4 | 8.1 | **1.0** | **1.0** |
| human | 38.8 | 39.6 | 1.02 | 0.0231 | 49.1 | 13.5 | 56.5 | 104.1 | 23.7 | 25.8 | **3.0** | 4.0 |
| p2p-Gnutella31 | 48.4 | 55.3 | 1.14 | 0.7725 | 120.6 | 28.4 | 89.2 | 52.3 | 43.8 | 44.5 | **5.0** | 7.0 |
| email-EuAll | 230.8 | 223.0 | 0.97 | 5.0732 | 945.2 | 115.3 | 340.5 | 241.3 | 170.1 | 171.4 | **24.8** | 32.0 |
| web-Google | 371.8 | 517.8 | 1.39 | 14.8090 | 5 783.6 | 369.3 | 928.1 | 918.4 | 452.6 | 472.0 | **73.8** | 88.0 |
| soc-LiveJournal1 | 970.3 | 1 024.1 | 1.06 | 5.3781 | 3 663.5 | 739.6 | 2 086.3 | 1 827.9 | 1 160.5 | 1 181.4 | **142.3** | 173.0 |
| wiki-Talk | 2 281.9 | 2 311.6 | 1.01 | 0.8117 | 6 347.0 | 1 492.1 | 4 317.8 | 2 715.4 | 2 597.7 | 2 620.7 | **269.9** | 343.5 |

(*email-EuAll*), peer-to-peer network (*p2p-Gnutella31*), social network (*soc-LiveJournal1*), web graph (*web-Google*), as well as the communication network (*wiki-Talk*) are part of SNAP and were first used in [3].

**Queries.** Following the methodology of [22], we generated three sets of 100 000 queries each: *positive*, *negative*, and *random*. Each set consists of random queries, which were generated by picking two vertices uniformly at random and filtering out negative or positive queries for the *positive* and *negative* query sets, respectively. The fourth query set, *mixed*, is a randomly shuffled union of all queries from *positive* and *negative* and hence contains 200 000 pairs of vertices. As the order of the queries within each set had an observable effect on the running time due to caching effects and memory layout, we randomly shuffled every query set five times and used a different permutation for each repetition of an experiment to ensure equal conditions for all algorithms.

## 5.1 Experimental Results

We ran `O'Reach` with $k = 16$ supportive vertices, picked from 1 200 candidates ($p = 75$, $h = 8$) and $d = 4$ extended topological orderings. We ran `IP` with the two configurations used also by the authors [36] and refer to the resulting algorithms as `IP(s)` (*sparse*, $h_{\texttt{IP}} = k_{\texttt{IP}} = 2$)

**Table 3** Average query time per algorithm and query set.

|  | O'R+ |  | O'R+ |  | O'R+ |  | O'R+ |  | O'R+ |  | O'R+ |  | O'R+ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Query set | pBiBFS | PReaCH | PReaCH | PPL | PPL | IP(s) | IP(s) | IP(d) | IP(d) | BFL(s) | BFL(s) | BFL(d) | BFL(d) |
| *random* | 3.523 | 1.596 | 1.483 | 0.271 | **0.149** | 12.865 | 11.193 | 9.778 | 8.516 | 6.645 | 5.073 | 5.063 | 3.361 |
| *mixed* | 19.964 | 6.351 | 6.102 | 0.352 | **0.258** | 80.572 | 73.625 | 60.352 | 56.433 | 32.456 | 28.496 | 22.002 | 17.541 |
| *positive* | 37.554 | 11.508 | 11.069 | 0.399 | **0.345** | 156.016 | 145.532 | 118.835 | 109.014 | 62.338 | 54.329 | 42.632 | 33.699 |
| *negative* | 2.382 | 1.188 | 1.154 | 0.260 | **0.149** | 5.342 | 5.059 | 3.727 | 3.793 | 2.496 | 2.506 | 1.345 | 1.358 |

and IP(d) (*dense*, $h_{\mathtt{IP}} = k_{\mathtt{IP}} = 5$). Similarly, we evaluated BFL [28] with configuration *sparse* as BFL(s) ($s_{\mathtt{BFL}} = 64$) and *dense* as BFL(d) ($s_{\mathtt{BFL}} = 160$), following the presets given by the authors.

**Average query times.** Table A.6 lists the average time per query for the query sets *negative* and *positive*. All missing values are due to a memory requirement of more than 32 GB (TF) and Matrix (256 GB). For each instance and query set, the running time of the fastest algorithm is printed in bold. If Matrix was fastest, also the running time of the second-best algorithm is highlighted. Besides Matrix, the table shows the running times of PReaCH, PPL, IP(d), and BFL(d) alone as well as multiple versions for O'Reach: one with a pruned bidirectional BFS (O'R+pBiBFS) as fallback as well as one per competitor (O'R+...), where O'Reach was run without fallback and the queries left unanswered were fed to the competitor. Analogously, the running times for IP(s), BFL(s), and TF alone and as fallback for O'Reach are given in Table A.9.

Our results by and large *confirm* the performance comparison of PReaCH, PPL, and TF conducted by Merz and Sanders [22]. PReaCH was the fastest on three out of five Kronecker graphs for the negative query set, once beaten by O'R+PReaCH and O'R+PPL each, whereas PPL and O'R+PPL dominated all others on the positive query set in this class as well as on three of the five random graphs, while O'R+TF was slightly faster on the other two. PReaCH was also the dominating approach on the small real sparse and SNAP instances in the aforementioned study [22]. By contrast, it was *outperformed* on these classes here by O'Reach with almost any fallback on all instances for the positive query set, and by either IP(d) or BFL(s) on almost all instances for the negative query set. On the Delaunay and large real instances, BFL(s) often was the fastest algorithm on the set of negative queries. The results also reveal that BFL and in particular IP have a weak spot in answering positive queries. *On average over all instances*, O'R+PPL had the *fastest average query time* both for *negative* and *positive* queries.

Notably, Matrix was *outperformed* quite often, especially for queries in the set *negative*, which correlates with the fact that a large portion of these queries could be answered by constant-time observations (see also the detailed analysis of observation effectiveness below) and is due to its larger memory footprint. Across all instances and seeds, more than 95 % of all queries in this set could be answered by O'Reach directly. On the set *positive*, the average query time for Matrix was in almost all cases less than on the *negative* query set, which is explained by the small reachability of the instances and a resulting higher spatial locality and better cacheability of the few and naturally clustered one-entries in the matrix. Consequently, this effect was distinctly reduced for the *mixed* query set, as shown in Table A.7.

There are some instances where O'Reach had a fallback rate of over 90 % for the *positive* query set, e.g., on *cit-Patents*, which is clearly reflected in the running time. Except for PPL, all algorithms had difficulties with positive queries on this instance. Conversely, the fallback rate on all *uniprotenc_** instances and *citeseer.scc*, e.g., was 0 %. On average across all instances and seeds, O'Reach could answer over 70 % of all *positive* queries by constant-time observations.

**Table 4** Mean speedups with O’Reach plus fallback over pure fallback algorithm. Values greater 1.00 are highlighted.

| | negative | | | | positive | | | | random | | | | mixed | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | PReaCH | PPL | IP(d) | BFL(d) | PReaCH | PPL | IP(d) | BFL(d) | PReaCH | PPL | IP(d) | BFL(d) | PReaCH | PPL | IP(d) | BFL(d) |
| Geometric Mean | **1.10** | **2.22** | 0.92 | **1.06** | **1.33** | **1.90** | **3.98** | **3.14** | **1.29** | **2.53** | **1.26** | **2.40** | **1.29** | **2.04** | **2.77** | **2.31** |
| Ratio Runtime Avgs | **1.03** | **1.75** | 0.98 | 0.99 | **1.04** | **1.16** | **1.09** | **1.27** | **1.08** | **1.82** | **1.15** | **1.51** | **1.04** | **1.36** | **1.07** | **1.25** |
| Average | **1.13** | **2.32** | 0.98 | **1.35** | **1.41** | **2.25** | **5.87** | **6.25** | **1.33** | **2.69** | **1.41** | **8.22** | **1.33** | **2.23** | **3.37** | **3.63** |

The results on the query sets *random* and *mixed* are similar and listed in Table A.7 and Table A.10. Once again, O’R+PPL showed the *fastest query time on average across all instances* for both query sets. As the reachability in a DAG is low in general (see also Theorem 1) and particularly in the benchmark instances, the average query times for *random* resemble those for *negative*. On the other hand, the results for the *mixed* query set are more similar to those for the *positive* query set, as the relative differences in performance among the algorithms are more pronounced there. Table 3 compactly shows the average query time over all instances for each query set. Only PPL and O’R+PPL achieved an average query time of less than 1 µs (and even less than 0.35 µs).

**Speedups by O’Reach.** We next investigate the relative speedup of O’Reach with different fallback solutions over running only the fallback algorithms. Table A.8 lists the ratios of the average query time of each competitor algorithm run standalone divided by the average query time of O’Reach plus that algorithm as fallback, for all four query sets. A compact version is also given in Table 4. In the large majority of cases, using O’Reach as a preprocessor resulted in a speedup, except in case of *negative* or *random* queries for BFL and partially IP on the large real instances as well as for PReaCH and partially again IP on the small real sparse and SNAP instances. The largest speedup of around 105 could be achieved for BFL on *kegg* for random queries. The mean speedup (geometric) is at least 1.29 for all fallback algorithms on the query sets *positive*, *random*, and *mixed*, where the maximum was reached for IP(s) on *positive* queries with a factor of 4.21. Only for purely *negative* queries, IP(d) and BFL(s) were a bit faster alone in the mean values. *In summary*, given that the algorithms are often already faster than single memory lookups, the speedups achieved by O’Reach are quite high.

**Initialization Times (Table 2, right).** On all graphs, BFL(s) had the fastest initialization time, followed by BFL(d) and PReaCH. For O’Reach, the overhead of computing the comparatively large out- and in-reachabilities of all 1 200 candidates for $k = 16$ supportive vertices is clearly reflected in the running time on denser instances and can be reduced greatly if lower parameters are chosen, albeit at the expense of a slightly reduced query performance, e.g., for $k = 8$. PPL often consumed a lot of time in this step, especially on denser instances, with a maximum of 2.6 h on *randn20-23*.

Based on the average query time per instance, the *minimum number of random queries necessary to amortize* the additional investment in initialization time if O’Reach is run as preprocessor is between 9.6 thousand (O’R+BFL(d)) and 499 thousand (O’R+PReaCH). Counting cases where O’Reach could not achieve a speedup in the average query time as infinity, the *median number of random queries* required for amortization is between 2.5 million (O’R+BFL(d)) and 101 million (O’R+IP(d)). For the on average fastest algorithm, O’R+PPL, the initialization cost is recovered after 210 thousand (*nasa*) to 6.15 billion (*kron_logn21*) *random* queries, which equals about 0.77 % (*nasa*) and 0.14 % (*kron_logn21*) of all vertex pairs, respectively.

**Effectiveness of Observations.** We collected a vast amount of statistical data to perform an analysis of the effectiveness of the different observations used in `O'Reach`. To make the analysis more robust, we increased the number of seeds to 25 here.

First, we look only at *fast queries*, i.e., those queries that could be answered without a fallback. Across all query sets, the *most effective* observation was the negative basic observation on topological orderings, (B4), which answered around 30 % of all fast queries. As the average reachability in the *random* query set is very low, negative queries predominate in the overall picture. It thus does not come as a surprise that the most effective observation is a negative one. On the *negative* query set, (B4) could answer 45 % of all fast queries. After lowering the number of topological orderings to $d = 2$, (B4) was still the most effective and could answer 23 % of all fast queries and 33 % of those in the *negative* query set. The negative observations second to (B4) in effectiveness were those looking at the forward and backward topological levels, Observation (B5) and (B6), which could answer around 15 % each on the *negative* query set and around 10 % of all fast queries. Note that we increased the counter for *all* observations that could answer a query for this analysis, not just the first in order, which is why there may be overlaps. The observations using the max and min indices of extended topological orderings, (T2) and (T5), could answer 9 % and 6 % of the fast queries in the *negative* query set, and the observations based on supportive vertices, (S2) and (S3), around 3 % each. Reducing the number of topological orderings to $d = 2$ decreased the effectiveness of (T2) and (T5) to around 5 %.

The *most effective positive observation* and the second-best among all query sets, was the supportive-vertices-based Observation (S1), which could answer almost 16 % of all fast queries and almost 55 % in the *positive* query set. Follow-up observations were the ones using high and low indices, (T1) and (T4), with 18 % and 16 % effectiveness for the *positive* query set. The remaining two, (T2) and (T5), could answer 6 % and 4 % in this set. Reducing the number of topological orderings to $d = 2$ led to a slight deterioration in case of (T1) and (T4) to 14 %, and to 5 % and 3 % in case of (T2) and (T5), each with respect to the *positive* query set.

Among all fast queries that could be answered by *only one* observation, the most effective observation was the positive supportive-vertices-based Observation (S1) with over 40 % for all query sets and 68 % for the *positive* query set, followed by the negative basic observation using topological orderings, (B4), with a bit over 20 % for all query sets and 52 % for the *negative* query set.

Looking now at the entire query sets, our statistics show that 95 % of all *queries could be answered via an observation* on the *negative* set. In 70 % of all cases, (B5) in the second test, which uses topological forward levels, could already answer the query. In further 16 % of all cases, the observation based on topological backward levels, (B6), was successful. On the *positive* query set, the fallback rate was 28 % and hence higher than on the *negative* query set. 52 % of all queries in this set could be answered by the supportive-vertices-based observation (S1), and the high and low indices of extended topological orderings (T1) and (T4) were responsible for another 7 % each. Observe that here, the first observation in the order that can answer a query "wins the point", i.e., there are no overlaps in the reported effectiveness.

**Memory Consumption.** Table 5 lists the memory each algorithm used for their *reachability index*. As `O'Reach` was configured with $k = 16$ and $d = 4$, its index size is 64n Byte. Consequently, the reachability indices of `O'Reach`, `PReaCH`, `PPL`, `IP`, `BFL`, and, with one exception for `TF`, fit in the L3 cache of 6 MB for all small real instances. For `Matrix`, this

**Table 5** Real index size in memory (in MB).

| Instance | `O'Reach` | `PReaCH` | `PPL` | `TF` | `IP(s)` | `IP(d)` | `BFL(s)` | `BFL(d)` | `Matrix` |
|---|---|---|---|---|---|---|---|---|---|
| kron_logn12 | 0.3 | 0.2 | **0.1** | 19.2 | **0.1** | 0.2 | **0.1** | 0.2 | 2.0 |
| kron_logn16 | 4.0 | 3.5 | 1.5 | 0.0 | 1.5 | 3.1 | **1.3** | 2.3 | 512.0 |
| kron_logn17 | 8.0 | 7.0 | 3.0 | 0.0 | 2.9 | 6.1 | **2.5** | 4.6 | 2 047.9 |
| kron_logn20 | 64.0 | 56.0 | 25.1 | 0.0 | 22.1 | 44.8 | **18.9** | 34.1 | 131 070 |
| kron_logn21 | 128.0 | 112.0 | 50.4 | 0.0 | 43.5 | 87.3 | **37.1** | 66.4 | 0.0 |
| randn20-21 | 64.0 | 56.0 | 24.2 | 64.8 | **18.0** | 31.5 | 20.6 | 38.7 | 131 070 |
| randn20-22 | 64.0 | 56.0 | 136.8 | 482.3 | **19.0** | 37.6 | 22.3 | 43.3 | 131 070 |
| randn20-23 | 64.0 | 56.0 | 4 380.3 | 0.0 | **19.5** | 40.8 | 23.1 | 45.6 | 131 070 |
| randn23-24 | 512.0 | 448.0 | 193.7 | 518.2 | **144.3** | 252.3 | 164.5 | 309.4 | 0.0 |
| randn23-25 | 512.0 | 448.0 | 1 073.3 | 3 844.1 | **152.0** | 300.7 | 178.0 | 346.0 | 0.0 |
| delaunay_n15 | 2.0 | 1.7 | 0.8 | 4.7 | **0.6** | 1.2 | 0.7 | 1.4 | 128.0 |
| delaunay_n20 | 64.0 | 56.0 | 33.0 | 126.7 | **19.1** | 38.1 | 22.5 | 43.9 | 131 070 |
| delaunay_n22 | 256.0 | 224.0 | 135.0 | 497.9 | **76.6** | 152.5 | 90.0 | 175.8 | 0.0 |
| citeseer.scc | 42.4 | 37.1 | 7.1 | 28.3 | 9.4 | 11.3 | **9.2** | 13.7 | 57 406.5 |
| citeseerx | 399.2 | 349.3 | 120.9 | 1 773.0 | 111.8 | 151.0 | **107.6** | 185.1 | 0.0 |
| cit-Patents | 230.4 | 201.6 | 659.2 | 780.0 | 72.9 | 138.0 | **71.7** | 132.9 | 0.0 |
| go_uniprot | 425.3 | 372.1 | 261.0 | 680.2 | **106.4** | 184.7 | 113.1 | 193.1 | 0.0 |
| uniprotenc_22m | 97.4 | 85.2 | 18.5 | 67.2 | **24.5** | 24.7 | 26.1 | 44.8 | 0.0 |
| uniprotenc_100m | 981.9 | 859.2 | 197.2 | 690.4 | **251.2** | 269.1 | 270.8 | 471.9 | 0.0 |
| uniprotenc_150m | 1 528.2 | 1 337.1 | 318.5 | 1 087.0 | **395.0** | 439.6 | 428.5 | 753.8 | 0.0 |
| go_sub | 0.4 | 0.4 | 0.2 | 0.4 | **0.1** | 0.2 | **0.1** | 0.3 | 5.5 |
| pubmed_sub | 0.5 | 0.5 | 0.3 | 1.1 | **0.1** | 0.2 | 0.2 | 0.3 | 9.7 |
| yago_sub | 0.4 | 0.4 | 0.2 | 0.5 | **0.1** | 0.2 | **0.1** | 0.2 | 5.3 |
| citeseer_sub | 0.7 | 0.6 | 0.3 | 1.2 | **0.2** | 0.3 | **0.2** | 0.4 | 13.7 |
| arXiv | 0.4 | 0.7 | 0.3 | 14.9 | **0.1** | 0.3 | **0.1** | 0.2 | 4.3 |
| amaze | 0.2 | 0.2 | 0.0 | 0.2 | **0.1** | **0.1** | **0.1** | **0.1** | 1.6 |
| kegg | 0.2 | 0.2 | **0.1** | 0.2 | **0.1** | **0.1** | **0.1** | **0.1** | 1.6 |
| nasa | 0.3 | 0.3 | **0.1** | 0.3 | **0.1** | 0.2 | **0.1** | 0.2 | 3.7 |
| xmark | 0.4 | 0.3 | 0.2 | 0.4 | **0.1** | 0.2 | **0.1** | 0.2 | 4.4 |
| vchocyc | 0.6 | 0.5 | **0.2** | 0.7 | **0.2** | 0.3 | **0.2** | 0.3 | 10.7 |
| mtbrv | 0.6 | 0.5 | **0.2** | 0.4 | **0.2** | 0.3 | **0.2** | 0.3 | 11.0 |
| anthra | 0.8 | 0.7 | **0.2** | 0.8 | **0.2** | 0.4 | **0.2** | 0.4 | 18.6 |
| ecoo | 0.8 | 0.7 | **0.2** | 0.9 | **0.2** | 0.4 | **0.2** | 0.4 | 19.0 |
| agrocyc | 0.8 | 0.7 | **0.2** | 0.9 | **0.2** | 0.4 | **0.2** | 0.4 | 19.2 |
| human | 2.4 | 2.1 | **0.6** | 2.1 | 0.7 | 1.2 | **0.6** | 1.1 | 179.6 |
| p2p-Gnutella31 | 3.0 | 2.6 | 0.7 | 2.1 | 0.9 | 1.5 | **0.8** | 1.4 | 279.7 |
| email-EuAll | 14.1 | 12.3 | 2.6 | 9.7 | **3.7** | 5.8 | **3.7** | 6.4 | 6 349.8 |
| web-Google | 22.7 | 19.9 | 5.4 | 16.7 | 7.0 | 11.2 | **6.5** | 11.5 | 16 475.5 |
| soc-LiveJournal1 | 59.2 | 51.8 | 13.0 | 41.0 | 19.1 | 31.8 | **15.9** | 27.2 | 112 225 |
| wiki-Talk | 139.3 | 121.9 | 26.2 | 95.9 | 52.0 | 103.5 | **37.1** | 63.3 | 0.0 |

was only the case for the four smallest instances from the small real sparse set, three of the small real dense ones, and the smallest Kronecker graph, which is clearly reflected in its average query time for the *negative*, *random*, and, to a slightly lesser extent, *mixed* query sets. Whereas for `O'Reach`, `PReaCH`, and `Matrix`, the index size depends solely on the number of vertices, `IP`, `BFL`, `PPL` and `TF` consumed more memory the larger the density $\frac{m}{n}$. `IP(s)` usually was the most space-efficient and never used more than 395 MB, followed by `BFL(s)` (429 MB), `IP(d)` (440 MB), `BFL(d)` (754 MB), `PReaCH` (1.3 GB), `O'Reach` (1.5 GB), and `PPL` (4.4 GB). All these algorithms are hence suitable to handle graphs with several millions of vertices even on hardware with relatively little memory (with respect to current standards). `TF` used up to 3.8 GB (*randn23-25*), but required even more than 64 GB at least during initialization on all instances where the data is missing in the table.

## 6 Conclusion

In this paper, we revisited existing techniques for the static reachability problem and combined them with new approaches to support a large portion of *reachability queries* in constant time using a linear-sized *reachability index*. Our extensive experimental evaluation shows that in almost all scenarios, combining any of the existing algorithms with our new techniques implemented in `O'Reach` can speed up the query time by several factors. In particular *supportive vertices* have proven to be effective to answer positive queries quickly. As a further plus, `O'Reach` is flexible: memory usage, initialization time, and expected query time can be influenced directly by three parameters, which allow to trade space for time or initialization time for query time. Moreover, our study demonstrates that, due to cache effects, a high investment in space does not necessarily pay off: *Reachability queries* can often be answered even significantly faster than single memory accesses in a precomputed full reachability matrix.

The on average fastest algorithm across all instances and types of queries was a combination of `O'Reach` and `PPL` with an average query time of less than 0.35 µs. As the initialization time of `PPL` is relatively high, we also recommend `O'Reach` combined with `PReaCH` as a less expensive alternative solution with respect to initialization time and partially also memory, which still achieved an average query time of at most 11.1 µs on all query sets.

## References

1 D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

2 Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 893–902. IEEE Computer Society, 2008. `doi:10.1109/ICDE.2008.4497498`.

3 James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 193–204. ACM, 2013. `doi:10.1145/2463676.2465286`.

4 Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*, volume 3896 of *Lecture Notes in Computer Science*, pages 961–979. Springer, 2006. `doi:10.1007/11687238_56`.

5 Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003. `doi:10.1137/S0097539702403098`.

6 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to algorithms, 2009.

7 R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962. `doi:10.1145/367766.368168`.

8 Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018*, 2018.

9 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

10 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

11 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Faster Fully Dynamic Transitive Closure in Practice. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SEA.2020.14.

12 Kathrin Hanauer, Christian Schulz, and Jonathan Trummer. O'Reach: Even faster reachability in static graphs. *CoRR*, abs/2008.10932, 2021. arXiv:2008.10932.

13 H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990. doi:10.1145/99935.99944.

14 Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: scaling reachability computation on large graphs. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 169–180. ACM, 2012. doi:10.1145/2213836.2213856.

15 Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7:1–7:44, 2011. doi:10.1145/1929934.1929941.

16 Ruoming Jin and Guan Wang. Simple, fast, and scalable reachability oracle. *Proc. VLDB Endow.*, 6(14):1978–1989, 2013. doi:10.14778/2556549.2556578.

17 Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 813–826, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1559845.1559930.

18 Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 595–608. ACM, 2008. doi:10.1145/1376616.1376677.

19 A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962. doi:10.1145/368996.369025.

20 F. Le Gall. Powers of tensors and fast matrix multiplication. In K. Nabeshima, K. Nagasaka, F. Winkler, and Á. Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014. doi:10.1145/2608628.2608664.

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

22 F. Merz and P. Sanders. Preach: A fast lightweight reachability index using pruning and contraction hierarchies. In A. S. Schulz and D. Wagner, editors, *European Symposium on Algorithms*, pages 701–712, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

23 Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

24 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

25 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

26 Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: an efficient connection index for complex XML document collections. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 237–255. Springer, 2004. doi:10.1007/978-3-540-24741-8_15.

27 B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34, 2008.

28 Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.*, 29(3):683–697, 2017. doi:10.1109/TKDE.2016.2631160.

29 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

30 Robert Endre Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

31 Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 845–856. ACM, 2007. doi:10.1145/1247480.1247573.

32 Sebastiaan J. van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 913–924. ACM, 2011. doi:10.1145/1989323.1989419.

33 Renê Rodrigues Veloso, Loïc Cerf, Wagner Meira, and Mohammed J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *EDBT*, pages 511–522, 2014.

34 Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 75. IEEE Computer Society, 2006. doi:10.1109/ICDE.2006.53.

35 S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962. doi:10.1145/321105.321107.

36 Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: an independent permutation labeling approach. *VLDB J.*, 27(1):1–26, 2018. doi:10.1007/s00778-017-0468-3.

37 Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1601–1606. ACM, 2013. doi:10.1145/2505515.2505724.

38 Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. Grail: Scalable reachability index for large graphs. *Proc. VLDB Endow.*, 3(1–2):276–284, 2010. doi:10.14778/1920841.1920879.

39 Hilmi Yıldırım, Vineet Chaoji, and Mohammed J Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, 2012.

40 Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 181–215. Springer, 2010. doi:10.1007/978-1-4419-6045-0_6.

## A Appendix

**Table A.6** Average query times in µs for 100 000 negative (left) and positive queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

| | ← negative | | | | | | | | | | positive→ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | `O'R+ pBiBFS` | `PReaCH` | `O'R+ PReaCH` | `PPL` | `O'R+ PPL` | `IP(d)` | `O'R+ IP(d)` | `BFL(d)` | `O'R+ BFL(d)` | `Matrix` | `O'R+ pBiBFS` | `PReaCH` | `O'R+ PReaCH` | `PPL` | `O'R+ PPL` | `IP(d)` | `O'R+ IP(d)` | `BFL(d)` | `O'R+ BFL(d)` | `Matrix` |
| kron_logn12 | 0.031 | 0.020 | **0.017** | 0.030 | 0.018 | 0.028 | 0.028 | 0.097 | 0.046 | **0.017** | 0.347 | 0.361 | 0.251 | 0.035 | **0.032** | 2.213 | 0.824 | 3.278 | 0.984 | **0.014** |
| kron_logn16 | 0.094 | **0.057** | 0.069 | 0.109 | 0.075 | 0.078 | 0.113 | 0.161 | 0.153 | 0.533 | 3.246 | 3.467 | 2.637 | 0.115 | **0.106** | 25.690 | 10.530 | 25.007 | 9.725 | 0.262 |
| kron_logn17 | 0.122 | **0.072** | 0.078 | 0.135 | 0.086 | 0.095 | 0.131 | 0.117 | 0.117 | 1.111 | 2.365 | 2.537 | 0.692 | 0.152 | **0.100** | 20.548 | 4.597 | 9.929 | 1.734 | 0.881 |
| kron_logn20 | 0.184 | **0.117** | 0.128 | 0.221 | 0.119 | 0.167 | 0.201 | 0.325 | 0.353 | 2.413 | 46.186 | 25.092 | 23.331 | 0.274 | **0.265** | 342.887 | 163.902 | 373.848 | 162.405 | 1.778 |
| kron_logn21 | 0.224 | 0.146 | 0.162 | 0.250 | **0.139** | 0.207 | 0.257 | 0.237 | 0.286 | | 67.184 | 15.416 | 5.998 | **0.313** | 0.355 | 306.828 | 193.212 | 203.695 | 113.502 | |
| randn20-21 | 0.255 | 0.342 | 0.234 | 0.278 | 0.154 | 0.218 | 0.191 | 0.056 | 0.122 | 1.692 | 1.298 | 1.044 | 0.858 | 0.482 | 0.404 | 2.513 | 1.792 | 0.856 | 0.693 | 0.844 |
| randn20-22 | 3.016 | 2.473 | 2.298 | 0.424 | 0.335 | 2.569 | 2.519 | 0.357 | 0.429 | 1.056 | 35.747 | 21.873 | 21.832 | **1.153** | 1.278 | 53.591 | 53.431 | 9.369 | 9.345 | **0.361** |
| randn20-23 | 82.960 | 36.193 | 36.139 | 3.092 | **2.449** | 139.544 | 142.731 | 51.125 | 51.046 | 4.993 | 870.787 | 247.700 | 246.218 | **4.625** | 4.775 | 3 524.734 | 3 462.790 | 971.807 | 944.526 | **1.822** |
| randn23-24 | 0.356 | 0.494 | 0.426 | 0.402 | 0.216 | 0.310 | 0.264 | 0.055 | 0.168 | | 1.747 | 1.366 | 1.169 | 0.680 | 0.557 | 3.272 | 2.345 | 1.074 | 0.892 | |
| randn23-25 | 4.130 | 3.244 | 3.011 | 0.546 | 0.426 | 3.237 | 3.143 | 0.409 | 0.529 | | 51.713 | 29.098 | 29.108 | **1.212** | 1.379 | 70.327 | 70.425 | 11.781 | 11.825 | |
| delaunay_n15 | 0.125 | 0.172 | 0.076 | 0.118 | 0.055 | 0.189 | 0.088 | 0.053 | 0.045 | 0.063 | 1.227 | 0.150 | 0.156 | **0.102** | 0.108 | 5.207 | 2.693 | 0.762 | 0.569 | **0.050** |
| delaunay_n20 | 0.237 | 0.350 | 0.187 | 0.288 | 0.142 | 0.355 | 0.183 | 0.055 | 0.124 | 0.924 | 2.984 | **0.334** | 0.397 | 0.417 | 0.419 | 9.180 | 6.407 | 2.173 | 1.817 | 0.505 |
| delaunay_n22 | 0.280 | 0.418 | 0.274 | 0.363 | 0.177 | 0.415 | 0.221 | 0.054 | 0.154 | | 3.354 | **0.425** | 0.590 | 0.560 | 0.561 | 9.249 | 7.219 | 2.945 | 2.532 | |
| citeseer.scc | 0.056 | 0.075 | 0.056 | 0.229 | 0.056 | 0.050 | 0.056 | 0.043 | 0.056 | 1.312 | 0.113 | **0.089** | 0.112 | 0.303 | 0.112 | 0.384 | 0.112 | 0.179 | 0.112 | 0.653 |
| citeseerx | 0.211 | 0.212 | 0.214 | 0.358 | 0.141 | 0.160 | 0.172 | **0.058** | 0.141 | | 0.544 | 0.248 | 0.222 | 0.296 | **0.085** | 2.511 | 0.426 | 1.686 | 0.291 | |
| cit-Patents | 3.965 | 2.732 | 2.562 | 0.526 | **0.329** | 3.915 | 3.797 | 0.658 | 0.737 | | 238.913 | 118.427 | 117.240 | **1.961** | 2.123 | 473.083 | 482.668 | 118.097 | 116.791 | |
| go_uniprot | 0.098 | 0.121 | 0.099 | 0.385 | 0.098 | 0.068 | 0.098 | 0.042 | 0.098 | | 208.494 | 1.026 | 0.902 | 0.534 | **0.435** | 1.054 | 0.712 | 0.688 | 0.519 | |
| uniprotenc_22m | 0.067 | 0.068 | 0.066 | 0.254 | 0.066 | 0.045 | 0.066 | 0.043 | 0.066 | | **0.072** | 0.076 | **0.072** | 0.274 | **0.072** | 0.334 | **0.072** | 0.196 | **0.072** | |
| uniprotenc_100m | 0.130 | 0.163 | 0.131 | 0.410 | 0.131 | 0.098 | 0.131 | 0.043 | 0.131 | | 0.118 | **0.108** | 0.118 | 0.452 | 0.118 | 0.504 | 0.118 | 0.233 | 0.118 | |
| uniprotenc_150m | 0.152 | 0.206 | 0.153 | 0.444 | 0.153 | 0.116 | 0.153 | 0.044 | 0.153 | | 0.139 | **0.121** | 0.139 | 0.509 | 0.139 | 0.565 | 0.139 | 0.239 | 0.139 | |
| go_sub | 0.033 | 0.046 | 0.028 | 0.058 | 0.026 | 0.050 | 0.031 | 0.057 | **0.024** | 0.025 | 0.198 | 0.139 | 0.088 | 0.092 | **0.055** | 2.447 | 0.448 | 0.355 | 0.154 | **0.012** |
| pubmed_sub | 0.078 | 0.076 | 0.066 | 0.068 | 0.044 | 0.058 | 0.057 | 0.061 | **0.039** | **0.029** | 0.546 | 0.491 | 0.340 | 0.090 | **0.073** | 1.441 | 0.577 | 0.922 | 0.399 | **0.019** |
| yago_sub | 0.025 | 0.030 | 0.023 | 0.058 | 0.023 | 0.022 | 0.023 | 0.048 | 0.022 | 0.026 | 0.146 | 0.097 | 0.074 | 0.086 | **0.057** | 0.342 | 0.137 | 0.225 | 0.102 | **0.020** |
| citeseer_sub | 0.083 | 0.089 | 0.059 | 0.071 | 0.038 | 0.072 | 0.054 | 0.055 | **0.029** | 0.032 | 0.580 | 0.285 | 0.223 | 0.095 | **0.087** | 1.187 | 0.642 | 0.574 | 0.318 | **0.026** |
| arXiv | 0.247 | 0.258 | 0.223 | 0.076 | **0.047** | 0.299 | 0.242 | 0.130 | 0.091 | **0.024** | 1.209 | 1.216 | 0.637 | **0.046** | 0.047 | 6.445 | 2.790 | 3.464 | 1.657 | **0.018** |
| amaze | 0.012 | 0.014 | 0.013 | 0.030 | 0.013 | **0.011** | 0.013 | 0.048 | 0.013 | 0.016 | 0.010 | 0.015 | **0.009** | 0.031 | **0.009** | 0.089 | **0.009** | 0.102 | **0.009** | **0.009** |
| kegg | 0.014 | 0.015 | 0.015 | 0.033 | 0.015 | 0.014 | 0.015 | 0.053 | 0.015 | 0.032 | 0.010 | 0.016 | **0.009** | 0.031 | **0.009** | 0.094 | **0.009** | 0.102 | **0.009** | 0.010 |
| nasa | **0.026** | 0.031 | 0.029 | 0.048 | 0.027 | 0.032 | 0.031 | 0.054 | **0.026** | **0.022** | 0.061 | 0.058 | 0.044 | 0.044 | **0.022** | 1.627 | 0.148 | 0.351 | 0.044 | **0.008** |
| xmark | 0.031 | 0.032 | 0.027 | 0.052 | 0.026 | 0.042 | 0.031 | 0.055 | **0.023** | 0.024 | 0.036 | 0.049 | 0.026 | 0.032 | **0.014** | 0.432 | 0.045 | 2.163 | 0.021 | **0.008** |
| vchocyc | 0.016 | 0.016 | 0.017 | 0.050 | 0.017 | **0.013** | 0.017 | 0.047 | 0.017 | 0.029 | 0.015 | 0.024 | **0.014** | 0.039 | **0.014** | 0.241 | 0.015 | 0.096 | 0.015 | **0.007** |
| mtbrv | 0.017 | 0.016 | 0.018 | 0.050 | 0.018 | **0.013** | 0.018 | 0.047 | 0.018 | 0.029 | 0.017 | 0.025 | 0.017 | 0.039 | **0.016** | 0.233 | 0.019 | 0.105 | 0.017 | **0.006** |
| anthra | 0.017 | 0.018 | 0.019 | 0.054 | 0.019 | **0.013** | 0.019 | 0.047 | 0.020 | 0.033 | **0.014** | 0.025 | **0.014** | 0.043 | **0.014** | 0.283 | 0.015 | 0.087 | **0.014** | **0.005** |
| ecoo | 0.017 | 0.017 | 0.019 | 0.053 | 0.019 | **0.013** | 0.019 | 0.047 | 0.019 | 0.055 | 0.015 | 0.027 | **0.014** | 0.040 | **0.014** | 0.266 | 0.015 | 0.111 | **0.014** | **0.006** |
| agrocyc | 0.018 | 0.017 | 0.021 | 0.054 | 0.021 | **0.013** | 0.021 | 0.046 | 0.021 | 0.033 | **0.014** | 0.027 | **0.014** | 0.042 | **0.014** | 0.249 | 0.015 | 0.139 | 0.015 | **0.006** |
| human | 0.025 | 0.025 | 0.033 | 0.097 | 0.033 | **0.015** | 0.033 | 0.045 | 0.033 | 0.072 | **0.022** | 0.036 | **0.022** | 0.083 | **0.022** | 0.281 | **0.022** | 0.118 | **0.022** | **0.006** |
| p2p-Gnutella31 | 0.031 | 0.030 | 0.037 | 0.111 | 0.037 | **0.017** | 0.037 | 0.046 | 0.036 | 0.100 | **0.026** | 0.037 | **0.026** | 0.070 | **0.026** | 0.191 | **0.026** | 0.274 | **0.026** | **0.023** |
| email-EuAll | 0.054 | 0.062 | 0.061 | 0.161 | 0.062 | 0.055 | 0.062 | 0.045 | 0.061 | 5.267 | **0.042** | 0.058 | **0.042** | 0.204 | **0.042** | 0.342 | **0.042** | 0.197 | **0.042** | 2.858 |
| web-Google | 0.077 | 0.079 | 0.076 | 0.175 | 0.075 | 0.081 | 0.076 | 0.052 | 0.070 | 1.400 | 0.049 | 0.068 | **0.048** | 0.190 | **0.048** | 0.458 | **0.048** | 0.237 | **0.048** | 1.405 |
| soc-LiveJournal1 | 0.065 | 0.062 | 0.070 | 0.192 | 0.072 | 0.057 | 0.072 | 0.046 | 0.069 | 3.785 | **0.058** | 0.077 | **0.058** | 0.240 | **0.058** | 0.446 | **0.058** | 0.201 | **0.058** | 1.806 |
| wiki-Talk | 0.075 | 0.073 | 0.083 | 0.269 | 0.083 | 0.049 | 0.083 | 0.044 | 0.083 | | 0.058 | 0.077 | **0.057** | 0.330 | **0.057** | 0.356 | **0.057** | 0.172 | **0.057** | |
| Min | 0.012 | 0.014 | 0.013 | 0.030 | 0.013 | **0.011** | 0.013 | 0.042 | 0.013 | | 0.010 | 0.015 | **0.009** | 0.031 | **0.009** | 0.089 | **0.009** | 0.087 | **0.009** | |
| Average | 2.382 | 1.188 | 1.154 | 0.260 | **0.149** | 3.727 | 3.793 | 1.345 | 1.358 | | 37.554 | 11.508 | 11.069 | 0.399 | **0.345** | 118.835 | 109.014 | 42.632 | 33.699 | |
| Max | 82.960 | 36.193 | 36.139 | 3.092 | **2.449** | 139.544 | 142.731 | 51.125 | 51.046 | | 870.787 | 247.700 | 246.218 | **4.625** | 4.775 | 3 524.734 | 3 462.790 | 971.807 | 944.526 | |

**Table A.7** Average query times in µs for 100 000 random (left) and 200 000 mixed queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

| | | | | | | | | | ← random | mixed→ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | O’R+ pBiBFS | PReaCH | O’R+ PReaCH | PPL | O’R+ PPL | IP(d) | O’R+ IP(d) | BFL(d) | O’R+ BFL(d) | Matrix | O’R+ pBiBFS | PReaCH | O’R+ PReaCH | PPL | O’R+ PPL | IP(d) | O’R+ IP(d) | BFL(d) | O’R+ BFL(d) | Matrix |
| kron_logn12 | 0.120 | 0.120 | 0.084 | 0.045 | **0.028** | 0.635 | 0.251 | 3.014 | 0.300 | **0.019** | 0.194 | 0.197 | 0.139 | 0.039 | **0.029** | 1.121 | 0.433 | 1.694 | 0.516 | **0.016** |
| kron_logn16 | 0.776 | 0.821 | 0.628 | 0.155 | **0.081** | 5.463 | 2.297 | 7.194 | 2.215 | 1.006 | 1.682 | 1.788 | 1.351 | 0.143 | **0.088** | 12.943 | 5.317 | 12.614 | 4.932 | 0.110 |
| kron_logn17 | 0.606 | 0.607 | 0.237 | 0.180 | **0.092** | 4.040 | 1.001 | 3.617 | 0.482 | 1.326 | 1.258 | 1.333 | 0.389 | 0.176 | **0.097** | 10.361 | 2.362 | 5.061 | 0.932 | 0.173 |
| kron_logn20 | 7.327 | 4.273 | 3.998 | 0.276 | **0.150** | 53.853 | 26.239 | 59.319 | 26.638 | 2.430 | 23.183 | 12.651 | 11.753 | 0.285 | **0.207** | 171.584 | 82.016 | 187.036 | 81.384 | 0.393 |
| kron_logn21 | 10.364 | 2.458 | 1.064 | 0.311 | **0.179** | 44.566 | 27.592 | 33.318 | 17.089 | | 33.677 | 7.806 | 3.085 | 0.323 | **0.266** | 153.798 | 97.018 | 102.013 | 56.982 | |
| randn20-21 | 0.260 | 0.419 | 0.306 | 0.284 | 0.152 | 0.224 | 0.189 | 0.056 | 0.116 | 1.771 | 0.795 | 0.710 | 0.560 | 0.391 | 0.291 | 1.387 | 1.005 | 0.466 | 0.420 | 0.372 |
| randn20-22 | 3.007 | 2.552 | 2.363 | 0.431 | 0.339 | 2.494 | 2.445 | 0.355 | 0.420 | 1.060 | 19.356 | 12.227 | 12.090 | 0.806 | 0.824 | 28.163 | 28.046 | 4.882 | 4.904 | **0.338** |
| randn20-23 | 110.867 | 42.880 | 42.520 | 2.963 | **2.405** | 278.295 | 279.266 | 89.953 | 87.178 | 5.242 | 477.009 | 141.865 | 140.705 | 3.977 | **3.715** | 1 794.925 | 1 805.922 | 511.440 | 497.870 | **0.355** |
| randn23-24 | 0.355 | 1.033 | 0.784 | 0.413 | 0.224 | 0.310 | 0.266 | 0.056 | 0.171 | | 1.073 | 0.928 | 0.739 | 0.553 | 0.401 | 1.819 | 1.314 | 0.580 | 0.547 | |
| randn23-25 | 4.160 | 3.916 | 3.671 | 0.555 | 0.433 | 3.323 | 3.230 | 0.419 | 0.540 | | 27.927 | 16.212 | 16.092 | **0.901** | 0.920 | 36.916 | 36.914 | 6.130 | 6.175 | |
| delaunay_n15 | 0.133 | 0.175 | 0.077 | 0.133 | 0.054 | 0.229 | 0.102 | 0.091 | **0.045** | 0.064 | 0.685 | 0.166 | 0.120 | 0.129 | **0.080** | 2.846 | 1.406 | 0.420 | 0.307 | **0.056** |
| delaunay_n20 | 0.236 | 0.408 | 0.226 | 0.295 | 0.139 | 0.363 | 0.176 | 0.058 | 0.118 | 0.992 | 1.639 | 0.361 | 0.301 | 0.368 | **0.292** | 4.800 | 3.318 | 1.128 | 0.982 | 0.384 |
| delaunay_n22 | 0.280 | 0.650 | 0.315 | 0.373 | 0.181 | 0.417 | 0.220 | 0.056 | 0.154 | | 1.856 | 0.430 | **0.371** | 0.478 | 0.377 | 4.900 | 3.731 | 1.502 | 1.354 | |
| citeseer.scc | 0.057 | 0.077 | 0.057 | 0.235 | 0.057 | 0.050 | 0.057 | 0.043 | 0.057 | 1.323 | 0.105 | **0.095** | 0.106 | 0.294 | 0.106 | 0.243 | 0.106 | 0.124 | 0.106 | 0.385 |
| citeseerx | 0.208 | 0.253 | 0.238 | 0.369 | 0.138 | 0.164 | 0.163 | 0.079 | 0.134 | | 0.397 | 0.204 | 0.179 | 0.386 | **0.121** | 1.407 | 0.301 | 0.890 | 0.227 | |
| cit-Patents | 4.107 | 3.029 | 2.857 | 0.532 | **0.329** | 3.998 | 3.889 | 0.680 | 0.743 | | 121.444 | 60.488 | 59.975 | 1.301 | **1.255** | 236.166 | 240.466 | 59.523 | 58.810 | |
| go_uniprot | 0.108 | 0.123 | 0.103 | 0.394 | 0.101 | 0.069 | 0.101 | 0.042 | 0.101 | | 103.633 | 0.411 | 0.348 | 0.485 | **0.288** | 0.610 | 0.435 | 0.378 | 0.337 | |
| uniprotenc_22m | 0.067 | 0.068 | 0.068 | 0.260 | 0.068 | 0.045 | 0.068 | 0.043 | 0.068 | | 0.093 | 0.099 | **0.092** | 0.277 | **0.092** | 0.213 | **0.092** | 0.130 | **0.092** | |
| uniprotenc_100m | 0.132 | 0.165 | 0.134 | 0.419 | 0.134 | 0.098 | 0.134 | 0.043 | 0.134 | | 0.149 | 0.152 | 0.148 | 0.450 | 0.148 | 0.342 | 0.148 | 0.149 | 0.148 | |
| uniprotenc_150m | 0.154 | 0.210 | 0.156 | 0.454 | 0.156 | 0.116 | 0.156 | 0.044 | 0.156 | | 0.170 | 0.172 | 0.170 | 0.499 | 0.170 | 0.383 | 0.170 | 0.152 | 0.170 | |
| go_sub | 0.034 | 0.046 | 0.025 | 0.064 | 0.023 | 0.054 | 0.030 | 0.076 | **0.022** | 0.027 | 0.122 | 0.099 | 0.060 | 0.077 | **0.041** | 1.259 | 0.247 | 0.212 | 0.091 | **0.021** |
| pubmed_sub | 0.083 | 0.083 | 0.061 | 0.077 | 0.038 | 0.070 | 0.054 | 0.118 | **0.033** | **0.030** | 0.318 | 0.296 | 0.205 | 0.089 | **0.059** | 0.740 | 0.316 | 0.497 | 0.220 | **0.026** |
| yago_sub | 0.025 | 0.031 | 0.018 | 0.064 | 0.018 | 0.022 | 0.018 | 0.061 | **0.016** | 0.027 | 0.092 | 0.070 | 0.051 | 0.076 | **0.041** | 0.188 | 0.081 | 0.147 | 0.064 | **0.023** |
| citeseer_sub | 0.085 | 0.092 | 0.060 | 0.083 | 0.041 | 0.076 | 0.056 | 0.088 | **0.030** | 0.033 | 0.338 | 0.196 | 0.146 | 0.095 | **0.066** | 0.634 | 0.347 | 0.320 | 0.179 | **0.029** |
| arXiv | 0.379 | 0.377 | 0.253 | 0.085 | **0.049** | 1.215 | 0.622 | 1.774 | 0.322 | **0.026** | 0.743 | 0.752 | 0.433 | 0.068 | **0.049** | 3.424 | 1.606 | 1.795 | 0.877 | **0.023** |
| amaze | 0.015 | 0.017 | **0.014** | 0.041 | **0.014** | 0.030 | **0.014** | 1.262 | **0.014** | 0.019 | **0.015** | 0.020 | **0.015** | 0.039 | **0.015** | 0.057 | **0.015** | 0.086 | **0.015** | **0.015** |
| kegg | **0.015** | 0.017 | **0.015** | 0.043 | **0.015** | 0.035 | **0.015** | 1.542 | **0.015** | 0.018 | 0.016 | 0.020 | **0.015** | 0.039 | **0.015** | 0.060 | **0.015** | 0.086 | **0.015** | **0.015** |
| nasa | 0.026 | 0.030 | 0.023 | 0.054 | 0.021 | 0.041 | 0.024 | 0.094 | **0.019** | 0.024 | 0.048 | 0.051 | 0.037 | 0.056 | **0.025** | 0.838 | 0.088 | 0.209 | 0.036 | **0.019** |
| xmark | 0.029 | 0.030 | 0.023 | 0.059 | 0.023 | 0.049 | 0.029 | 0.188 | **0.020** | 0.025 | 0.037 | 0.046 | 0.029 | 0.053 | **0.022** | 0.239 | 0.040 | 1.118 | 0.024 | **0.020** |
| vchocyc | 0.017 | 0.016 | 0.017 | 0.058 | 0.016 | **0.014** | 0.016 | 0.059 | 0.016 | 0.031 | 0.020 | 0.025 | **0.019** | 0.056 | **0.019** | 0.136 | **0.019** | 0.080 | 0.020 | 0.023 |
| mtbrv | 0.016 | 0.016 | 0.017 | 0.058 | 0.016 | **0.013** | 0.016 | 0.060 | 0.016 | 0.031 | 0.021 | 0.026 | **0.020** | 0.056 | **0.020** | 0.132 | 0.021 | 0.085 | **0.020** | 0.024 |
| anthra | 0.017 | 0.019 | 0.017 | 0.064 | 0.017 | **0.013** | 0.017 | 0.054 | 0.017 | 0.035 | 0.020 | 0.027 | **0.019** | 0.061 | 0.020 | 0.161 | 0.020 | 0.075 | 0.020 | 0.025 |
| ecoo | 0.017 | 0.018 | 0.017 | 0.064 | 0.017 | **0.013** | 0.017 | 0.056 | 0.017 | 0.035 | **0.020** | 0.027 | **0.020** | 0.060 | **0.020** | 0.148 | **0.020** | 0.088 | **0.020** | 0.026 |
| agrocyc | 0.017 | 0.018 | 0.018 | 0.064 | 0.018 | **0.014** | 0.017 | 0.055 | 0.017 | 0.036 | **0.020** | 0.027 | **0.020** | 0.061 | **0.020** | 0.143 | **0.020** | 0.100 | **0.020** | 0.026 |
| human | 0.024 | 0.026 | 0.026 | 0.109 | 0.026 | **0.015** | 0.026 | 0.048 | 0.026 | 0.074 | 0.028 | 0.033 | **0.027** | 0.108 | **0.027** | 0.159 | **0.027** | 0.091 | **0.027** | 0.046 |
| p2p-Gnutella31 | 0.030 | 0.034 | 0.031 | 0.123 | 0.030 | **0.019** | 0.030 | 0.104 | 0.030 | 0.102 | 0.033 | 0.037 | **0.032** | 0.126 | **0.032** | 0.116 | 0.033 | 0.173 | **0.032** | 0.072 |
| email-EuAll | 0.058 | 0.069 | **0.056** | 0.175 | 0.057 | 0.075 | 0.057 | 0.451 | **0.056** | 5.497 | 0.062 | 0.072 | **0.060** | 0.200 | **0.060** | 0.218 | 0.061 | 0.134 | **0.060** | 0.318 |
| web-Google | 0.081 | 0.094 | 0.079 | 0.204 | 0.079 | 0.147 | 0.079 | 1.187 | **0.074** | 1.461 | 0.075 | 0.084 | 0.071 | 0.224 | 0.072 | 0.292 | 0.073 | 0.157 | **0.070** | 0.262 |
| soc-LiveJournal1 | 0.075 | 0.077 | 0.074 | 0.239 | 0.076 | 0.152 | 0.075 | 1.668 | **0.073** | 3.074 | 0.080 | 0.089 | **0.077** | 0.260 | 0.078 | 0.274 | 0.078 | 0.135 | **0.077** | 0.390 |
| wiki-Talk | 0.076 | 0.075 | 0.076 | 0.278 | 0.076 | **0.054** | 0.076 | 0.105 | 0.076 | | **0.087** | 0.095 | 0.088 | 0.319 | 0.088 | 0.228 | 0.088 | 0.123 | 0.088 | |
| Min | 0.015 | 0.016 | 0.014 | 0.041 | 0.014 | **0.013** | 0.014 | 0.042 | 0.014 | | **0.015** | 0.020 | **0.015** | 0.039 | **0.015** | 0.057 | **0.015** | 0.075 | **0.015** | |
| Average | 3.523 | 1.596 | 1.483 | 0.271 | **0.149** | 9.778 | 8.516 | 5.063 | 3.361 | | 19.964 | 6.351 | 6.102 | 0.352 | **0.258** | 60.352 | 56.433 | 22.002 | 17.541 | |
| Max | 110.867 | 42.880 | 42.520 | 2.963 | **2.405** | 278.295 | 279.266 | 89.953 | 87.178 | | 477.009 | 141.865 | 140.705 | 3.977 | **3.715** | 1 794.925 | 1 805.922 | 511.440 | 497.870 | |

**Table A.8** Speedups with `O'Reach` plus fallback over pure fallback algorithm. Values greater 1.00 are highlighted.

| | negative | | | | | | | positive | | | | | | | random | | | | | | | mixed | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | PReaCH | PPL | TF | IP(s) | IP(d) | BFL(s) | BFL(d) | PReaCH | PPL | TF | IP(s) | IP(d) | BFL(s) | BFL(d) | PReaCH | PPL | TF | IP(s) | IP(d) | BFL(s) | BFL(d) | PReaCH | PPL | TF | IP(s) | IP(d) | BFL(s) | BFL(d) |
| kron_logn12 | 1.16 | 1.65 | 2.99 | 0.97 | 1.00 | 1.87 | 2.11 | 1.43 | 1.11 | 2.30 | 2.45 | 2.69 | 3.13 | 3.33 | 1.43 | 1.58 | 2.58 | 2.34 | 2.53 | 9.86 | 10.06 | 1.42 | 1.32 | 2.39 | 2.41 | 2.59 | 3.06 | 3.28 |
| kron_logn16 | 0.83 | 1.45 | | 0.68 | 0.69 | 0.99 | 1.05 | 1.31 | 1.08 | | 2.29 | 2.44 | 2.42 | 2.57 | 1.31 | 1.90 | | 2.27 | 2.38 | 3.16 | 3.25 | 1.32 | 1.61 | | 2.28 | 2.43 | 2.41 | 2.56 |
| kron_logn17 | 0.92 | 1.57 | | 0.73 | 0.73 | 0.93 | 0.99 | 3.67 | 1.52 | | 4.34 | 4.47 | 5.14 | 5.72 | 2.57 | 1.94 | | 3.98 | 4.04 | 6.92 | 7.50 | 3.42 | 1.82 | | 4.27 | 4.39 | 4.88 | 5.43 |
| kron_logn20 | 0.92 | 1.85 | | 0.84 | 0.83 | 0.90 | 0.92 | 1.08 | 1.03 | | 2.07 | 2.09 | 2.21 | 2.30 | 1.07 | 1.84 | | 2.02 | 2.05 | 2.16 | 2.23 | 1.08 | 1.38 | | 2.07 | 2.09 | 2.21 | 2.30 |
| kron_logn21 | 0.90 | 1.80 | | 0.82 | 0.80 | 0.80 | 0.83 | 2.57 | 0.88 | | 1.65 | 1.59 | 1.75 | 1.79 | 2.31 | 1.74 | | 1.67 | 1.62 | 1.90 | 1.95 | 2.53 | 1.22 | | 1.65 | 1.59 | 1.75 | 1.79 |
| randn20-21 | 1.46 | 1.81 | 1.92 | 1.43 | 1.14 | 0.36 | 0.46 | 1.22 | 1.19 | 1.38 | 1.56 | 1.40 | 1.22 | 1.23 | 1.37 | 1.87 | 2.00 | 1.44 | 1.19 | 0.39 | 0.49 | 1.27 | 1.34 | 1.53 | 1.37 | 1.38 | 1.08 | 1.11 |
| randn20-22 | 1.08 | 1.27 | 1.50 | 1.03 | 1.02 | 0.94 | 0.83 | 1.00 | 0.90 | 1.15 | 1.01 | 1.00 | 1.01 | 1.00 | 1.08 | 1.27 | 1.52 | 1.05 | 1.02 | 0.94 | 0.85 | 1.01 | 0.98 | 1.23 | 1.02 | 1.00 | 1.00 | 1.00 |
| randn20-23 | 1.00 | 1.26 | | 1.05 | 0.98 | 1.01 | 1.00 | 1.01 | 0.97 | | 1.01 | 1.02 | 1.02 | 1.03 | 1.01 | 1.23 | | 1.03 | 1.00 | 1.02 | 1.03 | 1.01 | 1.07 | | 1.04 | 0.99 | 1.01 | 1.03 |
| randn23-24 | 1.16 | 1.87 | 2.08 | 1.38 | 1.17 | 0.27 | 0.33 | 1.17 | 1.22 | 1.43 | 1.44 | 1.40 | 1.19 | 1.20 | 1.32 | 1.84 | 2.06 | 1.47 | 1.17 | 0.26 | 0.33 | 1.26 | 1.38 | 1.62 | 1.41 | 1.38 | 1.03 | 1.06 |
| randn23-25 | 1.08 | 1.28 | 1.53 | 1.04 | 1.03 | 0.89 | 0.77 | 1.00 | 0.88 | 1.13 | 1.01 | 1.00 | 1.00 | 1.00 | 1.07 | 1.28 | 1.53 | 1.28 | 1.03 | 0.90 | 0.78 | 1.01 | 0.98 | 1.23 | 1.00 | 1.00 | 0.99 | 0.99 |
| delaunay_n15 | 2.24 | 2.16 | 2.71 | 2.80 | 2.14 | 0.88 | 1.19 | 0.96 | 0.94 | 1.34 | 2.14 | 1.93 | 1.34 | 1.34 | 2.26 | 2.46 | 3.02 | 2.75 | 2.24 | 1.73 | 2.02 | 1.39 | 1.62 | 1.83 | 2.12 | 2.02 | 1.33 | 1.37 |
| delaunay_n20 | 1.87 | 2.03 | 2.59 | 2.63 | 1.94 | 0.31 | 0.44 | 0.84 | 1.00 | 1.34 | 1.46 | 1.43 | 1.20 | 1.20 | 1.81 | 2.13 | 2.70 | 2.78 | 2.06 | 0.35 | 0.49 | 1.20 | 1.26 | 1.62 | 1.52 | 1.45 | 1.14 | 1.15 |
| delaunay_n22 | 1.52 | 2.05 | 2.68 | 2.51 | 1.88 | 0.25 | 0.35 | 0.72 | 1.00 | 1.29 | 1.29 | 1.28 | 1.17 | 1.16 | 2.06 | 2.06 | 2.66 | 2.53 | 1.89 | 0.26 | 0.36 | 1.16 | 1.27 | 1.61 | 1.45 | 1.31 | 1.11 | 1.11 |
| citeseer.scc | 1.35 | 4.11 | 0.42 | 0.93 | 0.89 | 0.62 | 0.78 | 0.79 | 2.70 | 2.68 | 2.85 | 3.43 | 1.37 | 1.60 | 1.35 | 4.11 | 0.51 | 0.90 | 0.87 | 0.60 | 0.75 | 0.89 | 2.77 | 2.02 | 2.00 | 2.29 | 0.98 | 1.16 |
| citeseerx | 0.99 | 2.55 | 2.97 | 1.00 | 0.93 | 0.41 | 0.41 | 1.12 | 3.47 | 16.99 | 4.12 | 5.89 | 4.17 | 5.79 | 1.07 | 2.67 | 3.01 | 1.06 | 1.01 | 0.54 | 0.59 | 1.14 | 3.20 | 9.66 | 3.51 | 4.67 | 3.18 | 3.93 |
| cit-Patents | 1.07 | 1.60 | 2.02 | 1.03 | 1.03 | 0.97 | 0.89 | 1.01 | 0.92 | 1.16 | 0.99 | 0.98 | 1.00 | 1.01 | 1.06 | 1.62 | 2.03 | 1.06 | 1.03 | 0.98 | 0.92 | 1.01 | 1.04 | 1.26 | 1.03 | 0.98 | 1.00 | 1.01 |
| go_uniprot | 1.22 | 3.92 | 1.08 | 0.70 | 0.70 | 0.34 | 0.43 | 1.14 | 1.23 | 1.38 | 1.45 | 1.48 | 1.26 | 1.33 | 1.20 | 3.89 | 1.07 | 0.68 | 0.68 | 0.33 | 0.42 | 1.18 | 1.68 | 1.39 | 1.36 | 1.40 | 1.06 | 1.12 |
| uniprotenc_22m | 1.02 | 3.82 | 1.20 | 0.68 | 0.68 | 0.50 | 0.65 | 1.05 | 3.80 | 2.49 | 4.60 | 4.63 | 2.41 | 2.72 | 1.00 | 3.83 | 1.20 | 0.67 | 0.67 | 0.49 | 0.63 | 1.08 | 3.02 | 1.78 | 2.33 | 2.33 | 1.14 | 1.42 |
| uniprotenc_100m | 1.25 | 3.14 | 1.43 | 0.76 | 0.75 | 0.25 | 0.33 | 0.92 | 3.83 | 2.95 | 4.21 | 4.27 | 1.70 | 1.97 | 1.24 | 3.14 | 1.43 | 0.74 | 0.73 | 0.25 | 0.32 | 1.03 | 3.03 | 2.10 | 2.27 | 2.31 | 0.84 | 1.01 |
| uniprotenc_150m | 1.35 | 2.91 | 1.50 | 0.77 | 0.76 | 0.22 | 0.29 | 0.87 | 3.67 | 2.97 | 3.97 | 4.07 | 1.50 | 1.72 | 1.35 | 2.92 | 1.51 | 0.76 | 0.75 | 0.22 | 0.28 | 1.01 | 2.93 | 2.14 | 2.25 | 2.25 | 0.74 | 0.89 |
| go_sub | 1.65 | 2.25 | 1.63 | 2.28 | 1.60 | 1.77 | 2.33 | 1.58 | 1.67 | 4.46 | 6.28 | 5.47 | 2.44 | 2.31 | 1.82 | 2.76 | 2.04 | 2.34 | 1.83 | 2.84 | 3.44 | 1.66 | 1.88 | 3.79 | 6.17 | 5.10 | 2.34 | 2.33 |
| pubmed_sub | 1.14 | 1.53 | 1.46 | 1.06 | 1.02 | 1.24 | 1.58 | 1.44 | 1.22 | 1.43 | 2.08 | 2.50 | 2.36 | 2.31 | 1.35 | 2.05 | 1.92 | 1.34 | 1.29 | 2.91 | 3.61 | 1.45 | 1.51 | 1.53 | 1.98 | 2.34 | 2.30 | 2.26 |
| yago_sub | 1.28 | 2.54 | 1.03 | 1.05 | 0.93 | 1.74 | 2.22 | 1.31 | 1.51 | 1.40 | 2.22 | 2.50 | 1.96 | 2.19 | 1.71 | 3.65 | 1.64 | 1.43 | 1.25 | 3.10 | 3.79 | 1.37 | 1.86 | 1.44 | 2.07 | 2.31 | 1.95 | 2.29 |
| citeseer_sub | 1.50 | 1.89 | 1.71 | 1.51 | 1.33 | 1.51 | 1.87 | 1.28 | 1.10 | 1.28 | 1.87 | 1.85 | 1.89 | 1.81 | 1.53 | 2.03 | 1.91 | 1.55 | 1.36 | 2.49 | 2.90 | 1.34 | 1.44 | 1.44 | 1.90 | 1.82 | 1.85 | 1.79 |
| arXiv | 1.16 | 1.61 | 2.67 | 1.25 | 1.24 | 1.28 | 1.42 | 1.91 | 0.98 | 1.61 | 2.12 | 2.31 | 2.12 | 2.09 | 1.49 | 1.75 | 2.42 | 1.92 | 1.95 | 4.72 | 5.52 | 1.74 | 1.38 | 2.04 | 1.95 | 2.13 | 2.07 | 2.05 |
| amaze | 1.05 | 2.31 | 0.84 | 0.85 | 0.83 | 2.98 | 3.72 | 1.68 | 3.40 | 2.38 | 9.11 | 9.80 | 7.76 | 11.21 | 1.17 | 2.91 | 1.35 | 1.94 | 2.10 | 87.92 | 90.01 | 1.34 | 2.62 | 1.62 | 3.59 | 3.86 | 4.15 | 5.85 |
| kegg | 0.97 | 2.14 | 0.85 | 1.01 | 0.94 | 2.72 | 3.54 | 1.66 | 3.34 | 2.28 | 9.17 | 10.03 | 7.29 | 10.88 | 1.17 | 2.87 | 1.36 | 2.24 | 2.33 | 105.29 | 105.07 | 1.31 | 2.62 | 1.61 | 3.73 | 3.96 | 4.01 | 5.79 |
| nasa | 1.05 | 1.77 | 1.50 | 1.34 | 1.04 | 1.63 | 2.09 | 1.33 | 1.99 | 5.17 | 13.37 | 11.02 | 6.38 | 7.89 | 1.35 | 2.61 | 2.23 | 2.06 | 1.71 | 4.36 | 4.91 | 1.37 | 2.26 | 3.55 | 11.49 | 9.55 | 4.87 | 5.88 |
| xmark | 1.18 | 2.01 | 1.57 | 1.41 | 1.36 | 1.85 | 2.36 | 1.87 | 2.19 | 3.99 | 9.48 | 9.68 | 97.90 | 103.17 | 1.31 | 2.53 | 1.97 | 1.81 | 1.71 | 8.77 | 9.40 | 1.60 | 2.37 | 2.69 | 6.13 | 5.99 | 44.76 | 45.83 |
| vchocyc | 0.93 | 2.95 | 1.84 | 0.87 | 0.75 | 2.21 | 2.81 | 1.72 | 2.86 | 5.57 | 38.80 | 16.40 | 5.34 | 6.43 | 0.97 | 3.59 | 2.31 | 0.97 | 0.85 | 3.03 | 3.68 | 1.32 | 2.94 | 3.31 | 15.10 | 6.99 | 3.25 | 4.10 |
| mtbrv | 0.87 | 2.75 | 1.46 | 0.83 | 0.69 | 2.05 | 2.62 | 1.49 | 2.36 | 4.31 | 30.50 | 12.26 | 4.61 | 6.25 | 0.98 | 3.59 | 1.95 | 0.99 | 0.82 | 3.12 | 3.76 | 1.27 | 2.81 | 2.88 | 14.41 | 6.18 | 3.09 | 4.20 |
| anthra | 0.96 | 2.85 | 1.78 | 0.72 | 0.71 | 1.91 | 2.42 | 1.84 | 3.19 | 22.65 | 25.90 | 18.64 | 4.87 | 6.37 | 1.10 | 3.72 | 2.42 | 0.79 | 0.78 | 2.63 | 3.18 | 1.36 | 3.12 | 9.41 | 10.81 | 7.93 | 2.81 | 3.74 |
| ecoo | 0.90 | 2.88 | 1.85 | 0.80 | 0.69 | 2.05 | 2.52 | 1.95 | 2.88 | 7.09 | 20.26 | 17.74 | 5.82 | 7.77 | 1.03 | 3.73 | 2.48 | 0.87 | 0.78 | 2.65 | 3.30 | 1.39 | 3.08 | 4.03 | 8.11 | 7.30 | 3.28 | 4.44 |
| agrocyc | 0.82 | 2.58 | 1.67 | 0.71 | 0.62 | 1.77 | 2.21 | 1.91 | 3.04 | 29.05 | 37.58 | 16.48 | 8.19 | 9.28 | 0.99 | 3.65 | 2.45 | 1.00 | 0.79 | 2.63 | 3.15 | 1.38 | 3.13 | 11.87 | 14.22 | 7.07 | 4.12 | 5.07 |
| human | 0.78 | 2.97 | 1.23 | 0.46 | 0.45 | 1.07 | 1.40 | 1.65 | 3.86 | 23.00 | 14.94 | 12.74 | 4.42 | 5.47 | 1.01 | 4.26 | 1.99 | 0.59 | 0.57 | 1.46 | 1.86 | 1.20 | 3.99 | 10.70 | 6.75 | 5.81 | 2.59 | 3.34 |
| p2p-Gnutella31 | 0.81 | 3.01 | 1.28 | 0.46 | 0.46 | 0.97 | 1.25 | 1.43 | 2.70 | 4.47 | 6.66 | 7.36 | 8.31 | 10.57 | 1.10 | 4.09 | 1.94 | 0.63 | 0.64 | 3.13 | 3.51 | 1.16 | 3.92 | 3.15 | 3.25 | 3.56 | 4.27 | 5.38 |
| email-EuAll | 1.02 | 2.62 | 0.59 | 0.91 | 0.89 | 0.57 | 0.73 | 1.40 | 4.92 | 4.05 | 8.04 | 8.24 | 3.86 | 4.75 | 1.23 | 3.07 | 1.04 | 1.30 | 1.31 | 8.06 | 8.04 | 1.20 | 3.32 | 2.48 | 3.50 | 3.58 | 1.72 | 2.23 |
| web-Google | 1.04 | 2.32 | 1.81 | 1.11 | 1.07 | 0.56 | 0.74 | 1.40 | 3.92 | 5.08 | 9.12 | 9.44 | 4.18 | 4.90 | 1.19 | 2.58 | 2.24 | 1.84 | 1.86 | 16.62 | 16.03 | 1.18 | 3.13 | 3.20 | 3.88 | 4.00 | 1.82 | 2.24 |
| soc-LiveJournal1 | 0.89 | 2.69 | 1.40 | 0.80 | 0.80 | 0.49 | 0.66 | 1.32 | 4.11 | 5.09 | 7.39 | 7.63 | 2.90 | 3.44 | 1.04 | 3.15 | 2.29 | 1.96 | 2.01 | 23.84 | 22.75 | 1.15 | 3.35 | 3.17 | 3.40 | 3.50 | 1.36 | 1.75 |
| wiki-Talk | 0.89 | 3.24 | 1.15 | 0.60 | 0.59 | 0.40 | 0.53 | 1.35 | 5.76 | 5.17 | 6.01 | 6.21 | 2.22 | 3.00 | 0.98 | 3.65 | 1.34 | 0.70 | 0.71 | 1.22 | 1.38 | 1.09 | 3.63 | 2.88 | 2.52 | 2.60 | 1.05 | 1.40 |
| Geometric Mean | 1.10 | 2.22 | | 1.00 | 0.92 | 0.88 | 1.06 | 1.33 | 1.90 | | 4.21 | 3.98 | 2.79 | 3.14 | 1.29 | 2.53 | | 1.35 | 1.26 | 2.12 | 2.40 | 1.29 | 2.04 | | 2.94 | 2.77 | 2.02 | 2.31 |
| Ratio Runtime Avgs | 1.03 | 1.75 | | 1.06 | 0.98 | 1.00 | 0.99 | 1.04 | 1.16 | | 1.07 | 1.09 | 1.15 | 1.27 | 1.08 | 1.82 | | 1.15 | 1.15 | 1.31 | 1.51 | 1.04 | 1.36 | | 1.09 | 1.07 | 1.14 | 1.25 |
| Average | 1.13 | 2.32 | | 1.09 | 0.98 | 1.11 | 1.35 | 1.41 | 2.25 | | 7.59 | 5.87 | 5.48 | 6.25 | 1.33 | 2.69 | | 1.52 | 1.41 | 7.96 | 8.22 | 1.33 | 2.23 | | 4.05 | 3.37 | 3.21 | 3.63 |

**Table A.9** Average query times in µs for 100 000 negative (left) and positive queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

← negative positive→

| | | O’R+ | | O’R+ | | O’R+ | | O’R+ | | O’R+ | | O’R+ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instance | `TF` | `TF` | `IP(s)` | `IP(s)` | `BFL(s)` | `BFL(s)` | `TF` | `TF` | `IP(s)` | `IP(s)` | `BFL(s)` | `BFL(s)` |
| kron_logn12 | 0.448 | 0.150 | 0.025 | 0.025 | 0.074 | 0.039 | 2.222 | 0.966 | 2.214 | 0.903 | 3.100 | 0.992 |
| kron_logn16 | | | 0.072 | 0.106 | 0.177 | 0.179 | | | 29.244 | 12.765 | 23.413 | 9.661 |
| kron_logn17 | | | 0.091 | 0.124 | 0.111 | 0.119 | | | 27.734 | 6.396 | 9.437 | 1.835 |
| kron_logn20 | | | 0.164 | 0.195 | 0.351 | 0.388 | | | 345.677 | 167.109 | 341.645 | 154.522 |
| kron_logn21 | | | 0.204 | 0.249 | 0.225 | 0.281 | | | 316.522 | 191.688 | 184.889 | 105.423 |
| randn20-21 | 0.287 | 0.150 | 0.319 | 0.223 | **0.044** | 0.123 | 0.501 | **0.364** | 2.832 | 1.815 | 0.837 | 0.687 |
| randn20-22 | 0.449 | **0.299** | 4.248 | 4.126 | 0.840 | 0.898 | 1.337 | 1.160 | 84.935 | 83.959 | 18.779 | 18.685 |
| randn20-23 | | | 198.518 | 188.459 | 96.362 | 95.814 | | | 4 720.272 | 4 656.298 | 1 683.989 | 1 656.785 |
| randn23-24 | 0.438 | 0.211 | 0.453 | 0.328 | **0.046** | 0.171 | 0.732 | **0.513** | 3.785 | 2.635 | 1.045 | 0.880 |
| randn23-25 | 0.607 | **0.396** | 5.394 | 5.178 | 0.950 | 1.064 | 1.589 | 1.404 | 113.423 | 112.633 | 23.804 | 23.875 |
| delaunay_n15 | 0.150 | 0.055 | 0.336 | 0.120 | **0.040** | 0.045 | 0.243 | 0.181 | 5.105 | 2.385 | 0.655 | 0.490 |
| delaunay_n20 | 0.367 | 0.141 | 0.588 | 0.223 | **0.038** | 0.124 | 0.664 | 0.495 | 8.549 | 5.864 | 2.085 | 1.739 |
| delaunay_n22 | 0.475 | 0.177 | 0.667 | 0.266 | **0.039** | 0.154 | 0.818 | 0.635 | 8.575 | 6.658 | 2.818 | 2.403 |
| citeseer.scc | **0.023** | 0.056 | 0.052 | 0.056 | 0.034 | 0.056 | 0.301 | 0.112 | 0.320 | 0.112 | 0.154 | 0.112 |
| citeseerx | 0.450 | 0.152 | 0.183 | 0.183 | 0.063 | 0.154 | 2.615 | 0.154 | 2.792 | 0.678 | 2.007 | 0.482 |
| cit-Patents | 1.078 | 0.533 | 6.259 | 6.049 | 1.845 | 1.904 | 10.640 | 9.168 | 701.034 | 708.037 | 245.211 | 244.524 |
| go_uniprot | 0.115 | 0.107 | 0.069 | 0.098 | **0.033** | 0.098 | 44.738 | 32.490 | 0.924 | 0.637 | 0.613 | 0.488 |
| uniprotenc_22m | 0.080 | 0.066 | 0.045 | 0.066 | **0.033** | 0.066 | 0.180 | **0.072** | 0.332 | **0.072** | 0.174 | **0.072** |
| uniprotenc_100m | 0.187 | 0.131 | 0.099 | 0.131 | **0.033** | 0.131 | 0.348 | 0.118 | 0.497 | 0.118 | 0.201 | 0.118 |
| uniprotenc_150m | 0.229 | 0.153 | 0.117 | 0.153 | **0.034** | 0.153 | 0.411 | 0.139 | 0.551 | 0.139 | 0.208 | 0.139 |
| go_sub | 0.042 | 0.026 | 0.089 | 0.039 | 0.044 | 0.025 | 0.338 | 0.076 | 4.302 | 0.685 | 0.385 | 0.158 |
| pubmed_sub | 0.069 | 0.047 | 0.070 | 0.066 | 0.055 | 0.044 | 0.228 | 0.160 | 1.482 | 0.714 | 1.260 | 0.535 |
| yago_sub | 0.024 | 0.023 | 0.026 | 0.024 | 0.037 | **0.021** | 0.085 | 0.060 | 0.250 | 0.113 | 0.178 | 0.091 |
| citeseer_sub | 0.066 | 0.038 | 0.100 | 0.066 | 0.046 | 0.030 | 0.155 | 0.121 | 1.247 | 0.666 | 0.600 | 0.317 |
| arXiv | 0.681 | 0.255 | 0.354 | 0.283 | 0.173 | 0.136 | 1.470 | 0.915 | 6.698 | 3.161 | 4.315 | 2.034 |
| amaze | **0.011** | 0.013 | **0.011** | 0.013 | 0.039 | 0.013 | 0.022 | **0.009** | 0.083 | **0.009** | 0.071 | **0.009** |
| kegg | **0.013** | 0.015 | 0.015 | 0.015 | 0.041 | 0.015 | 0.021 | **0.009** | 0.086 | **0.009** | 0.068 | **0.009** |
| nasa | 0.039 | **0.026** | 0.046 | 0.034 | 0.042 | **0.026** | 0.130 | 0.025 | 2.216 | 0.166 | 0.307 | 0.048 |
| xmark | 0.040 | 0.025 | 0.047 | 0.033 | 0.043 | **0.023** | 0.081 | 0.020 | 0.461 | 0.049 | 2.160 | 0.022 |
| vchocyc | 0.031 | 0.017 | 0.015 | 0.017 | 0.037 | 0.017 | 0.076 | **0.014** | 0.571 | 0.015 | 0.080 | 0.015 |
| mtbrv | 0.026 | 0.018 | 0.015 | 0.018 | 0.037 | 0.018 | 0.071 | **0.016** | 0.569 | 0.019 | 0.078 | 0.017 |
| anthra | 0.033 | 0.019 | 0.014 | 0.019 | 0.037 | 0.019 | 0.307 | **0.014** | 0.385 | 0.015 | 0.067 | **0.014** |
| ecoo | 0.034 | 0.019 | 0.015 | 0.019 | 0.038 | 0.019 | 0.100 | **0.014** | 0.308 | 0.015 | 0.084 | **0.014** |
| agrocyc | 0.035 | 0.021 | 0.015 | 0.021 | 0.037 | 0.021 | 0.402 | **0.014** | 0.559 | 0.015 | 0.118 | **0.014** |
| human | 0.040 | 0.033 | **0.015** | 0.033 | 0.035 | 0.033 | 0.496 | **0.022** | 0.328 | **0.022** | 0.096 | **0.022** |
| p2p-Gnutella31 | 0.047 | 0.037 | **0.017** | 0.037 | 0.035 | 0.036 | 0.115 | **0.026** | 0.173 | **0.026** | 0.215 | **0.026** |
| email-EuAll | 0.036 | 0.061 | 0.056 | 0.062 | **0.035** | 0.061 | 0.168 | **0.042** | 0.334 | **0.042** | 0.160 | **0.042** |
| web-Google | 0.135 | 0.074 | 0.086 | 0.077 | **0.039** | 0.070 | 0.246 | **0.048** | 0.442 | **0.048** | 0.202 | **0.048** |
| soc-LiveJournal1 | 0.099 | 0.071 | 0.057 | 0.072 | **0.034** | 0.069 | 0.298 | **0.058** | 0.432 | **0.058** | 0.170 | **0.058** |
| wiki-Talk | 0.095 | 0.083 | 0.050 | 0.083 | **0.033** | 0.083 | 0.297 | **0.057** | 0.344 | **0.057** | 0.127 | **0.057** |
| Min | | | **0.011** | 0.013 | 0.033 | 0.013 | | | 0.083 | **0.009** | 0.067 | **0.009** |
| Average | | | 5.342 | 5.059 | 2.496 | 2.506 | | | 156.016 | 145.532 | 62.338 | 54.329 |
| Max | | | 198.518 | 188.459 | 96.362 | 95.814 | | | 4 720.272 | 4 656.298 | 1 683.989 | 1 656.785 |

**Table A.10** Average query times in µs for 100 000 random (left) and 200 000 mixed queries (right). Highlighted results are the overall best/second-best after `Matrix` per query set over *all* tested algorithms.

| | | | | | | ← random | mixed→ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | `O'R+` | | `O'R+` | | `O'R+` | | `O'R+` | | `O'R+` | | `O'R+` |
| Instance | `TF` | `TF` | `IP(s)` | `IP(s)` | `BFL(s)` | `BFL(s)` | `TF` | `TF` | `IP(s)` | `IP(s)` | `BFL(s)` | `BFL(s)` |
| kron_logn12 | 0.995 | 0.385 | 0.631 | 0.269 | 2.933 | 0.297 | 1.349 | 0.564 | 1.128 | 0.469 | 1.594 | 0.520 |
| kron_logn16 | | | 6.212 | 2.731 | 6.794 | 2.148 | | | 14.705 | 6.440 | 11.845 | 4.923 |
| kron_logn17 | | | 5.507 | 1.385 | 3.515 | 0.508 | | | 13.973 | 3.269 | 4.795 | 0.983 |
| kron_logn20 | | | 54.122 | 26.731 | 54.180 | 25.126 | | | 173.231 | 83.873 | 170.936 | 77.473 |
| kron_logn21 | | | 45.584 | 27.339 | 30.225 | 15.939 | | | 158.059 | 95.937 | 92.489 | 52.906 |
| randn20-21 | 0.293 | 0.147 | 0.329 | 0.228 | **0.047** | 0.118 | 0.413 | **0.269** | 1.593 | 1.160 | 0.450 | 0.417 |
| randn20-22 | 0.452 | **0.297** | 4.161 | 3.978 | 0.840 | 0.895 | 0.921 | **0.747** | 44.638 | 43.895 | 9.833 | 9.813 |
| randn20-23 | | | 393.758 | 382.669 | 161.427 | 157.768 | | | 2 454.174 | 2 367.299 | 891.487 | 879.033 |
| randn23-24 | 0.449 | 0.218 | 0.450 | 0.306 | **0.044** | 0.173 | 0.610 | **0.377** | 2.139 | 1.513 | 0.556 | 0.542 |
| randn23-25 | 0.619 | **0.405** | 5.551 | 4.324 | 0.993 | 1.106 | 1.131 | 0.919 | 59.395 | 59.119 | 12.398 | 12.506 |
| delaunay_n15 | 0.168 | 0.055 | 0.371 | 0.135 | 0.077 | **0.045** | 0.212 | 0.116 | 2.742 | 1.292 | 0.359 | 0.271 |
| delaunay_n20 | 0.372 | 0.138 | 0.604 | 0.217 | **0.041** | 0.118 | 0.533 | 0.330 | 4.657 | 3.064 | 1.075 | 0.946 |
| delaunay_n22 | 0.479 | 0.180 | 0.671 | 0.265 | **0.040** | 0.154 | 0.669 | 0.415 | 4.744 | 3.268 | 1.429 | 1.290 |
| citeseer.scc | **0.029** | 0.057 | 0.052 | 0.057 | 0.035 | 0.057 | 0.215 | 0.106 | 0.213 | 0.106 | 0.104 | 0.106 |
| citeseerx | 0.448 | 0.149 | 0.184 | 0.174 | **0.078** | 0.143 | 1.587 | 0.164 | 1.543 | 0.440 | 1.048 | 0.329 |
| cit-Patents | 1.064 | 0.525 | 6.626 | 6.270 | 1.869 | 1.911 | 5.937 | 4.717 | 353.571 | 343.027 | 123.587 | 123.261 |
| go_uniprot | 0.109 | 0.101 | 0.069 | 0.101 | **0.033** | 0.101 | 22.618 | 16.328 | 0.540 | 0.397 | 0.342 | 0.322 |
| uniprotenc_22m | 0.081 | 0.068 | 0.046 | 0.068 | **0.033** | 0.068 | 0.163 | **0.092** | 0.213 | **0.092** | 0.104 | **0.092** |
| uniprotenc_100m | 0.191 | 0.134 | 0.098 | 0.134 | **0.033** | 0.134 | 0.311 | 0.148 | 0.337 | 0.148 | **0.124** | 0.148 |
| uniprotenc_150m | 0.236 | 0.156 | 0.118 | 0.156 | **0.034** | 0.156 | 0.365 | 0.170 | 0.382 | 0.170 | **0.126** | 0.170 |
| go_sub | 0.047 | 0.023 | 0.091 | 0.039 | 0.063 | **0.022** | 0.196 | 0.052 | 2.166 | 0.351 | 0.220 | 0.094 |
| pubmed_sub | 0.080 | 0.042 | 0.084 | 0.062 | 0.115 | 0.039 | 0.158 | 0.103 | 0.787 | 0.398 | 0.667 | 0.290 |
| yago_sub | 0.030 | 0.018 | 0.027 | 0.019 | 0.050 | **0.016** | 0.062 | 0.043 | 0.145 | 0.070 | 0.115 | 0.059 |
| citeseer_sub | 0.077 | 0.040 | 0.106 | 0.068 | 0.078 | 0.031 | 0.119 | 0.083 | 0.693 | 0.364 | 0.330 | 0.179 |
| arXiv | 0.751 | 0.311 | 1.291 | 0.674 | 1.929 | 0.408 | 1.112 | 0.545 | 3.571 | 1.832 | 2.253 | 1.088 |
| amaze | 0.019 | **0.014** | 0.028 | **0.014** | 1.232 | **0.014** | 0.024 | **0.015** | 0.053 | **0.015** | 0.061 | **0.015** |
| kegg | 0.020 | **0.015** | 0.034 | **0.015** | 1.545 | **0.015** | 0.024 | **0.015** | 0.056 | **0.015** | 0.060 | **0.015** |
| nasa | 0.044 | 0.020 | 0.055 | 0.027 | 0.083 | **0.019** | 0.092 | 0.026 | 1.150 | 0.100 | 0.181 | 0.037 |
| xmark | 0.044 | 0.022 | 0.053 | 0.029 | 0.174 | **0.020** | 0.067 | 0.025 | 0.261 | 0.043 | 1.106 | 0.025 |
| vchocyc | 0.037 | 0.016 | 0.016 | 0.016 | 0.049 | 0.016 | 0.063 | **0.019** | 0.294 | **0.019** | 0.064 | 0.020 |
| mtbrv | 0.031 | 0.016 | 0.016 | 0.016 | 0.050 | 0.016 | 0.058 | **0.020** | 0.307 | 0.021 | 0.063 | **0.020** |
| anthra | 0.041 | 0.017 | 0.014 | 0.017 | 0.045 | 0.017 | 0.183 | **0.019** | 0.219 | 0.020 | 0.056 | 0.020 |
| ecoo | 0.043 | 0.017 | 0.015 | 0.017 | 0.045 | 0.017 | 0.079 | **0.020** | 0.165 | **0.020** | 0.065 | **0.020** |
| agrocyc | 0.043 | 0.017 | 0.018 | 0.018 | 0.046 | 0.017 | 0.232 | **0.020** | 0.287 | **0.020** | 0.082 | **0.020** |
| human | 0.051 | 0.026 | **0.015** | 0.026 | 0.037 | 0.026 | 0.290 | **0.027** | 0.184 | **0.027** | 0.070 | **0.027** |
| p2p-Gnutella31 | 0.058 | 0.030 | **0.019** | 0.030 | 0.093 | 0.030 | 0.102 | **0.032** | 0.106 | 0.033 | 0.138 | **0.032** |
| email-EuAll | 0.059 | 0.057 | 0.074 | 0.057 | 0.452 | **0.056** | 0.150 | **0.060** | 0.213 | 0.061 | 0.103 | **0.060** |
| web-Google | 0.175 | 0.078 | 0.147 | 0.080 | 1.231 | **0.074** | 0.229 | 0.072 | 0.285 | 0.073 | 0.127 | **0.070** |
| soc-LiveJournal1 | 0.172 | 0.075 | 0.148 | 0.075 | 1.748 | **0.073** | 0.246 | 0.078 | 0.266 | 0.078 | 0.105 | **0.077** |
| wiki-Talk | 0.102 | 0.076 | **0.054** | 0.076 | 0.093 | 0.076 | 0.253 | 0.088 | 0.221 | 0.088 | 0.093 | 0.088 |
| Min | | | 0.014 | 0.014 | 0.033 | 0.014 | | | 0.053 | **0.015** | 0.056 | **0.015** |
| Average | | | 12.865 | 11.193 | 6.645 | 5.073 | | | 80.572 | 73.625 | 32.456 | 28.496 |
| Max | | | 393.758 | 382.669 | 161.427 | 157.768 | | | 2 454.174 | 2 367.299 | 891.487 | 879.033 |

# Approximation Algorithms for 1-Wasserstein Distance Between Persistence Diagrams

Samantha Chen
University of California at San Diego, La Jolla, CA, USA

Yusu Wang
University of California at San Diego, La Jolla, CA, USA

## Abstract

Recent years have witnessed a tremendous growth using topological summaries, especially the persistence diagrams (encoding the so-called persistent homology) for analyzing complex shapes. Intuitively, persistent homology maps a potentially complex input object (be it a graph, an image, or a point set and so on) to a unified type of feature summary, called the persistence diagrams. One can then carry out downstream data analysis tasks using such persistence diagram representations. A key problem is to compute the distance between two persistence diagrams efficiently. In particular, a persistence diagram is essentially a multiset of points in the plane, and one popular distance is the so-called 1-Wasserstein distance between persistence diagrams. In this paper, we present two algorithms to approximate the 1-Wasserstein distance for persistence diagrams in near-linear time. These algorithms primarily follow the same ideas as two existing algorithms to approximate optimal transport between two finite point-sets in Euclidean spaces via randomly shifted quadtrees. We show how these algorithms can be effectively adapted for the case of persistence diagrams. Our algorithms are much more efficient than previous exact and approximate algorithms, both in theory and in practice, and we demonstrate its efficiency via extensive experiments. They are conceptually simple and easy to implement, and the code is publicly available in github.

**2012 ACM Subject Classification** Theory of computation → Approximation algorithms analysis

**Keywords and phrases** persistence diagrams, approximation algorithms, Wasserstein distance, optimal transport

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.14

**Supplementary Material** *Software (Source Code)*: `https://github.com/chens5/w1estimators.git` archived at `swh:1:dir:03da011d5be7f5de530383a67da81499fb8b195c`

**Funding** This work is partially supported by National Science Foundation (NSF) via grants OAC-2039794 and IIS- 2050360.

**Acknowledgements** We want to thank Chen Cai for providing the reddit-binary and ModelNet10 datasets used in the experiments.

## 1 Introduction

Recent years have witnessed a tremendous growth using topological summaries, especially the persistence diagrams (encoding the so-called persistent homology) for analyzing complex shapes. Indeed, persistent homology is one of the most important development in the field of topological data analysis in the past two decades [11, 10]. Given an object, e.g, a mesh, an image, a point cloud, or a graph, by taking a specific view of how the object evolves (more formally, a filtration of it), persistent homology maps the input, a potentially complex object, to a topological summary, called the persistence diagram, which captures multiscale features of this objects w.r.t. this view. Persistent homology thus provides a unifying way of mapping complex objects to a common feature space: the space of persistence diagrams. One can then carry out data analysis tasks of the original objects, e.g, clustering or classifying a collection of graphs, in this feature space. Indeed, in the past decade, persistence diagram summaries have been used for a range of applications in various domains, e.g, in material science [5, 14, 23], neuroanatomy [17, 24], graphics [8, 29], medicine /biology [13, 27], etc.

A key component involved in such a persistent-homology based data analysis framework is to put a suitable metric on the space of persistence diagrams, and compute such distances efficiently. One classic distance measure developed for persistence diagrams is the $p$-th Wasserstein distance, and in practice, a popular choice for $p$ is $p = 1$, i.e, the 1-Wasserstein distance. This paper focuses on developing efficient, practical and light-weight algorithms to approximate the 1-Wasserstein distance for persistence diagrams.

In particular, a persistence diagram consists of a multiset of points in the plane, where each point $(b, d)$ corresponds to the creation and death of some topological feature w.r.t. some specific filtration (view) of the input object. Given two persistence diagrams $\mathsf{P}$ and $\mathsf{Q}$, the 1-Wasserstein distance between them, denoted by $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$, is similar to the standard 1-Wasserstein distance (also known as the earth-mover distance) between these two multisets of planar points, but with an important distinction where points are also allowed to be matched to points in the diagonal $\mathcal{L}$ (the line defined by equation $y = x$) in the plane. Intuitively, the topological features associated to points matched to the diagonal are considered as noise.

The 1-Wasserstein distance for persistence diagrams can be computed using the Hungarian algorithm [20] in $O(n^3)$ time where $n$ is the total number of points in the two persistence diagrams. This algorithm is implemented in the widely used Dionysus package [26]. In [18], Kerber et al. develops a more efficient algorithm to approximate the 1-Wasserstein distance between finite persistence diagrams within constant factors. Their algorithm is based on the auction algorithm of Bertsekas [4], but with a geometric twist: given that points in the persistence diagrams are all in the plane, they use the weighted kd-tree to provide more efficient search inside the auction algorithm. In [4], the time complexity of the auction algorithm is stated to be $O(A \cdot n^{1/2} \log(nC))$ where, in the case of persistence diagrams, $A$ is the number of possible pairings of points between persistence diagrams ($A = \Theta(n^2)$ in the worst case) and $C$ is $\max\{||x - y||_q\}$ over all possible pairings between persistence diagrams. While Kerber et al. did not provide an asymptotic time complexity for their approximation algorithm, they provided an empirical estimation of $O(n^{1.6})$ (not true asymptotic time complexity) by using linear regression on the observed running time versus the size of problems. They further show via extensive experiments that their approximation algorithm has a speed-up factor of 50 for small instances to a speed-up factor of 400 for larger instances in comparison to the Hungarian algorithm based implementation.

**Related work in optimal transport for Euclidean point sets.** As we will formally introduce in Section 2), 1-Wasserstein distance for persistence diagrams can be viewed as the standard 1-Wasserstein distance for discrete planar point sets with special inclusion of points in the diagonal. In what follows, to avoid confusion, we refer to the standard 1-Wasserstein distance between point sets as the *optimal transport (OT)* distance. Starting from [2] and [3], there has been a long line of work to approximate the OT-distance for Euclidean point sets using randomly shifted quadtrees (e.g, [7, 19, 15, 22, 28, 1]). In particular, we consider two such approaches, the $L_1$*-embedding approach* by [15], and the *flowtree approach* by [1]. The former maps an input point set $P$ to a certain count-vector $V^P$ with the help of a randomly shifted quadtree, and uses the $L_1$ distance $\|V^P - V^Q\|_1$ between two such count-vectors to approximate the OT-distance between $P$ and $Q$. The latter also uses a randomly shifted quadtree and embeds input points to quadtree cells. It then shows that a certain distance computed from an optimal OT-flow induced by the tree metric (which can be computed by a greedy algorithm in linear time) can approximate the OT-distance between the original point sets. Let $\Delta$ denote the spread of the union of two input point sets. Both approaches give an $O(\log \Delta)$-approximation of the OT-distance between original point sets, in time $O(n \log \Delta)$.

Recently, the idea of using metric trees to approximate OT-distance has also been extended to a more general unbalanced optimal transport problem (where $|P| \neq |Q|$) in [28]. In [28], Sato et al. develops an $O(n \log^2 n)$ time algorithm to approximate unbalanced optimal transport on tree metrics using dynamic programming.

**New work.** In practice, for applications such as nearest neighbor search, clustering and classification on large data sets, huge numbers of distance computations will be needed. The time complexity of the aforementioned algorithms for persistence diagrams using the Hungarian algorithm or the geometric variant of the Auction algorithm still causes a significant computational burden. In this paper, we aim to develop *near-linear time* approximation algorithms for the 1-Wasserstein distance between persistence diagrams. Specifically:

- In Section 3, we show how to modify the algorithms of [15] and [1] to approximate the 1-Wasserstein distances between persistence diagrams within the same approximation factor (Theorems 7 and 10). Note that in the literature (e.g, [18]), it is known that $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$ between two persistence diagrams can be computed by (i) first augmenting $\mathsf{P}$ and $\mathsf{Q}$ to be $\widehat{\mathsf{P}} = \mathsf{P} \cup \pi(\mathsf{Q})$ and $\widehat{\mathsf{Q}} = \mathsf{Q} \cup \pi(\mathsf{P})$, respectively, where $\pi(x)$ projects a point $x$ to its nearest neighbor in the diagonal $\mathcal{L}$; and then (ii) compute the OT-distance between $\widehat{\mathsf{P}}$ and $\widehat{\mathsf{Q}}$, although it is important to note that the cost of matching two diagonal points needs to be set to be 0, instead of the standard Euclidean distance. However, this requires the modification of the cost for diagonal points; in addition, this also needs to modify a diagram $\mathsf{P}$ depending on which other diagram $\mathsf{Q}$ it is to be compared with. We instead develop a modification where such projection is not needed.
- Our modified approaches maintain the simplicity of the original approximation algorithms and are easy to implement. In comparison to approximation for unbalanced optimal transport presented in [28], our modified approaches are specific to persistence diagrams and the data structures needed for both of our approaches are much simpler than those of [28]. Our code is publicly available in github. In Section 4, we present various experimental results of our new algorithms. We show that both are orders of magnitude faster than previous approaches, although at the price of worsened approximation error. However, note that in practice, the approximate factors are rather small, not as large as the worst case approximation factor. We also note that the modified flowtree algorithm achieves a more accurate approximation of the 1-Wasserstein distance for persistence diagrams than the modified $L_1$-embedding approach empirically, although the latter is significantly faster than the former. However, the $L_1$-embedding approach is easier to combine with proximity search data structures e.g, locality sensitive hashing (LSH), given that each input persistence diagram is mapped to a vector and the distance computation is the $L_1$-distance between two such vectors.

## 2 Preliminaries

In this section, we first introduce the persistence diagrams and the 1-Wassertein distance between them, which is related to the optimal transport distance (standard 1-Wasserstein distance) for Euclidean point sets. We next describe two existing approximation algorithms for optimal transport distance [1, 15] based on the use of randomly shifted quadtrees. Our new algorithms (in section 3) will be based on these two approximation algorithms.

## 2.1 Persistence Diagrams and 1-Wasserstein distance

We first give a brief introduction of persistent homology and its associated persistence diagram summary. See [10] for a more detailed treatment of these topics. Suppose we are given a topological space X. A filtration of X is a growing sequence of sub-spaces

$$\mathbb{F}: \quad \emptyset = \mathrm{X}_0 \subseteq \mathrm{X}_1 \subseteq \mathrm{X}_2 \subseteq \cdots \mathrm{X}_m = \mathrm{X}$$

which can be viewed as a specific way to inspect X. For example, a popular way to generate a filtration of X is by taking some meaningful descriptor function $f : \mathrm{X} \to \mathbb{R}$ on X, and take the growing sequence of sub-level sets $\mathrm{X}_a := f^{-1}(-\infty, a] = \{x \in \mathrm{X} \mid f(x) \leq a\}$ as $a$ increases to be the filtration. Now given a filtration $\mathbb{F}$, through its course, new topological features (e.g, components, independent loops and voids, which are captured by the so-called homology classes) will sometimes appear and sometimes disappear. The persistent homology encodes the *birth* and *death* of such features in the *persistence diagram* $\mathsf{dgm}\mathbb{F}$. In particular, $\mathsf{dgm}\mathbb{F}$ consists of a *multiset* of points in the plane, that is, a set of points with multiplicities, where each point $(b, d)$ with multiplicity $m$ intuitively means that $m$ independent topological features (homology classes) are created in $\mathrm{X}_b$ and killed in $\mathrm{X}_d$. Thus, we also refer to $b$ and $d$ as the *birth-time* and *death-time*. The *persistence* of this feature is $|d - b|$ which is the lifetime of this feature. We refer to points in the persistence diagram as *persistent-points*.

Note that, in general, persistent-points lie above the diagonal $\mathcal{L} = \{(x, x) \mid x \in \mathbb{R}\}$ in the plane. Points closer to the diagonal $\mathcal{L}$ have lower lifetime (persistence) and thus are less important, with a point $(x, x) \in \mathcal{L}$ intuitively meaning a feature with persistence 0.

To compare two persistence diagrams P and Q, intuitively, we wish to find a one-to-one correspondence between their multiset of points (and thus between the features they capture). However, the two sets may be of different cardinality, and we also wish to allow a persistent-point from one diagram to be "noise" and not present in the other diagram, which can be captured by allowing this point $p = (p.x, p.y)$ to be matched to its nearest neighbor projection $\pi(p)$ in $\mathcal{L}$. Let $\pi : \mathbb{R}^2 \to \mathcal{L}$ be this projection, where $\pi(p) := (\frac{p.x+p.y}{2}, \frac{p.x+p.y}{2})$. The following $p$-th Wasserstein distance essentially captures this intuition [10].

▶ **Definition 1** (p-Wasserstein distance for persistence diagrams). *Given a persistence diagram* P*, its* augmentation $\mathsf{aug}(\mathsf{P})$ *consists of* P *together with all points in* $\mathcal{L}$ *each with infinite multiplicity. Given two persistence diagrams* P *and* Q*, with their augmentations* $\mathsf{aug}(\mathsf{P})$ *and* $\mathsf{aug}(\mathsf{Q})$*, respectively, the p-Wassertein distance between them is*

$$\mathrm{d}^{\mathrm{per}}_{\mathrm{W,p}}(\mathsf{P}, \mathsf{Q}) := \inf_{\mu : \mathsf{aug}(\mathsf{P}) \to \mathsf{aug}(\mathsf{Q})} \left( \sum_{p \in \mathsf{aug}(\mathsf{P})} \|p - \mu(p)\|_q^p \right)^{1/p}, \tag{1}$$

*where* $\mu : \mathsf{aug}(\mathsf{P}) \to \mathsf{aug}(\mathsf{Q})$ *ranges over all possible bijections among the two sets.*

Note that $q$ is used to denote the inner $L_p$-norm. If $p = \infty$, the $\infty$-Wasserstein distance is the classic *bottleneck distance* between persistence diagrams [9][18]. In this paper, we are interested in the case when $p = 1$. It turns out that an equivalent definition (which we will use in this paper) is as follows:

▶ **Definition 2** (1-Wasserstein distance for persistence diagrams, version 2). *Given two point sets $A$ and $B$ in $\mathbb{R}^2$, an* augmented (perfect) matching *for them is a subset* $\Gamma \subset \big(A \cup \pi(B)\big) \times \big(B \cup \pi(A)\big)$ *such that (i) each $a \in A$ or $b \in B$ appears in exactly one pair in $\Gamma$, and (ii) each $(a, b) \in \Gamma$ is of the following three forms: (1) $a \in A, b \in B$, (2) $a \in A, b = \pi(a) \in \pi(A)$, or (3) $a = \pi(b) \in \pi(B), b \in B$.*

*Given two persistence diagrams* $\mathsf{P}$ *and* $\mathsf{Q}$*, the 1-Wasserstein distance between them is:*

$$\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P},\mathsf{Q}) := \min_{\Gamma} \sum_{(p,q)\in\Gamma} \|p-q\|_p, \tag{2}$$

*where* $\Gamma$ *ranges over all possible augmented matchings for* $\mathsf{P}$ *and* $\mathsf{Q}$*.*

## 2.2 Relation to optimal transport

Readers may have already noticed the similarity between Definition 1 with the standard $p$-th Wasserstein distance between two probability measures. To avoid confusion, from now on we refer to 1-Wassertein distance as *optimal transport* so as to differentiate from the use of 1-Wasserstein distance of persistence diagrams.

▶ **Definition 3** (Optimal transport)**.** *Given a finite metric space* $(X, d_X)$ *and two measures* $\mu, \nu \in X \to \mathbb{R}$*, the* optimal transport *between them is*

$$\mathrm{d}_{\mathrm{OT}}(\mu,\nu) := \min_{\tau : X\times X\to\mathbb{R}} \sum_{x,y\in X} \tau(x,y)\cdot d_X(x,y), \tag{3}$$

*where* $\tau$*, called a* transport plan *or a* flow*, is a measure on* $X \times X$ *whose marginals equal to* $\mu$ *and* $\nu$*, respectively; that is,* $\tau(\cdot, Y) = \mu(\cdot)$ *and* $\tau(X, \cdot) = \nu(\cdot)$*.*

Given a multiset of points $A$ in the plane, note that we can view this as a discrete measure supported on points in $A$, such that for each subset $S$ of $A$, $\mu_A(S) = \sum_{a\in S} c_a\delta_a$ where $c_a$ is the multiplicity of $a$ in $A$, while $\delta_a$ is the Dirac measure supported at $a$. Hence in what follows, we sometimes abuse the notations and equate a multiset of points with the discrete measure induced by it, and talk about optimal transport between two multisets of points.

As shown in [18], one can consider $\widehat{\mathsf{P}} := \mathsf{P} \cup \pi(\mathsf{Q})$ and $\widehat{\mathsf{Q}} := \mathsf{Q} \cup \pi(\mathsf{P})$ and modify the Euclidean distance so that $d(x, y) = 0$ for $x, y \in \pi(P) \cup \pi(Q)$ to obtain a modified pseudo-metric space $(\mathbb{R}^2, d)$. In this case, $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P},\mathsf{Q})$ becomes the optimal transport between the discrete measures induced by $\widehat{\mathsf{P}}$ and $\widehat{\mathsf{Q}}$ under this modified pseudo-metric.

We can also relate $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P},\mathsf{Q})$ to the optimal transport between the discrete measures induced by $\widehat{\mathsf{P}}$ and $\widehat{\mathsf{Q}}$ with the following observation (simple proof is in Appendix A):

▶ **Observation 4.** *Let* $\mu_{\widehat{\mathsf{P}}}$ *be the discrete measure induced by* $\widehat{\mathsf{P}}$ *and* $\nu_{\widehat{\mathsf{Q}}}$ *be the discrete measure induced by* $\widehat{\mathsf{Q}}$*. Then* $\mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}}) \le 2 \cdot \mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P},\mathsf{Q})$*.*

Given two discrete measures $\mu, \nu \in X \times \mathbb{R}$ on a finite metric space $(X, d_X)$, computing the optimal transport distance can be reduced to finding the optimal min-cost flow on a complete bipartite graph using combinatorial flow algorithms as described in [20]. In our setting later, $\mu$ and $\nu$ will both be induced by point sets in $\mathbb{R}^2$, and $d_X$ is the standard Euclidean distance.

## 2.3 Quadtree-based approximation algorithms for optimal transport

In this section, we briefly review two algorithms to approximate the optimal transport for two discrete measures $\mu$ and $\nu$. Both of these algorithms use a randomly shifted quadtree, which we introduce first.

**Randomly-shifted quadtree.** Let $X \subseteq \mathbb{R}^d$ be a finite set of points (for our setting, $d = 2$ for persistence diagrams). To simplify the description, we will assume that the minimum pairwise distance between any two points in $X$ is 1 and that $X$ is contained in $[0, \Delta]^d$ (where $\Delta$, the ratio of the diameter of $X$ over the minimum pairwise distance, is also called the *spread* of $X$). First, let $H_0 = [-\Delta, \Delta]^d$ be the hypercube with side length $2\Delta$ which is centered at the origin. Now shift $H_0$ by a random vector whose coordinates are from $[0, \Delta]$ to obtain $H$. Note that $H$ still encloses $X$ as $H$ has side length $2\Delta$.

Construct a tree of hypercubes by letting $H$ be associated to the root and halving $H$ along each dimension. Recurse on the resulting sub-hypercubes that contain at least one point from $X$, and stop when a hypercube contains exactly one point from $X$. Each leaf node of resulting quadtree $\mathrm{T}_X$ contains exactly one point in $X$, and there are exactly $|X|$ leaves. The resulting quadtree $\mathrm{T}_X$ has at most $O(\log(d\Delta))$ levels. To see that $\mathrm{T}_X$ has at most $O(\log(d\Delta))$ levels, consider the depth $i$ of some internal node. We know that the hypercube associated with the node has a side length of $\frac{\Delta}{2^i}$ and the distance between any two points in the hypercube, $c$, is less than or equal to $\frac{\Delta\sqrt{d}}{2^i}$. Then $i \leq \log(d\Delta)$ so there are at most $O(\log(d\Delta))$ levels in $\mathrm{T}_X$. Additionally, the size of $\mathrm{T}_X$ is $O(|X| \log(d\Delta))$. It can be constructed in $\tilde{O}(|X| \log(d\Delta))$ time where $\tilde{O}$ includes term polynomial in $\log |X|$. We set the root level as level $\log \Delta + 1$ and subsequent levels are labeled as $\log \Delta, \log \Delta - 1, \ldots$. The weight of each tree edge between level $\ell + 1$ and level $\ell$ is $2^\ell$. Note that the quadtree cell has side length $2^\ell$ at level $\ell$.

**Approximation algorithm 1: $L_1$-embedding via $\mathrm{T}_X$.** Given two discrete measures $\mu$ and $\nu$, let $X$ be the union of their support [1]. Construct the randomly shifted quadtree $\mathrm{T}_X$ as described above; $X$ is sometimes omitted from the subscript when its choice is clear from the context. Given a tree node $v \in \mathrm{T}_X$, its level is denoted by $\ell(v)$. We will abuse the notation slightly and use $v$ also to denote the quadtree cell (which is a hypercube of size length $2^{\ell(v)}$). Given a discrete $\mu$, then $\mu(v)$ denotes the total measure of points from $\mu$ contained within this quadtree cell, namely, the total size of points with multiplicity counted from $\mu$ within this quadtree cell. We can now map $\mu$ to a vector $\mathrm{V}^\mu$ where each index corresponds to a tree node $v \in \mathrm{T}_X$, and $\mathrm{V}^\mu[v]$ has coordinates $2^{\ell(v)}\mu(v)$. Similarly, map $\nu$ to vector $\mathrm{V}^\nu$. Then Indyk and Thaper [15] showed that $\|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1 = \sum_{v \in \mathrm{T}} 2^{\ell(v)} |\mu(u) - \nu(v)|$ gives an approximation to the optimal transport $\mathrm{d}_{\mathrm{OT}}(\mu, \nu)$ in expectation.

▶ **Theorem 5** ([15]). *Given two discrete measures $\mu, \nu$ such that $\mathsf{supp}(\mu) \cup \mathsf{supp}(\nu) \subseteq \mathbb{R}^2$ and $s = |\mathsf{supp}(\mu) \cup \mathsf{supp}(\nu)|$, using a randomly shifted quadtree, $\|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1$ can be calculated in time $O(s \log \Delta)$ and there are constants $C_1, C_2$ such that $C_1 \cdot \mathrm{d}_{\mathrm{OT}}(\mu, \nu) \leq E[\|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1] \leq C_2 \cdot \log \Delta \mathrm{d}_{\mathrm{OT}}(\mu, \nu)$. Here, $E[\cdot]$ stands for the expectation.*

We note that it also turns out that $\|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1$ gives exactly the optimal transport between $\mu$ and $\nu$ along the tree metric induced by $\mathrm{T}_X$. Specifically, for each $v \in \mathrm{T}_X$, set its weight to be $w(v) = 2^{\ell(v)}$. Then for any $x, x' \in X$, define $d_T(x, x')$ to be the total weight of the unique tree path connecting the quadtree leaf $v_x$ (containing $x$) and leaf $v_{x'}$ (containing $x'$). Then the optimal transport between $\mu$ and $\nu$ w.r.t. metric $d_T$, denoted by $\mathrm{d}_{\mathrm{OT},\mathrm{d_T}}(\mu, \nu)$, satisfies that $\mathrm{d}_{\mathrm{OT},\mathrm{d_T}}(\mu, \nu) = \|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1$.

[1] Note that in general, $X$ can be a superset of the support of $\mu$ and $\nu$. Indeed, if there are a set of $m$ measures and we perform kNN queries for a query measure, it is more convenient to set $X$ as the union of support of all these measures and build only a single quadtree $\mathrm{T}_X$.

Furthermore, we can consider that this optimal transport $\mathrm{d}_{\mathrm{OT},\mathrm{d}_\mathrm{T}}$ is generated by a following *greedy-flow* $f^*_G : X \times X \to \mathbb{R}$: Starting from leaf-nodes, we will match up as many unmatched points $\mu \cap v$ to $\nu \cap v$ as we can within each node $v$, and pass the remaining unmatched portion to its parent. In general, each tree node $v \in \mathrm{T}$ will have a $\mu$-demand $\widehat{\mu}(v)$ and $\widehat{\nu}(v)$, which collect all unmatched measure from its $2^d$ child nodes. $\mu$-demand (resp. $\nu$-demand) at a leaf node $v$ is initialized to be $\mu(v)$ (resp. $\nu(v)$). We then match these demand as much as we can and pass on $|\widehat{\mu}(v) - \widehat{\nu}(v)|$ to its parent as unmatched $\mu$-measure, or unmatched $\nu$-measure, whichever is left. Note that a greedy-flow $f_G$ is not unique, but it tuns out that any such greedy-flow (greedy transport plan) gives rise to the optimal transport distance between $\mu$ and $\nu$ w.r.t. the tree metric $d_T$ (See [16] for more detail): i.e,

$$(\|\mathrm{V}^\mu - \mathrm{V}^\nu\|_1 =)\ \mathrm{d}_{\mathrm{OT},\mathrm{d}_\mathrm{T}}(\mu,\nu) = \sum_{x,x' \in X} f^*_G(x,x') d_T(x,x'). \tag{4}$$

**Approximation algorithm 2: Flowtree.** The flowtree algorithm by [1] is based on the previous approach. The only modification is that, consider a greedy-flow $f^*_G$ as described above. Instead of using the tree metric $d_T$ to compute the optimal transport distance, the *flowtree estimate* computes the cost of this flow using the standard Euclidean distance:

$$\mathrm{d}^{\mathrm{flow}}_{\mathrm{OT}}(\mu,\nu) = \sum_{x,x' \in X} f^*_G(x,x')\|x - x'\|. \tag{5}$$

Comparing Equation (4) to the above equation, the difference is minor ($d_T(x,x')$ versus $\|x - x'\|$). However, in practice, $\mathrm{d}^{\mathrm{flow}}_{\mathrm{OT}}$ appears to provide a much more accurate estimate to the optimal transport distance $\mathrm{d}_{\mathrm{OT}}(\mu,\nu)$ w.r.t. the Euclidean distance. Unfortunately, unlike $\mathrm{d}_{\mathrm{OT},\mathrm{d}_\mathrm{T}}$, which can be computed as a $L_1$-distance between two specific vectors, to compute $\mathrm{d}^{\mathrm{flow}}_{\mathrm{OT}}$, we now have to compute a greedy-flow $f^*_G$ explicitly (which can be done linear in the size of quadtree; however conceptually, this is not as simple as $L_1$-distance). Overall, we have the following result:

▶ **Theorem 6** ([1]). *Given two discrete measures $\mu,\nu$ such that $\mathsf{supp}(\mu) \cup \mathsf{supp}(\nu) \subseteq \mathbb{R}^2$ and $s = |\mathsf{supp}(\mu) \cup \mathsf{supp}(\nu)|$, using a randomly shifted quadtree, $\mathrm{d}^{\mathrm{flow}}_{\mathrm{OT}}(\mu,\nu)$ can be computed in time $O(s \log \Delta)$ and there are constants $C_1, C_2$ such that $C_1 \cdot \mathrm{d}_{\mathrm{OT}}(\mu,\nu) \leq E[\mathrm{d}^{\mathrm{flow}}_{\mathrm{OT}}(\mu,\nu)] \leq C_2 \cdot \log\Delta \cdot \mathrm{d}_{\mathrm{OT}}(\mu,\nu)$. Here, $E[\cdot]$ stands for the expectation.*

## 3 Approximating 1-Wasserstein distances for persistence diagrams

We now present two algorithms to approximate the 1-Wasserstein distance for persistence diagrams, based on the approximation schemes of optimal transport in Section 2.3. Note that the results here are developed for the $L_2$ norm and through the equivalence of norms, can be generalized to any $L_p$ norm and only changes the constant factor in the distortion induced by each approximation.

### 3.1 Approximation algorithms via $L_1$ embedding

#### 3.1.1 Description of the new quadtree-based $L_1$-embedding

Let $\mathsf{P}$ and $\mathsf{Q}$ be two persistence diagrams and let $X = \mathsf{P} \uplus \mathsf{Q}$, the disjoint union of $\mathsf{P}$ and $\mathsf{Q}$. In what follows, for simplicity of presentation, we assume that the minimum distance between any two distinct points in $X$, as well as between any point in $X$ with a point in the diagonal $\mathcal{L}$, is 1. The latter constraint can be removed with some extra care on handling leaf nodes in the quadtree. Assume w.l.o.g that $\Delta$ is a power of 2.

Partition the (randomly shifted) hypercube $H$ described in section 2.3 into grids where the cells have side length $\Delta, \Delta/2, \ldots, 2^i, \ldots, 2, 1, \frac{1}{2}$. Note that each cell at the lowest level can contain at most one point, and if a leaf contains a point then it cannot intersect the diagonal $\mathcal{L}$. Let $\mathrm{T}_X$ be the resulting quadtree, where leaves are all cells that contain exactly one point from $X$. We further use $G_i$ to denote the set of quadtree cells with side-length $2^i$ (i.e, those in level-$i$); we refer to $G_i$ as the level-$i$ grid. Note that the size of the quadtree is $O(|X|\log\Delta)$. Additionally, we call a cell a *terminal cell* if it intersects the diagonal $\mathcal{L}$; otherwise, it is *non-terminal.*

Now for each grid $G_i$, construct a vector $\mathrm{V}_i^{\mathsf{P}}$ with one coordinate per cell, where each coordinate counts the number of points in the corresponding cell. The vector representation $\mathrm{V}^{\mathsf{P}}$ for $\mathsf{P}$ is then the concatenation of all these vectors $2^i\mathrm{V}_i^{\mathsf{P}}$ where $2^i$ is the cell side length for grid $G_i$:

$$\mathrm{V}^{\mathsf{P}} = [\frac{1}{2}\mathrm{V}_{-1}^{\mathsf{P}}, \mathrm{V}_0^{\mathsf{P}}, 2\mathrm{V}_1^{\mathsf{P}}, \ldots, 2^i\mathrm{V}_i^{\mathsf{P}}, \ldots, ]$$

Construct the vector $\mathrm{V}^{\mathsf{Q}}$ similarly. We use $p_k$ to denote the value of coordinate $k$ in $\mathrm{V}_i^{\mathsf{P}}$ and $q_k$ to denote the value of coordinate $k$ in $\mathrm{V}_i^{\mathsf{Q}}$. Now, we will describe a *modified-$L_1$ distance* $|\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T$ for these vectors, which is similar to the $L_1$ norm. To compute $|\mathrm{V}_i^{\mathsf{P}} - \mathrm{V}_i^{\mathsf{Q}}|_T$, we will define $|p_k - q_k|_T$. There are two cases for the $|p_k - q_k|_T$ to consider:
Case 1: if coordinate $k$ is not associated with a terminal cell, then use $|p_k - q_k|$ for $|p_k - q_k|_T$.
Case 2: if coordinate $k$ is associated with a terminal cell, then set $|p_k - q_k|_T = 0$.
Then we have $|\mathrm{V}_i^{\mathsf{P}} - \mathrm{V}_i^{\mathsf{Q}}|_T = \sum_{k=1}^{|G_i|} |p_k - q_k|_T$, and

$$\widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q}) := |\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T = \sum_{i=-1}^{\log_2\Delta} 2^i|\mathrm{V}_i^{\mathsf{P}} - \mathrm{V}_i^{\mathsf{Q}}|_T. \tag{6}$$

**An equivalent $L_1$-distance formulation.** We introduce the above vector representation and the modified $L_1$-distance as it is more convenient for later theoretical analysis. However, algorithmically, we wish to have a true $L_1$-embedding. It turns out that an equivalent formulation is as follows: Let $\widehat{G}_i$ denote the level-$i$ quadtree cells *that do not intersect the diagonal $\mathcal{L}$.* We then compute a vector representation $\widehat{\mathrm{V}}^{\mathsf{P}}$ (resp. $\widehat{\mathrm{V}}^{\mathsf{Q}}$) restricted only to cells in $\bigcup \widehat{G}_i$. In other words, all entries corresponding to cells intersecting the diagonal are ignored in constructing $\widehat{\mathrm{V}}^{\mathsf{P}}$ and $\widehat{\mathrm{V}}^{\mathsf{Q}}$. We then have that

$$\|\widehat{\mathrm{V}}^{\mathsf{P}} - \widehat{\mathrm{V}}^{\mathsf{Q}}\|_1 = |\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T = \widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q}). \tag{7}$$

That is, our quadtree-induced distance $\widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q})$ is a $L_1$-distance for suitably constructed vectors. Nevertheless, we use the definition as in Equation (6) to simplify proofs later.

It is easy to see that the construction takes the same time as the $L_1$-embedding approach described in Section 2.3. We now show that the $L_1$-distance $\widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q})$ approximates the 1-Wasserstein distance $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$ for the persistence diagrams. The main results for this $L_1$-embedding approach are summarized as follows, and we prove the approximation bound in Section 3.1.2.

▶ **Theorem 7.** *Given persistence diagrams* $\mathsf{P}$ *and* $\mathsf{Q}$ *such that* $s = |\mathsf{P}| + |\mathsf{Q}|$*, we can compute* $\widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q}) = |\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T$ *in time* $O(s\log\Delta)$ *using the randomly shifted quadtree. Furthermore, the expected value of* $\widehat{d}_{L_1}(\mathsf{P}, \mathsf{Q})$ *is an* $O(\log\Delta)$*-approximation of the 1-Wasserstein distance* $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$*; i.e, there are constants* $c_1$ *and* $c_2$ *such that* $c_1 \cdot \mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q}) \leq E[|\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T] \leq c_2 \log\Delta \cdot \mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$.

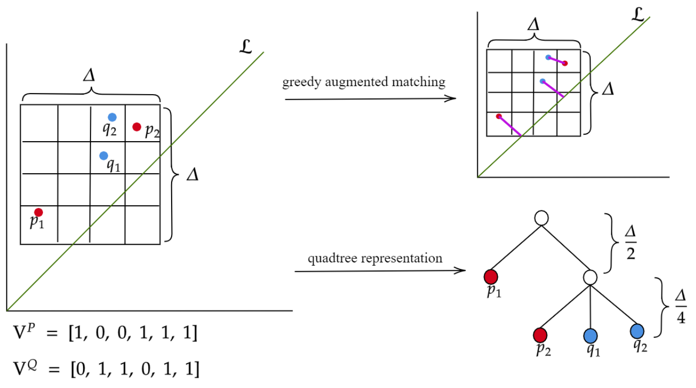


**Figure 1** A simple example of the vector and quadtree representations of the persistence diagrams $P = \{p_1, p_2\}$ and $Q = \{q_1, q_2\}$.

### 3.1.2 Approximation guarantees

Our approximation bound in Theorem 7 follows from Lemmas 8 and 9 below. To prove these lemmas, we will first introduce a *greedy augmented matching*.

**Greedy augmented matching $\widehat{\Gamma}$.** We construct the following augmented matching (recall Definition 2) $\widehat{\Gamma} \subseteq \big(\mathsf{P} \cup \pi(\mathsf{Q})\big) \times \big(\mathsf{Q} \cup \pi(\mathsf{P})\big)$ in a *bottom-up greedy manner*: Starting from the level $i = -1$, we will aim to match points in $\mathsf{P} \uplus \mathsf{Q}$ as much as we can within each level $G_i$. Those remaining unmatched points will then be considered at the next level $G_{i+1}$: in particular, within each *non-terminal cell* in $G_{i+1}$, we will match the maximal possible unmatched points in $\mathsf{P}$ to unmatched points in $\mathsf{Q}$ so far, and pass the remainder unmatched points (which can now only come from either $\mathsf{P}$ or $\mathsf{Q}$, but not both) to its parents. Within a *terminal cell* $v$ in $G_{i+1}$, we match every unmatched point $p$ from $\mathsf{P} \cap v$ and from $\mathsf{Q} \cap v$ to its closest point $\pi(p) \in \mathcal{L}$ in the diagonal. Finally, at the root cell (in level $\log \Delta$), any unpaired points from either $\mathsf{P}$ or $\mathsf{Q}$ will be paired to the diagonal, as the root is a terminal cell. Note that by construction, at any level $i$, $|\mathrm{V}_i^{\mathsf{P}} - \mathrm{V}_i^{\mathsf{Q}}|_T$ is exactly the number of points from $\mathsf{P} \uplus \mathsf{Q}$ that could not be matched at level $i$ or below under such a greedy augmented matching and will subsequently need to be matched in grid $G_j$, $j \geq i+1$. See Figure 1 for a simple illustration.

▶ **Lemma 8.** *There is a constant $C$ such that* $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q}) \leq C \cdot |\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T$.

**Proof.** Consider the greedy augmented matching $\widehat{\Gamma}$ we described above, induced by pairing points or pairing points with their diagonal projection greedily within the same cells of the grids $G_{-1}, G_0, G_1, \ldots$ in a bottom-up manner. First, given $(p, q) \in \widehat{\Gamma}$, we say that $(p, q)$ is paired in level-$i$, if the lowest level any quadtree cell containing both $p, q$ is level-$i$; intuitively, $(p, q)$ are paired greedily in this cell in level-$i$. We now use $\widehat{\Gamma}_i \subseteq \widehat{\Gamma}$ to denote the set of pairs from level-$i$. For each level $i$, let $cost(i) = \sum_{(p,q)\in\widehat{\Gamma}_i} \|p - q\|_2$ be the total cost incurred by all those pairs from $\widehat{\Gamma}_i$, and obviously,

$$\sum_i cost(i) = \sum_{(p,q)\in\widehat{\Gamma}} \|p - q\|_2 \geq \mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q}), \tag{8}$$

where the right inequality holds as $\mathrm{d}_{\mathrm{W},1}^{\mathrm{per}}(\mathsf{P}, \mathsf{Q})$ is the smallest total cost of any augmented matching and the greedy augmented matching $\widehat{\Gamma}$ is an augmented matching.

There are no pairings induced in the grid $G_{-1}$ (of size 1/2) since the minimum inter-point distance is 1 and the minimum distance between a point and its diagonal is 1 as well. Hence we have that $|\mathrm{V}^{\mathsf{P}}_{-1} - \mathrm{V}^{\mathsf{Q}}_{-1}|_T = |\mathsf{P}| + |\mathsf{Q}|$. Since all points from $P$ and $Q$ will remain unpaired in level $-1$ (i.e, within cells of grid $G_{-1}$), we then have that there are exactly

$$|\mathsf{P}| + |\mathsf{Q}| - |\mathrm{V}^{\mathsf{P}}_0 - \mathrm{V}^{\mathsf{Q}}_0|_T = |\mathrm{V}^{\mathsf{P}}_{-1} - \mathrm{V}^{\mathsf{Q}}_{-1}|_T - |\mathrm{V}^{\mathsf{P}}_0 - \mathrm{V}^{\mathsf{Q}}_0|_T$$

total number of points from $\mathsf{P} \uplus \mathsf{Q}$ that can either be paired to each other or matched to the diagonal in grid cells $G_0$. As the maximal distance between any pair of matched points is $2^i\sqrt{2}$ within a cell in $G_i$, the maximum cost incurred by all matched points in $G_0$ is $cost(0) \leq \sqrt{2}(|\mathrm{V}^{\mathsf{P}}_{-1} - \mathrm{V}^{\mathsf{Q}}_{-1}|_T - |\mathrm{V}^{\mathsf{P}}_0 - \mathrm{V}^{\mathsf{Q}}_0|_T)$. In general, the maximum cost of the matched points in $G_i$ is

$$cost(i) \leq 2^i \cdot \sqrt{2}(|\mathrm{V}^{\mathsf{P}}_{i-1} - \mathrm{V}^{\mathsf{Q}}_{i-1}|_T - |\mathrm{V}^{\mathsf{P}}_i - \mathrm{V}^{\mathsf{Q}}_i|_T).$$

Hence combining Equation (8), the total cost (i.e, $\sum_{(p,q)\in\widehat{\Gamma}} ||p-q||_2$) of the greedy augmented matching $\widehat{\Gamma}$, is bounded from above by:

$$\sum_{(p,q)\in\widehat{\Gamma}} ||p-q||_2 \leq \sum_{i=0}^{\log_2 \Delta+1} 2^i \cdot \sqrt{2}(|\mathrm{V}^{\mathsf{P}}_{i-1} - \mathrm{V}^{\mathsf{Q}}_{i-1}|_T - |\mathrm{V}^{\mathsf{P}}_i - \mathrm{V}^{\mathsf{Q}}_i|_T) \leq 2\sqrt{2}|\mathrm{V}^P - \mathrm{V}^Q|_T.$$

By the right inequality of Equation (8), the claim then follows. ◀

▶ **Lemma 9.** *There is a constant $C'$ such that the expected value of $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$ is bounded by $E[\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})] = E[|\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T] \leq C' \cdot \log\Delta \cdot \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$.*

**Proof.** Set $\widehat{\mathsf{P}} = \mathsf{P} \cup \pi(\mathsf{Q})$ and $\widehat{\mathsf{Q}} = \mathsf{Q} \cup \pi(\mathsf{P})$ as before. For a given grid $G_i$ and some coordinate $k$ in $\mathrm{V}^{\mathsf{P}}_i$ and $\mathrm{V}^{\mathsf{Q}}_i$, let $p_k$ be the value of coordinate $k$ in $\mathrm{V}^{\mathsf{P}}_i$ and $q_k$ be the value of coordinate $k$ in $\mathrm{V}^{\mathsf{Q}}_i$. Analogously, let $\mathrm{V}^{\widehat{\mathsf{P}}}$ and $\mathrm{V}^{\widehat{\mathsf{Q}}}$ be the vector representations w.r.t multisets $\widehat{P}$ and $\widehat{Q}$, respectively, and let $\widehat{p}_k$ (resp. $\widehat{q}_k$) be the value of coordinate $k$ in $\mathrm{V}^{\widehat{\mathsf{P}}}_i$ (resp. $\mathrm{V}^{\widehat{\mathsf{Q}}}_i$). Note that $\widehat{p}_k \geq p_k$ and $\widehat{q}_k \geq q_k$.

(i) Now if $\widehat{p}_k > p_k$ or $\widehat{q}_k > q_k$, then there exists at least one point $x \in \pi(\mathsf{P}) \cup \pi(\mathsf{Q})$ in the cell $v$ associated with coordinate $k$. In other words, this cell $v$ must be a terminal cell, and $|p_k - q_k|_T = |\widehat{p}_k - \widehat{q}_k|_T = 0$.

(ii) Otherwise if the conditions in (i) do not hold, it must be that $\widehat{p_k} = p_k$ and $\widehat{q_k} = q_k$, in which case we also have that $|p_k - q_k|_T = |\widehat{p}_k - \widehat{q}_k|_T$.

Combining (i) and (ii) we then have that $|\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T \leq |\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}|_T$.

On the other hand, by the definition of metric $|\cdot|_T$, we know that $|\widehat{p}_k - \widehat{q}_k|_T \leq |\widehat{p}_k - \widehat{q}_k|$, implying that $|\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}|_T \leq \|\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}\|_1$. Let $\mu_{\widehat{\mathsf{P}}}$ and $\nu_{\widehat{\mathsf{Q}}}$ be the discrete measures induced by $\widehat{\mathsf{P}}$ and $\widehat{\mathsf{Q}}$ respectively. By Theorem 5, there is some constant $C$ such that $E[\|\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}\|_1] \leq C \cdot \log\Delta \cdot \mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}})$. From Observation 4, $\mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}}) \leq 2 \cdot \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$. Therefore,

$$E[|\mathrm{V}^{\mathsf{P}} - \mathrm{V}^{\mathsf{Q}}|_T] \leq E[|\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}|_T] \leq E[\|\mathrm{V}^{\widehat{\mathsf{P}}} - \mathrm{V}^{\widehat{\mathsf{Q}}}\|_1] \leq 2 \cdot C \cdot \log\Delta \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q}).$$

The lemma then follows. ◀

## 3.2 Approximation algorithm via flowtree

We now propose an alternative approximation algorithm for $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$. The high level idea is the same as the flowtree algorithm described in Section 2.3: in particular, we first compute the optimal flow for points in $\mathsf{P}$ and $\mathsf{Q}$ along the randomly shifted quadtree T as constructed earlier, but now with the modification that a point can be paired to diagonal. It turns out that this leads to the same greedy augmented matching $\widehat{\Gamma}$ we described at the beginning of Section 3.1.2. Then, similar to flowtree, under this greedy augmented matching $\widehat{\Gamma}$, we use the Euclidean distance between a pair of matched points (instead of using the tree-distance as for $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$) to measure the cost of each pair of matched points. This leads to the following *modified flowtree estimate*:

$$\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q}) = \sum_{(p,q)\in\widehat{\Gamma}} ||p-q||_2, \tag{9}$$

and in our second algorithm, we will use $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q})$ as an approximation of the true 1-Wasserstein distance $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$.

From an implementation point of view, unlike $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$, which can be computed as the $L_1$-distance between two vectors, we now must explicitly compute the greedy augmented matching, $\widehat{\Gamma}$ between $\mathsf{P}$ and $\mathsf{Q}$. (Note that this greedy augmented matching was only used in proving the approximation guarantee for $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$, and not needed for its computation.) Computing this greedy augmented matching (and calculating its cost) takes the same time as computing the greedy flow in the original flowtree algorithm 2.3. Furthermore, it is easy to see that in the proof of Lemma 8, we in fact showed that $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q}) \leq \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q}) \leq C\cdot|\mathrm{V}^P - \mathrm{V}^Q|_T$ (see Eqn (8)). Combining this with Theorem 5, we thus obtain the following approximation result for this modified flowtree estimate.

▶ **Theorem 10.** *Given two persistence diagrams* $\mathsf{P}$ *and* $\mathsf{Q}$ *where* $s = \max(|P|,|Q|)$ *and* $\Delta$ *is the spread of point set* $\mathsf{P}\cup\mathsf{Q}$*, we can compute* $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q})$ *in time* $O(s\log\Delta)$ *using a randomly shifted quadtree. Additionally, the expected value of* $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q})$ *is an* $O(\log\Delta)$*-approximation of the 1-Wasserstein distance* $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$*; i.e. there are constants* $C_1$ *and* $C_2$ *such that* $C_1\cdot\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q}) \leq E[\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q})] \leq C_2\cdot\log\Delta\cdot\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$.

**Remark.** We remark that while these two approximation schemes, $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$ and $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1_F}(\mathsf{P},\mathsf{Q})$, have similar approximation guarantees for $\mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(\mathsf{P},\mathsf{Q})$, in practice, the modified flowtree based approach has much higher accuracy. This is consistent with the performance of flowtree algorithm versus the $L_1$-embedding approach for general optimal distance [1]. In contrast, the benefit of the $L_1$ embedding approach is that it is easy to compute $\widehat{d}_{L_1}(\mathsf{P},\mathsf{Q})$. Also, each persistence diagram is now mapped to a vector representation, and the distance is $L_1$-distance among these vector representations. One could combine this with methods such as locality-sensitive-hashing for more efficient approximate $k$-nearest neighbor queries. In general, such a $L_1$-norm also makes $\widehat{d}_{L_1}(\cdot,\cdot)$ potentially more suitable for downstream machine learning pipelines.

## 4 Experimental results

We evaluate both the runtime and accuracy of the modified flowtree and $L_1$-embedding against the *Hera* method of [18]. We use the implementation of Hera provided in the GUDHI library [25] for testing. We run the experiments using an Intel Core i7-1065G7 CPU @ 1.30 GHz and 12.0GB RAM. Additionally, the implementations for both the modified flowtree and $L_1$-embedding were done in C++ (wrapped in python for evaluation) and are based on the code provided in [1].

**Datasets.** For our experiments, we use both synthetic persistence diagrams, as well as persistence diagrams generated from real data. For synthetic data, we generate two sets of persistence diagrams: called "*synthetic-uniform*" and "*synthetic-Gaussian*", which are generated by a uniform sample and a sample w.r.t. a Gaussian distribution on the birth-death plane to obtain the persistence diagrams, respectively. For real datasets, we use persistence diagrams generated from the so-called Reddit data sets (which is a collection of graphs) [6], and from the ModelNet10 [30] dataset of shapes. Details of these datasets are in Appendix B.

**Speed comparison.** We compare the running time of our new approximation algorithms with that of Hera [18] – note that we do not directly compare with the exact algorithm by Dionysus, because as reported by [18], Hera is 50 times to 400 times faster than Dionysus. Note that Hera is also an approximation algorithm, and there is a parameter $\varepsilon$ to adjust its approximation factor $(1 + \varepsilon)$. By setting this parameter to be very small ($\varepsilon = 0.01$), we use the distance computed by Hera as ground truth later when we measure approximation accuracy; see Table 1.

The comparison of the running times of our approaches with that of Hera (for a range of different approximation factors) can be found in Figure 2, which summarizes the runtime for each method on a **log-scale** using both randomly generated diagrams as well as the reddit-binary dataset (real persistence diagrams from graphs). We compare the speed of our modified flowtree and $L_1$ embedding against Hera where the parameter $\varepsilon$ for Hera is set to be 300000. However, note that as one relaxes the approximation parameter $\varepsilon$, the speed of Hera in fact does not improve much (as shown in Figure 2). Thus our speed gains remain no matter which choice of $\varepsilon$ we use for Hera. Additionally, the true approximation error for Hera also does not decrease much; see Table 1. To get the approximation error for Hera, we find the true Wasserstein distance by using the `wasserstein_distance` function from the GUDHI library which uses the Python Optimal Transport library [12] and is based on ideas from [21]. The results in Figure 2 indicate that the modified flowtree approach is between 50 and 1000 times faster than Hera and the difference increases as the size of the diagrams increases. Similarly, the $L_1$ embedding approach is between 150 and 4900 times faster than Hera. Both are order of magnitudes faster than Hera; but the price to pay is that the approximation factor is worse for our approach as shown in Appendix C Figure 3.

**Approximation accuracy comparison.** To measure the accuracy of the approximation of both the modified flowtree and $L_1$ embedding approach, we first measure the average relative error and standard deviation of both methods on all datasets using $L_1$, $L_2$, and $L_\infty$ as the ground metrics. In particular, given a ground truth distance $d$ and an approximate distance $\tilde{d}$, the relative error is $\rho(d) = \frac{|d-\tilde{d}|}{d}$. As mentioned earlier, we use the output of Hera for $\varepsilon = 0.01$ as ground truth, and compare our approximated distance with that. The results are summarized in Figure 3 and a detailed table of the average relative error and standard deviation is in Table 2. Overall, while our modified flowtree is slower than the $L_1$-embedding approach, it achieves much better approximation error. For our experiments, we generate a quadtree only once for all persistence diagrams in a given dataset and calculate error for the approximated distances for the single quadtree. However, note that by constructing several quadtrees and averaging the distance estimates or taking the smallest estimated, we could potentially reduce the approximation error.

In addition to the average error of our approximate distances, we can also consider the efficacy of both methods in terms of nearest neighbor search and ranking accuracy. To evaluate nearest neighbor search, we first split the set of persistence diagrams into query diagrams and candidate diagrams. Then, we measure recall@$m$ accuracy where recall@$m$ is defined as the fraction of queries that have the true nearest neighbor within the top $m$-ranked candidates returned by the evaluated method. The results are reported in Figure 4. For ranking accuracy, the detailed results are in Appendix C. To summarize, the modified flowtree approach is more accurate than the $L_1$ embedding approach both in terms of nearest neighbor search and closeness to the true ranking of candidate diagrams for a fixed query diagram. Both approaches appear to have a lower degree of accuracy on diagrams where there a higher proportion of points near the diagonal. This may be due to the increased possibility of erroneously matching points to the diagonal.

In summary, we note that both our new approximation algorithms significantly improve the speed previously best-known Hera algorithm by orders of magnitudes, but with worse approximation factors. Empirically, the approximation factors remain constant despite our theoretical results suggestion an $O(\log \Delta)$-approximation. In particular, the relative approximation error of flowtree is often smaller than 0.50 for $L_2$ ground metric (see Appendix C Table 2).

**Table 1** Comparison of the maximum allowed relative error with the average experimental relative error for Hera on the reddit-binary dataset. The relative error was calculated using the GUDHI library's `wasserstein_distance` function which uses the Python Optimal Transport library to compute exact 1-Wasserstein distance.

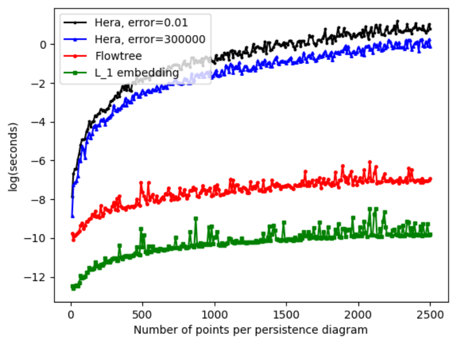
| Maximum allowed relative error | Average relative error |
|---|---|
| 0.1 | 0.00043768 |
| 1.0 | 0.003331 |
| 100000 | 0.0076055 |
| 200000 | 0.0076055 |

**Table 2** Average error and standard deviation for all datasets. We abbreviate the $L_1$ embedding approach to embd and the flowtree approach to ft.

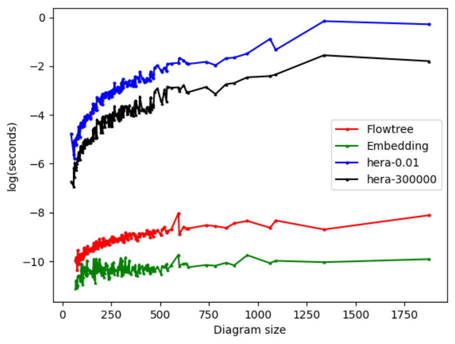
| | | $L_1$ | | $L_2$ | | $L_\infty$ | |
|---|---|---|---|---|---|---|---|
| | | embd | ft | embd | ft | embd | ft |
| synthetic-uniform | Avg. Error | 2.058 | 0.2846 | 3.161 | 0.2664 | 4.536 | 0.2595 |
| | Std. Dev. | 1.034 | 0.3891 | 1.189 | 0.3488 | 1.164 | 0.3176 |
| synthetic-Gaussian | Avg. Error | 1.341 | 0.3358 | 2.136 | 0.2860 | 3.035 | 0.2251 |
| | Std. Dev. | 0.3647 | 0.1809 | 0.4139 | 0.1519 | 0.3669 | 0.1178 |
| reddit-binary | Avg. Error | 2.112 | 0.2899 | 3.089 | 0.3080 | 3.921 | 0.2854 |
| | Std. Dev. | 1.275 | 0.3859 | 2.100 | 0.5126 | 2.427 | 0.4801 |
| ModelNet10 | Avg. Error | 2.189 | 0.7331 | 2.438 | 0.4929 | 3.061 | 0.9399 |
| | Std. Dev. | 0.9543 | 0.4132 | 0.8136 | 0.3171 | 1.051 | 0.4448 |

## 5 Concluding remarks

In this paper, we presented two algorithms for fast approximation of the 1-Wasserstein distance between persistence diagrams based on $L_1$ embedding. While the relative error incurred by both algorithms is higher than that of Hera, the runtime is significantly faster. We also observe that approximation methods introduced are more accurate on persistence diagrams with a lower proportion of points near the diagonal.

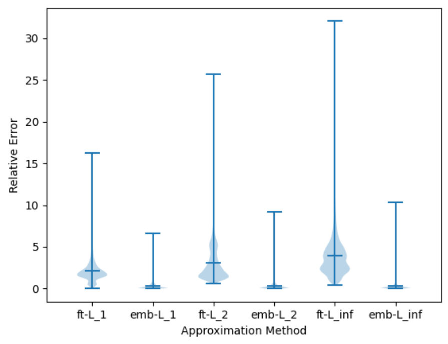


**(a)** synthetic-uniform.

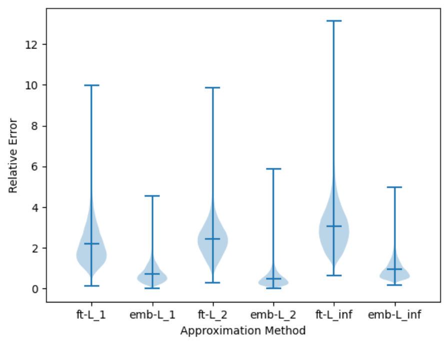


**(b)** reddit-binary.

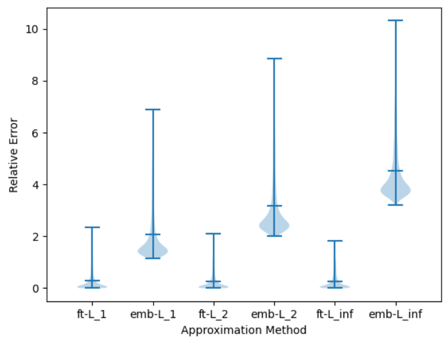
**Figure 2** Comparison of the runtimes of HERA, flowtree, and $L_1$ embedding using both generated and real data with $L_2$ as the ground metric.



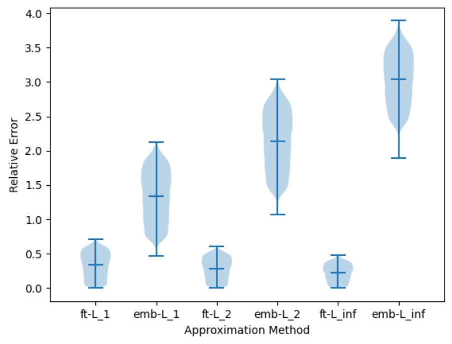
**(a)** reddit-binary.



**(b)** ModelNet10.

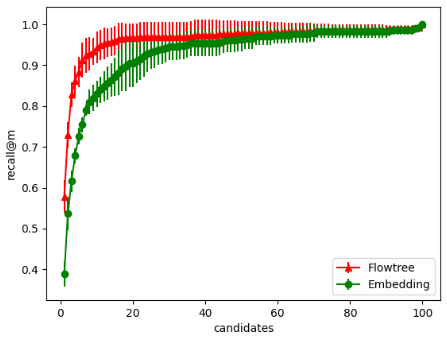


**(c)** synthetic-uniform.

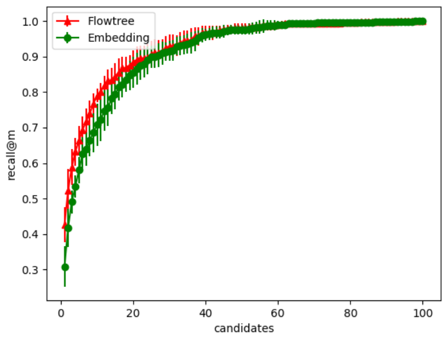


**(d)** synthetic-Gaussian.

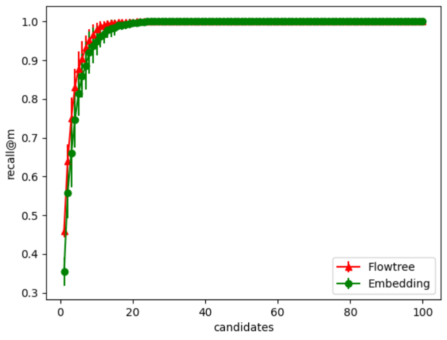
**Figure 3** Relative error for flowtree and quadtree approximations over $L_1$, $L_2$, and $L_\infty$ ground metrics for all datasets.



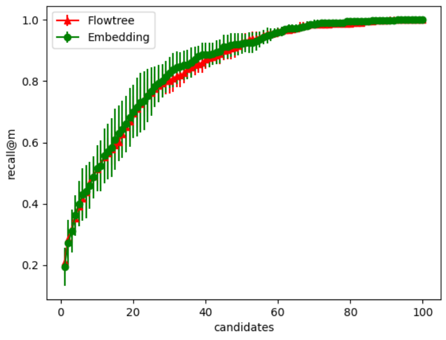
**(a)** Recall@$m$ accuracy on reddit-binary dataset with $L_2$ ground metric.



**(b)** Recall@$m$ accuracy ModelNet10 dataset with $L_2$ ground metric.



**(c)** Recall@$m$ accuracy on synthetic-uniform dataset with $L_2$ ground metric.



**(d)** Recall@$m$ accuracy on synthetic-Gaussian dataset with $L_2$ ground metric.

**Figure 4** Recall@$m$ accuracy on reddit-binary and ModelNet10 datasets with $L_2$ ground metric.

In the future, we are interested in using the $L_1$ embedding described with locality sensitive hashing for sub-linear nearest neighbor search. Additionally, it maybe be possible to use the ideas to compare persistence diagrams under some transformations: e.g, parallel shifting along the diagonal directions (which corresponding to that the input functions generating the persistence diagram is added by a constant term). It will also be interesting to expand this work to perform statistics on the space of persistence diagrams (e.g, computing 1-mean of persistence diagrams under our approximation distances).

## References

1 Arturs Backurs, Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Scalable nearest neighbor search for optimal transport. *arXiv preprint arXiv:1910.04126*, 2019.

2 Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 184–193. IEEE, 1996.

3 Yair Bartal. On approximating arbitrary metrices by tree metrics. *STOC*, 98:161–168, 1998.

4 Dimitri Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research*, 14(1):105–123, 1988.

5 Mickaël Buchet, Yasuaki Hiraoka, and Ippei Obayashi. Persistence homology and material and informatics. In Isao Tanaka, editor, *Nanoinformatics*, pages 75–95. Spring Singapore, Singapore, 2018. doi:10.1007/978-981-10-7617-6_5.

6 Chen Cai and Yusu Wang. Understanding the power of persistence pairing via permutation test. *arXiv preprint arXiv:2001.06058*, 2020.

7 Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thirty-fourth annual ACM symposium on theory of computing*, pages 2292–2300. ACM, 2002.

8 Frédéric Chazal, David Cohen-Steiner, Leonidas J. Guibas, Facundo Mémoli, and Steve Y. Oudot. Gromov-hausdorff stable signatures for shapes using persistence. *Computer Graphics Forum*, 28(5):1393–1403, 2009.

9 David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Stability of persistence diagrams. *Discrete Comput. Geom.*, 37(1):103–120, 2007.

10 Herbert Edelsbrunner and John Harer. *Computational topology: an introduction*. American Mathematical Society, 2010.

11 Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. *Discrete Comput. Gemo.*, 28:511–533, 2002.

12 Rémi Flamary and Nicolas Courty. Pot: Python optimal transport library, 2017. URL: https://pythonot.github.io/.

13 Jennifer Gamble and Giseon Ho. Exploring uses of persistence homology for statistical analysis of landmark-based shape data. *Journal of Multivariate Analysis*, 101(9):2184–2199, 2010.

14 Yasuaki Hiraoka, Takenobu Nakamura, Akihiko Hirata, Emerson G. Escolar, Kaname Matsue, and Yasumasa Nishiura. Hierarchical structures of amorphous solids characterized by persistent homology. *Proceedings of the National Academy of Sciences*, 113(26):7035–7040, 2016. doi:10.1073/pnas.1520877113.

15 Piotr Indyk and Nitin Thaper. Fast image retrieval via embeddings. In *3rd international workshop on statistical and computational theories of vision*, volume 2, page 5, 2003.

16 Bahman Kalantari and Iraj Kalantari. A linear-time algorithm for minimum cost flow on undirected one-trees. *Combinatorics Advances*, pages 217–223, 1995.

17 Lida Kanari, Paweł Dłotko, Martina Scolamiero, Ran Levi, Julian Shillcock, Kathryn Hess, and Henry Markram. A topological representation of branching neuronal morphologies. *Neuroinformatics*, 16(1):3–13, 2018. doi:10.1007/s12021-017-9341-1.

18 Michael Kerber, Dimitriy Morozov, and Arnur Nigmetov. Geometry helps to compare persistence diagrams. *Journal of Experimental Algorithmics(JEA)*, 22:1–20, 2017.

19 Jon Kleinberg and Eva Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and markov random fields. *Journal of the ACM (JACM)*, 49(5):616–639, 2002.

20 Harold Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

21 Théo Lacombe, Marco Cuturi, and Steve Oudot. Large scale computation of means and clusters for persistence diagrams using optimal transport, 2018. arXiv:1805.08331.

22 Tam Le, Makoto Yamada, Kenji Fukumizu, and Marco Cuturi. Tree-sliced approximation of wasserstein distances. *arXiv preprint arXiv:1902.00342*, 2019.

23 Yongjin Lee, Senja D. Barthel, Paweł Dłotko, S. Mohamad Moosavi, Kathryn Hess, and Berend Smit. Quantifying similarity of pore-geometry in nanoporous materials. *Nat Commun.*, 8:15396, 2017. doi:10.1038/ncomms15396.

24 Yanjie Li, Dingkang Wang, Giorgio Ascoli, Partha Mitra, and Yusu Wang. Metrics for comparing neuronal tree shapes based on persistent homology. *PLOS One*, 12(8):1–24, 2017. doi:10.1371/journal.pone.0182184.

25 Clément Maria, Jean-Daniel Boissonat, Marc Glisse, and Mariette Yvinec. The gudhi library: simplicial complexes and persistent homology, 2014. URL: http://gudhi.gforge.inria.fr/python/latest/index.html.

26 Dimitriy Morozov. Dionysus, 2010. URL: `mrzv.org/software/dionysus`.
27 Ahmet Sacan, Ozgur Ozturk, Hakan Ferhatosmanoglu, and Yusu Wang. Lfm-pro: a tool for detecting significant local structural sites in proteins. *Bioinformatics*, 23(6):709–716, 2007.
28 Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Fast unbalanced optimal transport on tree. *arXiv preprint arXiv:2006.02703*, 2020.
29 Primoz Skraba, Maks Ovsjanikov, Frédéric Chazal, and Leonidas Guibas. Persistence-based segmentation of deformable shapes. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 45–52, 2010. `doi:10.1109/CVPRW.2010.5543285`.
30 Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumentric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

## A Additional Proofs

**Proof of observation 4.** By the optimality of $\mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}})$, we know that

$$\mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}}) \leq \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(P, Q) + \sum_{(a,b)\in\Gamma_1} ||\pi(a) - \pi(b)||_q$$

where $\Gamma_1$ is the set of $(a, b) \in \Gamma$ that have first form given in definition 2. We know that $||\pi(a) - \pi(b)||_q \leq ||a - b||_q$ so $\mathrm{d}_{\mathrm{OT}}(\mu_{\widehat{\mathsf{P}}}, \nu_{\widehat{\mathsf{Q}}}) \leq \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(P, Q) + \sum_{(a,b)\in\Gamma_1} ||\pi(a) - \pi(b)||_q \leq 2 \cdot \mathrm{d}^{\mathrm{per}}_{\mathrm{W},1}(P, Q)$. ◀

## B Datasets

We use datasets of synthetic persistence diagrams as well as persistence diagrams generated from real data. For persistence diagrams from real data, we use both graph and shape datasets.

- *Synthetic data*: We generated two sets of persistence diagrams. For the first set of synthetic persistence diagrams, we find a random persistence of size at most $s$ where $s = 10x$ for $x \in \{1, \dots, 100\}$ by generating points $p_1, \dots, p_s$. To find each points $p_i$, we sample $p_i.x$ from a uniform distribution from 0 to 200. We then sample $p_i.y$ from a uniform distribution between $x$ and 300. We will refer to this set of synthetic persistence diagrams as synthetic-uniform. For the second set of synthetic persistence diagrams, we again generate a point $p_i$ by sampling $p_i.x$ from a uniform distribution from 0 to 200. We then sample $p_i.y$ from a Gaussian distribution centered about $x$ with standard deviation 1.0. We will refer to the second set of synthetic diagrams as synthetic-Gaussian.
- *Graphs*: For persistence diagrams generated from graphs, we use the reddit-binary graph dataset, which consists of graphs corresponding to online discussion on Reddit. In each graph, nodes correspond to users and there is an edge between two nodes if at least one of the corresponding users has responded to the other's comments. The data is taken from four popular subreddits: IAmA, AskReddit, TrollXChromosomes, and atheism. Additionally, the persistence diagrams are generated using node degree as the filtration function [6].
- *Shapes*: We use the ModelNet10 [30] dataset to generate persistence diagrams from shapes. ModelNet10 is comprised of 4899 CAD models from 10 object categories. The persistence diagrams are generated using closeness centrality as the filtration function [6].

The statistics of the ModelNet10 and reddit-binary datasets are summarized in Table 3. Note that a persistence point is considered close to the diagonal if its lifetime is less than one-tenth of the lifetime of the point with the largest lifetime.

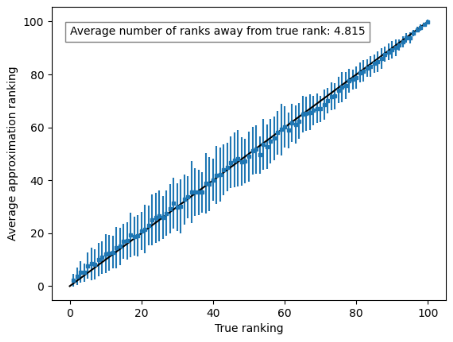
**Table 3** Diagram statistics for reddit-binary and ModelNet10 datasets.

| | Average # of PD points | Average # of points near diagonal |
|---|---|---|
| reddit-binary | 278.155 | 20.245 |
| ModelNet10 | 40.52 | 34.345 |

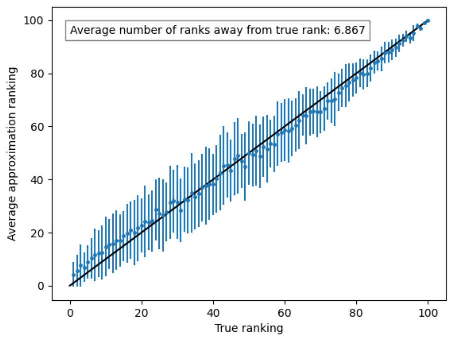
## C Results

To measure the accuracy of the rankings produced by the modified $L_1$ embedding and flowtree methods, we plot the true ranking of each candidate against the rank of the candidate in the rankings produced by the evaluated method. Note that we will do this using the $L_2$ norm as the ground metric. The ranking accuracies for the evaluated methods for the reddit-binary dataset and ModelNet10 are summarized in Figures 5, 6, 8, 7. The average number of ranks away from the true rank is less than 10 for both reddit-binary and synthetic-uniform whereas the same metric for synthetic-uniform and ModelNet10 is above ten for both datasets.

Both the modified flowtree and the $L_1$ embedding approximations seem to be less effective for estimating nearest neighbor for persistence diagrams where there is a high proportion of points near the diagonal. This may be because a higher proportion of points near the diagonal increases the possibility of erroneously matching points to the diagonal.

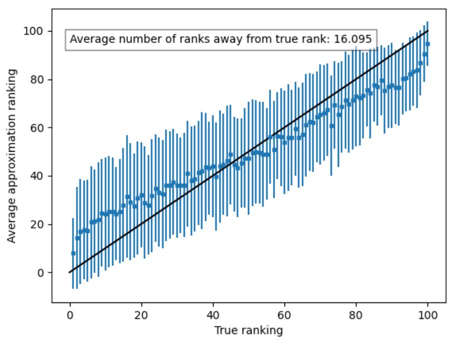


**(a)** Flowtree rankings.

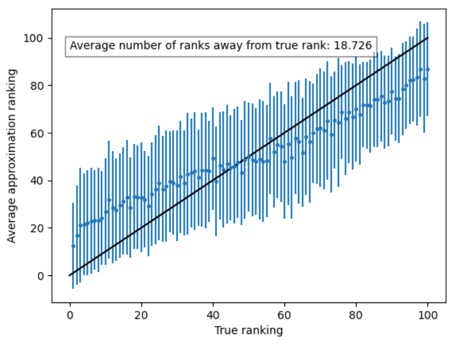


**(b)** $L_1$ embedding rankings.

**Figure 5** Comparison of rankings generated by the flowtree and $L_1$ embedding approximations with the true rankings of the candidate diagrams using the reddit-binary datset.

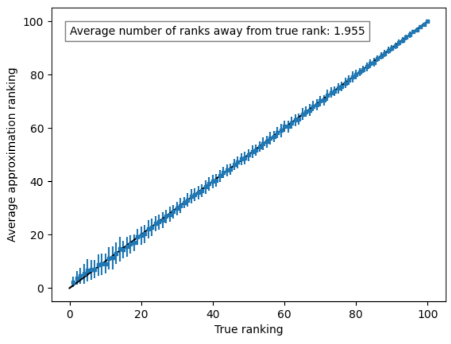


**(a)** Flowtree rankings.

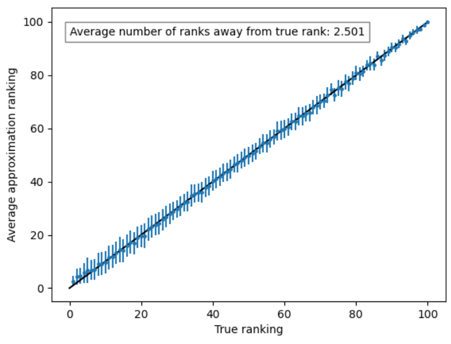


**(b)** $L_1$ embedding rankings.

**Figure 6** Comparison of rankings generated by the flowtree and $L_1$ embedding approximations with the true rankings of the candidate diagrams using the ModelNet10.

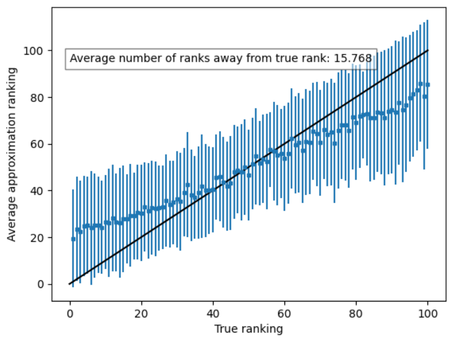


**(a)** Modified flowtree rankings.

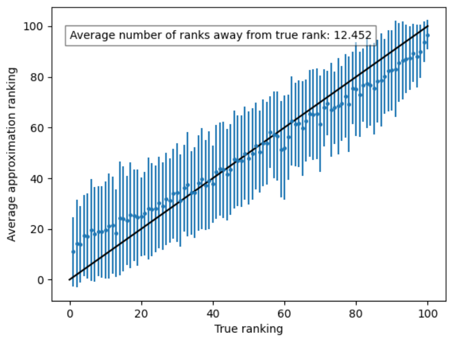


**(b)** $L_1$ embedding rankings.

**Figure 7** Comparison of rankings generated by the modified flowtree and $L_1$ embedding approximations with the true rankings of the candidate diagrams using synthetic-uniform dataset.



**(a)** Modified flowtree rankings.



**(b)** $L_1$ embedding rankings.

**Figure 8** Comparison of rankings generated by the modified flowtree and $L_1$ embedding approximations with the true rankings of the candidate diagrams using synthetic-Gaussian.

# Fréchet Mean and $p$-Mean on the Unit Circle: Decidability, Algorithm, and Applications to Clustering on the Flat Torus

**Frédéric Cazals**
Université Côte d'Azur, France
Inria, Sophia Antipolis, France

**Bernard Delmas**
INRAe, Jouy-en-Josas, France

**Timothee O'Donnell**
Université Côte d'Azur, France
Inria, Sophia Antipolis, France

## Abstract

The center of mass of a point set lying on a manifold generalizes the celebrated Euclidean centroid, and is ubiquitous in statistical analysis in non Euclidean spaces. In this work, we give a complete characterization of the weighted $p$-mean of a finite set of angular values on $S^1$, based on a decomposition of $S^1$ such that the functional of interest has at most one local minimum per cell. This characterization is used to show that the problem is decidable for rational angular values –a consequence of Lindemann's theorem on the transcendence of $\pi$, and to develop an effective algorithm parameterized by exact predicates. A robust implementation of this algorithm based on multi-precision interval arithmetic is also presented, and is shown to be effective for large values of $n$ and $p$. We use it as building block to implement the k-means and k-means++ clustering algorithms on the flat torus, with applications to clustering protein molecular conformations. These algorithms are available in the Structural Bioinformatics Library (`http://sbl.inria.fr`).

Our derivations are of interest in two respects. First, efficient $p$-mean calculations are relevant to develop principal components analysis on the flat torus encoding angular spaces–a particularly important case to describe molecular conformations. Second, our two-stage strategy stresses the interest of combinatorial methods for p-means, also emphasizing the role of numerical issues.

**2012 ACM Subject Classification** Theory of computation → Computational geometry

**Keywords and phrases** Frechét mean, p-mean, circular statistics, decidability, robustness, multi-precision, angular spaces, flat torus, clustering, molecular conformations

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.15

**Supplementary Material** *Software*: `https://sbl.inria.fr/`
*Text (Documentation)*: `https://sbl.inria.fr/doc/Frechet_mean_S1-user-manual.html`

**Acknowledgements** Chee Yap and Sylvain Pion are acknowledged for discussions on irrational number theory and number types, respectively.

## 1 Introduction

### 1.1 Statistics on manifolds and $p$-means on $S^1$

**Fréchet mean and generalizations.** The celebrated center of mass of a point set $P$ in a Euclidean space is the (a) point minimizing the sum of squared Euclidean distances to points in $P$. The center of mass plays a key role in data analysis at large, and in particular in principal components analysis since the data are centered prior to computing the covariance matrix and the principal directions. Generalizing these notions to non Euclidean spaces is an active area of research. Motivated by applications in structural biology (molecular conformations), robotics (robot conformations), and medicine (shape and relative positions of organs), early work focused on direct generalizations of Euclidean notions. Analysis tailored to the unit circle and sphere were developed under the umbrella of directional statistics [2, 19, 20]. In a more abstract setting, generalizations of the center of mass in general metric spaces were first worked out – the so-called Fréchet mean [11], followed by a generalization to distributions on such spaces – the so-called Karcher mean [12, 3, 23].

In fact, previous works span two complementary directions. On the one hand, efforts have focused on mathematical properties of spaces generalizing affine spaces, so as to provide statistical summaries of ensembles in terms of geometric objects of small dimension. On the other hand, algorithmic developments have been performed proposed to compute such objects. The case of the unit circle $S^1$ provides the simplest compact non Euclidean manifold to be analyzed. Despite its simplicity, this case turns out to be of high interest since $S^1$ encodes angles, a particularly important case e.g. to describe molecular conformations. In the following, we focus on $p$-means defined on the unit circle $S^1$, for $p > 1$. (The case $p = 1$ requires trivial adaptations.)

Consider $n$ angles $\Theta_0 = \{\theta_i\}_{i=1,\ldots,n}$. Practically, since real data are known with finite precision, we treat angles as rational numbers. Consider the embedding of an angle onto the unit circle, that is $X(\theta) = (\cos\theta, \sin\theta)^\intercal$. The geodesic distance between two points $X(\theta)$ and $X(\theta_i)$ on $S^1$, denoted $d(\cdot,\cdot)$, satisfies

$$d(X(\theta), X(\theta_i)) = \min(\mid \theta - \theta_i \mid, 2\pi - \mid \theta - \theta_i \mid) = 2 \arcsin \frac{\|X(\theta) - X(\theta_i)\|}{2}. \tag{1}$$

Consider a set of positive weights $\{w_i\}_{i=1,\ldots,n}$. For an integer $p \geq 1$, consider the function involving the weighted distances to all points, i.e.

$$F_p(\theta) = \sum_{i=1,\ldots,n} w_i f_i(\theta), \text{ with } f_i(\theta) = d^p(X(\theta), X(\theta_i)). \tag{2}$$

We denote its minimum

$$\theta^* = \arg \min_{\theta \in [0, 2\pi)} F_p(\theta). \tag{3}$$

For units weights and $p = 2$, the value obtained is the Fréchet mean. In that case, the candidate minimizers (local minima of Eq. 2) form the vertices of a regular polygon [13]. The previous expression can also be seen as a distance to a point mass probability distribution on $S^1$. For a general probability distribution on $S^1$, necessary and sufficient conditions for the existence of a Fréchet mean have been worked out [9]. In the same paper, the authors propose a quadratic algorithm–regardless of numerical issues–to compute the Fréchet mean for the particular case of a point mass probability distribution. In a more general setting, a stochastic algorithm finding $p$-means wrt a general measure on the circle has also been proposed [4].

▶ Remark 1. In the subsequent sections, the weights in Eq. 2 are omitted – rational weights do not change our analysis. Our implementation, however, does use them.

**Robustness and numerical issues.** From a mathematical standpoint, computing the $p$-mean is a non-convex optimization problem, and one may assume that calculations are carried out in the standard real RAM computer model, which assumes that exact operations on real numbers are available at constant time per operation [24]. From a practical standpoint though, numbers in real computers are represented with finite precision [21]. The ensuing rounding errors are such that algorithms written in the real RAM model may loop, crash, or terminate with an erroneous answer, even for the simplest 2D geometric calculations [16].

Robust geometric algorithms, which deliver what they are designed for, can be developed using the Exact Geometric Computation (EGC) paradigm [27], which is central in the Computational Geometry Algorithms Library (CGAL) [1]. The EGC relies on so-called *exact predicates* and *constructions*. A predicate is a function whose output belongs to a finite set, while a construction exhibits a new geometric object from the input data. For example, the predicate Sign$(x)$ returns the sign $\{negative, null, positive\}$ of the arithmetic expression $x$. As we shall see, designing robust predicates for $p$-means on $S^1$ is connected to transcendental number theory since expressions involving $\pi$ are dealt with. In particular, one needs to evaluate the sign of such expressions, which raises decidability issues [8].

**Combinatorial complexity issues.** The computation of the $p$-means also raises a combinatorial complexity issue. Function $F_p$ being a sum over $n$ terms, $k$ function evaluations yield a complexity $O(kn)$, which is quadratic if there is a linear number of local minima. Therefore, the fact that using candidate minimizers form a regular polygon [13] does not directly yield a linear time algorithm even if the angles are sorted. As we shall see, the piecewise maintenance of the expression of the function does so, though. For the sake of conciseness, combinatorial complexity is plainly referred to as complexity in the sequel.

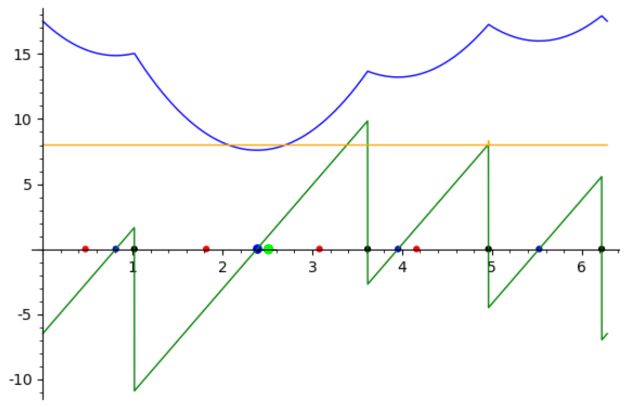


**Figure 1 Fréchet mean of four points on $S^1$. (Functions)** blue: function $F_2$; green: derivative $F_2^{'}$; orange: second derivative $F_2^{''}$ **(Points)** red bullets: data points; black bullets: antipodal points; blue bullets: local minima of the function; large blue bullet: Fréchet mean $\theta^*$; green bullet: circular mean Eq. 14.

## 1.2 Contributions

This paper makes three contributions regarding $p$-means of a finite point set. First, we show that the function $F_p$ is determined by a very simple combinatorial structure, namely a partition of $S^1$ into circle arcs. Second, we give an explicit expression for $F_p$, deduce that the problem is decidable, and present an algorithm computing $p$-means. Third, we present an effective and robust implementation, based on multi-precision interval arithmetic.

## 2 $p$-mean of a finite point set on $S^1$: characterization

### 2.1 Notations

In the following, angles are in $[0, 2\pi)$. We first define:

▶ **Definition 2.** *For each angle $\theta_i \in [0, \pi)$, we define $\theta_i^+ = \theta_i + \pi$. The set of all such angles is denoted $\Theta^+ = \{\theta_i^+\}$. For each angle $\theta_i \in [\pi, 2\pi)$, we define $\theta_i^- = \theta_i - \pi$. The set of all such angles is denoted $\Theta^- = \{\theta_i^-\}$. The* antipodal set *of $\Theta_0$ is the set of angles $\Theta^\pm = \Theta^+ \cup \Theta^-$.*

Altogether, these angles yield the larger set

$$\Theta = \Theta_0 \cup \Theta^\pm. \tag{4}$$

The $2n$ angles in $\Theta$ are generically denoted $\alpha_i$ or $\alpha_j$. Note however that when referring to an angle in the continuous interval $[0, 2\pi)$, $\theta$ is used.

To each angle $\theta_i$, we associate three so-called *elementary intervals* (Fig. 2):

- $\theta_i \in [0, \pi) : I_{i,1} = (0, \theta_i), I_{i,2} = (\theta_i, \theta_i^+), I_{i,3} = (\theta_i^+, 2\pi)$.
- $\theta_i \in [\pi, 2\pi) : I_{i,1} = (0, \theta_i^-), I_{i,2} = (\theta_i^-, \theta_i), I_{i,3} = (\theta_i, 2\pi)$.

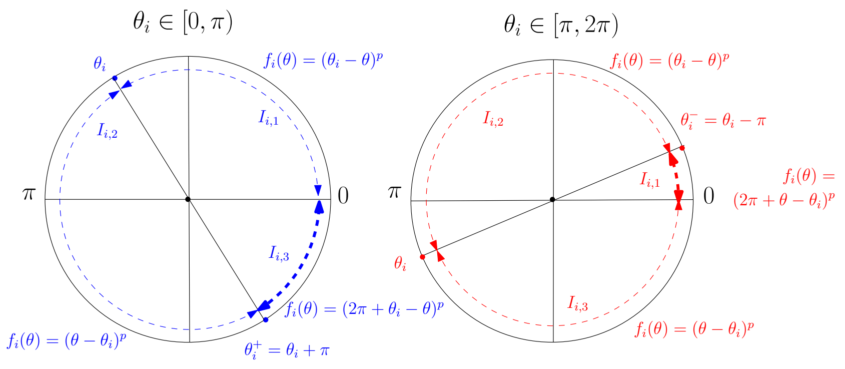

$\theta_i \in [0, \pi)$
$\theta_i$
$f_i(\theta) = (\theta_i - \theta)^p$
$I_{i,1}$
$I_{i,2}$
$\pi$
$0$
$I_{i,3}$
$f_i(\theta) = (2\pi + \theta_i - \theta)^p$
$f_i(\theta) = (\theta - \theta_i)^p$
$\theta_i^+ = \theta_i + \pi$
$\theta_i \in [\pi, 2\pi)$
$f_i(\theta) = (\theta_i - \theta)^p$
$I_{i,2}$
$\theta_i^- = \theta_i - \pi$
$f_i(\theta) =$
$(2\pi + \theta - \theta_i)^p$
$I_{i,1}$
$\pi$
$0$
$\theta_i$
$I_{i,3}$
$f_i(\theta) = (\theta - \theta_i)^p$

**Figure 2 The partition of $S^1$ into circle arcs, and the piecewise functions defining $F_p$.** The three elementary intervals defined by angles in $[0, \pi)$ and $[\pi, 2\pi)$ respectively. Bold circle arcs indicate that $f_i$ has a transcendental expression i.e. involves $\pi$.

## 2.2 Partition of $S^1$

We also consider the partition of $[0, 2\pi)$ induced by the intersection of the $3n$ intervals $\{I_{i,1}, I_{i,2}, I_{i,3}\}$ (Fig. 2). More specifically, we choose one interval (out of three) for each function $f_i$, and intersect them all:

▶ **Definition 3.** *The elementary intervals $I_{i,j}$ define a partition of $S^1$ based on the following intervals:*

$$\mathcal{I} = \{ \bigcap_{i=1,\ldots,n} (I_{i,1} \vee I_{i,2} \vee I_{i,3}) \text{ with } \bigcap_{i=1,\ldots,n} I_{i,\cdot} \neq \emptyset \}. \tag{5}$$

*In the following, open intervals from $\mathcal{I}$ are denoted $(\alpha_j, \alpha_{j+1})$.*

▶ Remark 4. From the previous definition, it appears that the intervals in $\mathcal{I}$ may be ascribed to nine types since the left endpoint is an angle $\theta_i$ or an antipodal angle $\theta_i^+$ or $\theta_i^-$, and likewise for the right endpoint.

## 2.3 Piecewise expression for $F_p$

We use the previous intervals to describe the piecewise structure of $F_p$. We define the following piecewise functions (Fig 2):

$$\theta_i \in [0,\pi) : f_i(\theta) = \begin{cases} (\theta_i - \theta)^p, \text{ for } \theta \in I_{i,1}, \\ (\theta - \theta_i)^p, \text{ for } \theta \in I_{i,2}, \\ (2\pi + \theta_i - \theta)^p, \text{ for } \theta \in I_{i,3}. \end{cases} \tag{6}$$

$$\theta_i \in [\pi, 2\pi) : f_i(\theta) = \begin{cases} (2\pi + \theta - \theta_i)^p, \text{ for } \theta \in I_{i,1}, \\ (\theta_i - \theta)^p, \text{ for } \theta \in I_{i,2}, \\ (\theta - \theta_i)^p, \text{ for } \theta \in I_{i,3}. \end{cases} \tag{7}$$

The previous equations give the piecewise expression of $F_p(\theta)$ (Eq. 2), from which one derives the following, which characterizes the derivative at points in $\alpha_j \in \Theta$:

$$\Delta f'_{i|\theta} = \lim_{\theta \searrow \alpha_j} f'_i(\theta) - \lim_{\theta \nearrow \alpha_j} f'_i(\theta) \tag{8}$$

▶ Remark 5. Let $\theta_{\max}$ be the antipodal value of the largest $\theta_i \in \Theta_0$ larger than $\pi$, and $\theta_{\min}$ the antipode of the smallest $\theta_i \in \Theta_0$ smaller than $\pi$. The function $F_p$ is transcendental in $[0, \theta_{\max})$ and $(\theta_{\min}, 2\pi]$ – its expression involves $\pi$. Also, the function $F_p$ is algebraic on $(\theta_{\max}, \theta_{\min})$. See Fig. 2.

Using Eq. 8, the following is immediate:

▶ **Lemma 6.** *For $p > 1$, the function $f_i$ and its derivatives satisfy:*

- *The function $f_i$ is continuous on $S^1$.*
- *The derivative $f'_i$ is continuous on $S^1$ except at the antipodal value of $\theta_i$, where $\Delta f'_{i|antipode(\theta_i)} = -2p\ \pi^{p-1}$.*
- *The second order derivative $f''_i$ is non negative on $S^1$.*

The previous lemma tells us that $F'_p$ incurs drops at antipodal points, and then keeps increasing again on the interval starting at that point. Finding local minima of $F_p$ therefore requires finding those intervals from $\mathcal{I}$ where $F'_p$ vanishes, which happens at most once:

▶ **Lemma 7.** *For $p > 1$, the function $F_p$ has at most one local min. on each interval in $\mathcal{I}$.*

## 3 Algorithm

The observations above are not sufficient to obtain an efficient algorithm: since there are $2n$ intervals and since the function has linear complexity on each of them, a linear number of function evaluations has quadratic complexity. We get around this difficulty by maintaining the expression of the function at angles in $\Theta$.

### 3.1 Analytical expressions and nullity of $F'_p$

**The function $F_p$ and its derivative.** We first derive a compact, analytical expression of $F_p$ and $F'_p$. Following Eqs. 6 and 7, the expressions of $f_i(\theta)$ and $f'_i(\theta)$ can be written as

$$f'_i(\theta) = k_i \times (a_i + \varepsilon_i \theta)^{p-1}, \text{ with } k_i \in \{-p, p\}, a_i \in \{-\theta_i, 2\pi - \theta_i, \theta_i, 2\pi + \theta_i\}, \varepsilon_i \in \{-1, +1\}. \tag{9}$$

On open intervals $(\alpha_j, \alpha_{j+1})$, the function reads as the following polynomial

$$F_p(\theta) = \sum_{i=1}^{n} (a_i + \varepsilon_i \theta)^p = \sum_{j=0}^{p} b_j \theta^j, \text{ with } b_j = \sum_{i=1}^{n} \binom{p}{j} a_i^{p-j} \varepsilon_i^j. \tag{10}$$

Similarly, the derivative $F_p^{'}(\theta)$ reads as a degree $p-1$ polynomial:

$$F_p^{'}(\theta) = \sum_{i=1}^{n} k_i (a_i + \varepsilon_i \theta)^{p-1} = \sum_{j=0}^{p-1} c_j \theta^j, \text{ with } c_j = \sum_{i=1}^{n} k_i \binom{p-1}{j} a_i^{p-1-j} \varepsilon_i^j. \tag{11}$$

In the following, we assume that the coefficients of $F_p$ and $F_p^{'}$ are stored in two vectors $B$ and $C$ of size $p+1$ and $p$ respectively, so that evaluating the function or its derivative at a given $\theta$ has cost $O(p)$.

**Nullity of $F_p^{'}$: algebraic versus transcendental expressions.** The previous equations call for two important comments. First, from the combinatorial complexity standpoint, if the coefficients of the polynomials are known, evaluating $F_p$ and $F_p^{'}$ has cost $O(p)$. Second, from the numerical standpoint, locating local minima of $F_p$ requires finding intervals from $\mathcal{I}$ on which $F_p^{'}$ vanishes. Identifying such intervals is key to the robustness of our algorithm. Practically, since an interval is defined by two consecutive values in the set $\Theta$, we need to check that the sign of $F_p^{'}$ differs at these endpoints. The cornerstone is therefore to decide the sign of $F_p^{'}$ at angles in $\Theta$ (input angles or their antipodes), and the following is a simple consequence of Lindemann's theorem on the transcendence of $\pi$:

▶ **Lemma 8.** *If the angular values $\theta_i \in \Theta_0$ are rational numbers, checking whether $F_p^{'}(\alpha_i) \neq 0$ for any $\alpha_i \in \Theta$ is decidable. Moreover, when $F_p^{'}$ has a transcendental expression and $\alpha_i$ is rational, $F_p^{'} \neq 0$.*

**Proof.** We first consider the case $\alpha_i \in \Theta_0$, and distinguish the two types of intervals – see Remark 5. First, consider an interval where $F_p$ has an algebraic expression. We face a purely algebraic problem, and deciding whether $F_p^{'}(\alpha_i) \neq 0$ can be done using classical bounds, e.g. Mahler bounds [18, 28]. Second, consider an interval where $F_p$ has a transcendental expression. Then, $F_p^{'}(\alpha_i)$ can be rewritten as a polynomial of degree $p-1$ in $\pi$. Lindemann's theorem on the transcendence of $\pi$ implies that $F_p^{'}(\alpha_i) \neq 0$.

Consider now the case where $\alpha_i \in \Theta^{\pm}$, that is $\alpha_i = \alpha_j \pm \pi$. Each individual term $f_i{'}(\alpha_i)$ also has the form $(c_i\pi + q_i)^{p-1}$, with $c_i \in \mathbb{N}$ and $q_i \in \mathbb{Q}$, so that the latter case also applies. ◀

## 3.2 Algorithm

Upon creating and sorting the set $\Theta$, which has complexity $O(n \log n)$, the algorithm involves four steps for each interval in $\mathcal{I}$.

**Identify the intervals where $F_p^{'}$ vanishes.** By lemmas 6 and 7, there is at most one local minimum per interval, which requires checking the signs of $F_p^{'}$ to the right and left bounds of an interval $(\alpha_j, \alpha_{j+1})$. Using the functional forms encoded in vector $C$, computing these derivatives has the same complexity as the previous step. However, this step calls for two important comments:

- For $\alpha_i \in \Theta$, checking whether $F_p^{'}(\alpha_i) \neq 0$ is decidable – Lemma 8. However, the arithmetic nature of the number $\alpha_i$ must be taken into account, as rational numbers (input angles) and transcendental numbers (antipodal points) must be dealt with using different arithmetic techniques. See below.
- Not all intervals $(\alpha_j, \alpha_{j+1})$ can provide a root. Indeed, once $F_p^{'}(\alpha_i) > 0$, since the individual second order derivatives are positive, $F_p^{'}$ cannot vanish until one crosses one $\alpha_j \in \Theta^{\pm}$. As we shall see, this observation is easily accommodated in Algorithm 1.

In the following, we denote $\mathrm{SD}(p-1)$ the cost of deciding the sign (negative, zero, positive) of $F_p^{'}(\theta)$, for $\theta \in \Theta$.

**Compute the unique root of $F_p^{'}$.** Since $F_p^{'}$ is piecewise polynomial, finding its real root has constant time complexity for $p \leq 5$. Otherwise, a numerical method can be used [17]. In the following, we denote $\mathrm{RF}(p-1)$ the cost of isolating the real root of a degree $p-1$ polynomial.

**Evaluate $F_p$ at a local minimum.** Once the angle $\theta_m$ corresponding to a local minimum has been computed, we evaluate $F_p(\theta_m)$ using Eq. 10. This evaluation has $O(p)$ complexity since the coefficients of the polynomial are known.

**Maintain the polynomials $F_p$ and $F_p^{'}$.** Following Eqs. 10 and 11, the function and its derivative only change when crossing an angle from $\Theta$. At such an angle, updating the vectors $B$ and $C$ has complexity $O(p)$. Overall, this step therefore has complexity $O(np)$.

We summarize with the following output-sensitive complexity:

▶ **Theorem 9.** *Algorithm 1 computes the p-mean with $O(n \log n + np + nSD(p-1) + kRF(p-1) + kp)$ complexity, with $k$ the number of local minima of $F_p$.*

## 3.3 Generic implementation

In the following, we present an implementation of our algorithm based on predicates, i.e. functions deciding branching points.

**Pseudo-code, predicates and constructions.** Our algorithm (Algo. 1) takes as input a list of angular values (in degrees or radians) and the value of $p$. Following Remark 1, an optional file containing the weights may be passed. If $p > 5$, we take for granted an algorithm computing the root of $F_p^{'}$ on an interval. As a default, we resort to a bisection method which divides the interval into two, checks which side contains the unique root of $F_p^{'}$, and iterates until the width of the interval is less than some user specified value $\tau$ (supporting information (SI) Algo. 3). The interval returned is called the *root isolation interval.* Our algorithm was implemented in generic C++ in the Structural Bioinformatics Library [6], as a template class whose main parameter is a geometric kernel providing the required predicates and constructions. We now discuss these–see Sec. 3.4 for their robust implementation.

**Predicates.** The algorithm involves two predicates:
- `Sign`$(F_p^{'}(\theta))$. Predicate used to determine the sign of the $F_p^{'}(\theta)$ with $\theta \in [0, 2\pi)$ (SI Algo. 3).
- `Interval_too_wide`$(\theta_l, \theta_r)$. Predicate used to determine whether the root isolation interval has width less than $\tau$ (SI Algo. 3). It is true if $\theta_r - \theta_l > \tau$, and false otherwise.

**Constructions.**

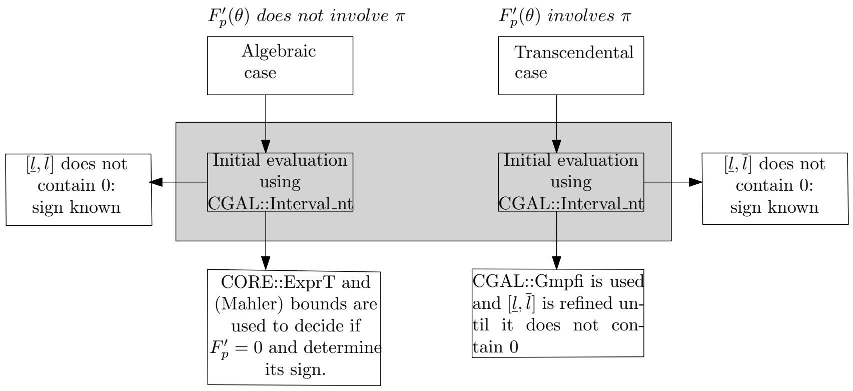
- **Updating representations.**. Updating the coefficients in $B$ and $C$ is necessary at each $\alpha_i \in \Theta$: for $F_p(\theta)$ (resp. $F_p^{'}(\theta)$), we subtract the contribution of $f_i(\theta)$ (resp. $f_i^{'}(\theta)$) before $\alpha_i$, and add that of $f_i(\theta)$ (resp. $f_i^{'}(\theta)$) after $\alpha_i$.
- **Find_root**. To computing the root of $F_p^{'}$ on an interval $(\alpha_j, \alpha_{j+1})$, we resort to a bisection method $p > 3$ (SI Algo. 3), with radical based formulae otherwise.

▶ Remark 10. A kernel based on floating point number types, the `double` type in our case, is easily assembled, see `SBL::GT::Inexact_predicates_kernel_for_frechet_mean` in SI Sec. 3.5. As noticed earlier, it comes with no guarantee. In particular, the algorithm may terminate with an erroneous result if selected predicates are falsely evaluated.

## 3.4 Robust implementation based on exact predicates

**Number types for lazy evaluations.** Following the Exact Geometric Computation exact predicates are gathered in a *kernel*. We circumvent rounding errors using interval number types which are certified to contain the exact value of interest. That is, an expression $x$ is represented by the interval $[\underline{x}, \overline{x}] \ni x$. The bounds of these intervals may have a fixed precision, which corresponds to the CGAL::Interval_nt number type [1]. Or the bounds may be multiprecision, e.g. `Gmpfr` from `Mpfr` [10], which corresponds to the CGAL::Gmpfi type [1]. We now explain how these types are used to code exact predicates.

**The `Sign` predicate.** We distinguish the algebraic and transcendental cases, performing multiprecision calculations only if needed (Fig. 3).



$F_p'(\theta)$ does not involve $\pi$
$F_p'(\theta)$ involves $\pi$
Algebraic case
Transcendental case
$[\underline{l}, l]$ does not contain 0: sign known
Initial evaluation using CGAL::Interval_nt
Initial evaluation using CGAL::Interval_nt
$[\underline{l}, \bar{l}]$ does not contain 0: sign known
CORE::ExprT and (Mahler) bounds are used to decide if $F_p' = 0$ and determine its sign.
CGAL::Gmpfi is used and $[\underline{l}, \bar{l}]$ is refined until it does not contain 0

**Figure 3 Number types used in the `Sign` predicate.** Note that CGAL::Interval_nt is used in the algebraic and transcendental cases, while the remaining number types are only used if required.

• **Transcendental case: multiprecision interval arithmetic.** When $F_p$ is transcendental and $\alpha_i$ rational, $F_p^{'}(\alpha_i)$ is positive or negative (lemma 8). Another case where $F_p^{'}(\alpha_i) \neq 0$ is when $\alpha_i \in \Theta^{\pm}$. In our implementation this situation is faced in two cases. First, in the main algorithm (Algo. 1), `Sign`($l$) or `Sign`($r$): $l$ and $r$ are transcendental if $\alpha_i \in \Theta^{\pm}$. Second, in the root finding algorithm(SI Algo. 3), `Sign`($F_p^{'}(c)$): $c$ is transcendental if $\alpha_{i-1}$ or $\alpha_i \in \Theta^{\pm}$. In both cases, we proceed in a lazy way: first, we try to conclude using CGAL::Interval_nt; if this interval contains zero, we switch to CGAL::Gmpfi (Fig. 3), refine the interval bounds, and conclude. Refining the interval consists of iteratively doubling the number of bits used to describe all numbers–including $\pi$, until a conclusion can be reached.

**Algorithm 1** $p$-mean calculation: generic algorithm for $p > 1$ in the real RAM model.

```
 1: $\Theta$: $vector[1, 2n]$ containing all the angles
 2: B: $vector[1, p+1]$ to store the coefficients of the polynomial $F_p(\theta)$ Eq. 10
 3: C: $vector[1, p]$ to store the coefficients of the polynomial $F_p^{'}(\theta)$ Eq. 11
 4: $\theta^*$ // Angle corresponding to the global minimum of $F_p$
 5: Root_remains = true // flag indicating whether a root must be sought on $(\alpha_j, \alpha_{j+1})$
 6:
 7: // Initialization
 8: Compute $\Theta^{\pm}$ and form sorted $\Theta$
 9: $\alpha_0$: first angle in $\Theta$
10: Store the coefficients of $F_p$ into the vector $B$ for the interval $(0, \alpha_0)$
11: Store the coefficients of $F_p^{'}$ into vector $C$ for the interval $(0, \alpha_0)$
12: Compute $l \leftarrow F_p^{'}(\theta)$ for $\theta \to 0^+$ using Eq. 11 and vector $C$
13: **Update_root**(Sign($l$))//Updates Root_remains see SI Algo. 2
14: **if** Sign($l$) is null **then**
15:     Compute $F_p(0)$ using vector B and Eq. 10, and possibly update $\theta^*$.
16:
17: // For each angle, handle {interval ending, coefficients in B and C, interval starting}
18: **for all** $\alpha_i$ **in** $\Theta$ **do**
19:     **if** Root_remains **then**
20:         Compute $r \leftarrow F_p^{'}(\theta)$ for $\theta \to \alpha_i^-$ using Eq. 11 and vector $C$
21:         **Update_root**(Sign($r$))//Updates Root_remains see Algo. SI 2
22:         **if** Sign($r$) is positive **then**
23:             $\theta_c \leftarrow$ **Find_root**$(\alpha_{i-1}, \alpha_i)$
24:             Compute $F_p(\theta_c)$ using vector B and Eq. 10, and possibly update $\theta^*$.
25:         **else if** Sign($r$) is null **then**
26:             Compute $F_p(\alpha_i)$ using vector B and Eq. 10, and possibly update $\theta^*$.
27:     Update the coefficients of $F_p$ stored in vector B upon crossing $\alpha_i$
28:     Update the coefficients of $F_p^{'}$ stored in vector C upon crossing $\alpha_i$
29:     **if** $\alpha_i \in \Theta^{\pm}$ **then**
30:         Compute $l \leftarrow F_p^{'}(\theta)$ for $\theta \to \alpha_i^+$ using Eq. 11 and vector $C$
31:         **Update_root**(Sign($l$))//Updates Root_remains see SI Algo. 2
32:         **if** Sign($l$) is null **then**
33:             Compute $F_p(\alpha_i)$ using vector B and Eq. 10, and possibly update $\theta^*$.
34:
35: // Process the interval ending at $2\pi$
36: Compute $r \leftarrow F_p^{'}(\theta)$ for $\theta \to 2\pi^-$ using Eq. 11 and vector $C$
37: **if** Root_remains **then**
38:     **if** Sign($r$) is positive **then**
39:         $\theta_c \leftarrow$ **Find_root**$(\theta_{2n}, 2\pi)$
40:         Compute $F_p(\theta_c)$ using vector B and Eq. 10, and possibly update $\theta^*$
41:     **else if** Sign($r$) is null **then**
42:         Compute $F_p(2\pi)$ using vector B and Eq. 10, and possibly update $\theta^*$.
```

• **Algebraic case: zero separation bounds.** When $F_p$ has a rational expression and $\alpha_i$ is rational, Sign($F_p^{'}(\alpha_i)$) may be zero (SI Fig. 7). In this case, an input angle may also correspond to a local minimum of $F_p$. To decide whether $F_p^{'}(\alpha_i) = 0$, we resort to zero separation bounds and multiprecision interval arithmetic.

Let us consider $F_p^{'}(\alpha_i)$ as an arithmetic expression $E$, using a number of authorized operations($\pm, \times, /$ in our case). A separation bound is a function *sep* such that the value $\xi$ of expression $E$ is lower bounded by $sep(E)$ in the following manner:

$$If\ \xi \neq 0 \text{ then } \mathrm{sep}(E) \leq |\xi| \tag{12}$$

Considering $\tilde{\xi}$ an approximation of $\xi$ and $\Delta$ an upper bounded error $|\tilde{\xi} - \xi|$.

$$If\ |\tilde{\xi}| + \Delta < \mathrm{sep}(E) \text{ then } \xi = 0. \tag{13}$$

Practically, we proceed in a lazy way, in two steps (Fig. 3). First, using CGAL::Interval_nt with double precision, we check whether we can conclude on $F_p^{'}(\alpha_i) \neq 0$. If not–the interval contains zero, we use CORE::ExprT[15] to determine the zero separation bound and decide if $F_p^{'}(\alpha_i) = 0$. If not, we finally determine the sign.

**Predicate** Interval_too_wide($\theta_l, \theta_r$). Returns true when $\underline{\theta_r} - \overline{\theta_l} > \tau$, false if $\overline{\theta_r} - \underline{\theta_l} \leq \tau$. Similarly to the sign predicate, we distinguish the transcendental and algebraic cases to check whether $\theta_l - \theta_r - \tau = 0$. Supposing $\tau$ and $\Theta_0$ are rational $\theta_l - \theta_r - \tau$ is transcendental if the initial $\alpha_{i-1}$ or $\alpha_i \in \Theta^{\pm}$. If transcendental the interval is refined in the same way as the transcendental case of the Sign predicate. Otherwise the expression is algebraic and the precision is raised until an exact computation can be performed.

## 3.5 Software availability

The source code is available in the package *Frechet mean for* $S^1$ of the Structural Bioinformatics Library (SBL), a library proposing state-of-the art methods in computational structural biology [6], see `https://sbl.inria.fr/doc/Frechet_mean_S1-user-manual.html` and `https://sbl.inria.fr/`.

For end-users, the package provides executables corresponding to the robust and non-robust implementations. Given a list of angles and the value of $p$, the program returns sorted list of pairs (angular value of local minimum, function value) by increasing value of $F_p$. A Jupyter notebook `Frechet_mean_S1.ipynb` using `SAGE` (`https://www.sagemath.org/`) is also provided.

For developers, The C++ code of our algorithm is provided in the class `SBL::GT::Frechet_mean_S1`, which is templated by the kernel. Two kernels are provided, namely (i) Non-robust kernel: `SBL::GT::Inexact_predicates_kernel_for_frechet_mean`. A plain floating point(double) number type is used, and (ii) Robust kernel: `SBL::GT::Lazy_exact_predicates_kernel _for_frechet_mean`. See Sec. 3.4.

## 4 Experiments

## 4.1 Overview

Our experiments target three aspects, namely (i) robustness, (ii), comparison of the Fréchet mean against the classical circular mean, and (iii) computational complexity. Practically, three sets of angles are used. (Dataset 1) Randomly generated angles. (Dataset 2) So-called

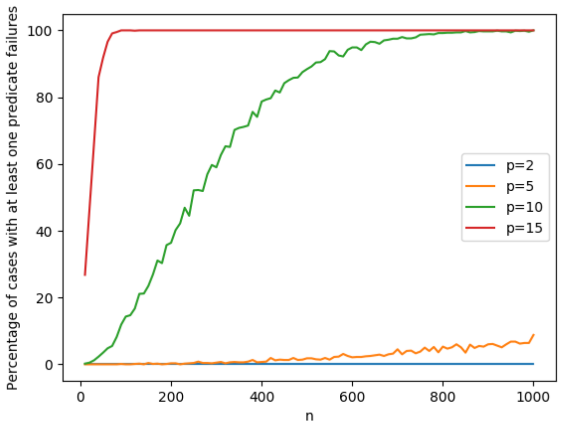


**Figure 4 Fraction of program runs for which at least one predicate execution triggers refinement, as a function of $n$ and $p$.** The number of repeats for each value of $n$ is 1000.

dihedral angles $\chi_i$ in proteins, defined by 4 consecutive atoms on the side chains of amino acids. (Recall that a protein is a polymer of amino acids, and that the 20 natural a.a. differ by their so-called side chains. See Fig. 6 for an example.) These angles are known to be dependent, and correlations between them are key to reduce the dimensionality of the conformation space of proteins [26]. Using the Protein Data Bank, we retained 27093 PDB files with a resolution of 3 angstroms or better. For all polypeptide chains in these files, we computed all dihedral angles of all standard (20) amino-acids. This results in 240 classes of dihedral angles, containing from 50,227 to 439,793 observations. (Dataset 3) Also protein dihedral angles, but from a so-called *rotamer* library [25]. Rotamers (rotational isomers) are preferred conformations adopted by side chains, used to characterize protein conformations.

Note that in all cases, angles being given with finite precision (they are derived from experimentally determined atomic coordinates), they are treated as rational numbers.

## 4.2 Robustness

Using our robust interval-based implementation, we count the fraction of cases for which at least one predicate triggers refinement during an execution. We use sets of $n \in [10, 1000]$ angles generated uniformly at random in $[0, 2\pi)$, and perform 1000 repeats for each value of $n$ (SI Fig. 4). For large values of $p$, whenever $n > 1000$, all executions require interval refinement. Even for $p = 2$ and $n = 10^5$, refinement is triggered in 1.3% of the cases. In all the cases where refinement was triggered, doubling the precision was sufficient to solve the predicate.

## 4.3 Fréchet mean

**Fréchet mean versus circular mean.** A classical way to estimate the circular mean of a set of angles is the *resultant* or *circular mean*, defined as follows [20]:

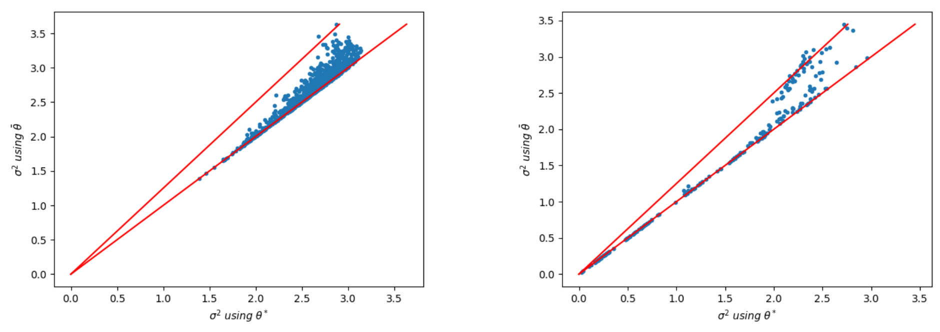
$$\overline{\theta} = \text{atan2}(\sum_i \sin\theta_i/n, \sum_i \cos\theta_i/n). \tag{14}$$

The circular mean does not minimize $F_p$, but minimizes instead [14, Section 1.3]:

$$\overline{\theta} = \arg\min \sum_{i=1,\ldots,n} d(\theta_i, \theta), \text{ with } d(\alpha, \beta) = 1 - \cos(\alpha - \beta). \tag{15}$$

Given a set of angles, we compare the variance of these angles with respect to the Fréchet mean $\theta^*$ and the circular mean $\overline{\theta}$, respectively. Two datasets were used for such experiments: first, randomly generated sets of $n = 30$ angles uniformly at random in $[0, 2\pi)$, with 1000 repeats; second, the aforementioned dihedral angles in protein structures.

For both types of data, the variance obtained for $\overline{\theta}$ is significantly larger than that obtained for $\theta^*$, typically up to 25% (Fig. 5). This shows the interest of using $\theta^*$ in data analysis in general, and to center angles prior to principal components analysis in particular.



Simulated data Torsion angles from protein structural data

**Figure 5 Variance of angles with respect to the Fréchet mean $\theta^*$ and the circular average $\overline{\theta}$. (Left)** Comparison using a simulated set with $n = 30$ angles at random in $[0, 2\pi)$, with 1000 repeats. **(Right)** Comparison for the 243 classes dihedral angles in protein structures–see text. **(Both panels)** In red $y = x$ and $y = 5/4x$.

## 4.4 Computation time and complexity

The complexity of Algorithm 1 (Theorem. 9) has three main components: the sorting step, the updates of vectors $B$ and $C$, and the numerics. We wish in particular to determine whether the $n \log n$ sorting term dominates.

For $p \in \{2, 5, 10, 15\}$, we use sets of $n \in [10^3, 10^5]$ angles generated uniformly at random in $[0, 2\pi)$, and perform 5 repeats for each value of $n$. For $p = 2$, the number of angles is pushed up to $n = 10^7$, with the same number of repeats. In any case, a linear complexity is practically observed (SI Fig. 8) showing that for the values of $n$ used, the constants associated with the linear time update of the data structures and the numerics take over the $n \log n$ term of the sorting step.

## 4.5 Application to clustering on the flat torus

Rotamers characterize the geometry of protein side chains (Sec. 4.1). State of the art rotameric libraries treat the dihedral angles independently [25]. For the a.a. lysine (LYS), (Fig. 6(Inset)), four angles and 3 canonical values for each yield $3^4 = 81$ rotamers.

We undertake the problem of clustering side chains conformations using k-means++ [5]. While k-means is a classical clustering method, the problem solved is non convex and inferring the *right* number of clusters is always problematic [7]. One way to mitigate this difficulty consists of tracking an elbow in the plot of the k-means functional [22]. Using the lysine (LYS) a.a. as example, we work directly on the 4D flat torus $(S^1)^4$, and center the data within a cluster using our Fréchet algorithm. Varying the value of $k$ shows a sharp decline of the k-means++ criterion circa $k = 40$, and then a gradual straightening of the average squared distance (Fig. 6). Working directly on the flat torus therefore makes it possible to capture correlations between individual dihedral angles. The application to a significant reduction (factor of two or so) of rotamers will be reported elsewhere.

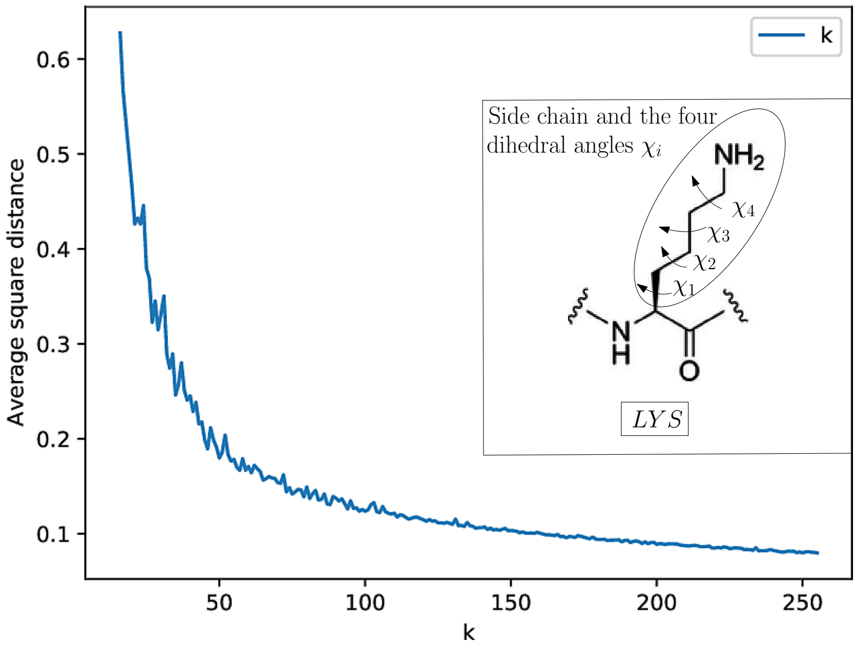


**Figure 6 k-means++ using Fréchet mean as center performed on 4-dimensional flat torus coding the conformational space of the side chain of the Lysine amino acid.** $x$-axis: number of clusters $k$. $y$-axis: average squared distance to the closest cluster center.

## 5 Outlook

The Fréchet mean and the $p$-mean are of central importance as zero dimensional statistical summaries of data which do not live in Euclidean spaces. For the particular case of $S^1$, this paper develops the first robust algorithm computing the $p$-mean. Our algorithm is effective for large number of angular values and large values of $p$ as well, yet, robustness requires predicates and constructions using interval multiprecision arithmetic. For the particular case of the Fréchet mean ($p = 2$), we show that the circular mean should not be used for a substitute to the circular center of mass, as it results in a significantly larger variance.

We foresee two main developments. Application-wise, our results on protein side chain conformations hint at a significant reduction (factor of two or so) of rotamers, which should prove instrumental to foster the diversity of conformational explorations. Also, our centering procedure will help generalizing principal components analysis (PCA) on the flat torus. In theoretical realm, our strategy may be used both to study the intrinsic difficulty of computing $p$-means (in terms of lower bounds), and to design effective algorithms. Indeed, as evidenced by the $S^1$ case, the combinatorial structure defined by the cut-loci of the points determines all key properties. A first case would be that of $p$-means on the unit sphere, for which there exist efficient algorithms to maintain arrangements of circles.

## References

1 Cgal, Computational Geometry Algorithms Library. http://www.cgal.org.

2 F. Allen and O. Johnson. Automated conformational analysis from crystallographic data. 4. statistical descriptors for a distribution of torsion angles. *Acta Crystallographica Section B: Structural Science*, 47(1):62–67, 1991.

3 M. Arnaudon and L. Miclo. Means in complete manifolds: uniqueness and approximation. *ESAIM: Probability and Statistics*, 18:185–206, 2014.

4 M. Arnaudon and L. Miclo. A stochastic algorithm finding $p$-means on the circle. *Bernoulli*, 22(4):2237–2300, 2016.

5 D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *ACM-SODA*, page 1035. Society for Industrial and Applied Mathematics, 2007.

6 F. Cazals and T. Dreyfus. The Structural Bioinformatics Library: modeling in biomolecular science and beyond. *Bioinformatics*, 7(33):1–8, 2017. doi:10.1093/bioinformatics/btw752.

7 F. Cazals, D. Mazauric, R. Tetley, and R. Watrigant. Comparing two clusterings using matchings between clusters of clusters. *ACM J. of Experimental Algorithms*, 24(1):1–42, 2019. doi:10.1145/3345951.

8 E-C. Chang, S.W. Choi, D.Y. Kwon, H. Park, and C. Yap. Shortest path amidst disc obstacles is computable. *International Journal of Computational Geometry Applications*, 16(05n06):567–590, 2006.

9 B. Charlier. Necessary and sufficient condition for the existence of a Fréchet mean on the circle. *ESAIM: Probability and Statistics*, 17:635–649, 2013.

10 L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13, 2007.

11 M. Fréchet. Les éléments aléatoires de nature quelconque dans un espace distancié. *Annales de l'institut Henri Poincaré*, 10(4):215–310, 1948.

12 K. Grove and H. Karcher. How to conjugatec 1-close group actions. *Mathematische Zeitschrift*, 132(1):11–20, 1973.

13 T. Hotz and s. Huckemann. Intrinsic means on the circle: Uniqueness, locus and asymptotics. *Annals of the Institute of Statistical Mathematics*, 67(1):177–193, 2015.

14 S.R. Jammalamadaka and A. SenGupta. *Topics in Circular Statistics.* World Scientific, 2001.

15 V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the fifteenth annual symposium on Computational geometry*, pages 351–359. ACM, 1999.

16 L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78, 2008.

17 A. Kobel, F. Rouillier, and M. Sagraloff. Computing real roots of real polynomials... and now for real! In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, pages 303–310. ACM, 2016.

18 C. Li, S. Pion, and C. Yap. Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming*, 64(1):85–111, 2005.

19 M. MacArthur and J. Thornton. Conformational analysis of protein structures derived from nmr data. *Proteins: Structure, Function, and Bioinformatics*, 17(3):232–251, 1993.

20 K. Mardia and P. Jupp. *Directional statistics*, volume 494. John Wiley and Sons, 2009.

21 J.-M. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of floating-point arithmetic.* Springer, 2018.

22 A. Ng. Clustering with the k-means algorithm. *Machine Learning*, 2012.

23 X. Pennec. Barycentric subspace analysis on manifolds. *The Annals of Statistics*, 46(6A):2711–2746, 2018.

24 F. Preparata and M. Shamos. *Computational geometry: an introduction.* Springer Science and Business Media, 1985.

**25** Maxim V Shapovalov and Roland L Dunbrack Jr. A smoothed backbone-dependent rotamer library for proteins derived from adaptive kernel density estimates and regressions. *Structure*, 19(6):844–858, 2011.

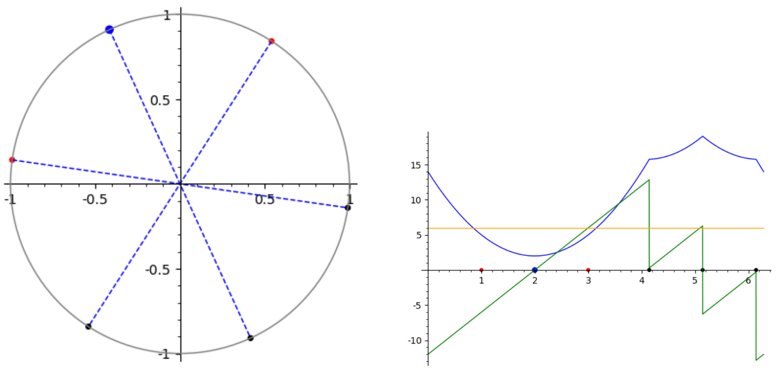
**26** D. Ting, G. Wang, M. Shapovalov, R. Mitra, M.I. Jordan, and R. Dunbrack. Neighbor-dependent ramachandran probability distributions of amino acids developed from a hierarchical dirichlet process model. *PLoS Comput Biol*, 6(4):e1000763, 2010.

**27** C. Yap and T. Dubé. The exact computation paradigm. In *Computing in Euclidean Geometry*, pages 452–492. World Scientific, 1995.

**28** J. Yu, C. Yap, Z. Du, S. Pion, and Hervé H. Brönnimann. The design of core 2: A library for exact numeric computation in geometry and algebra. In *International Congress on Mathematical Software*, pages 121–141. Springer, 2010.

## A Supporting information

### A.1 Algorithm



$n = 3, p = 2$ $n = 3, p = 2$

**Figure 7 An interval where $F_p$ has an algebraic expression and $F_p^{'}(\theta) = 0$.** Illustration of $F_p, F_p^{'}, F_p^{''}$ for $p = 2$ and three angles $\Theta_0 = \{\theta_1 = 1, \theta_2 = 2, \theta_3 = 3\}$. Color conventions as in Fig. 1. In this case, $F_2^{\prime}(\theta_2) = 0$, which must be numerically ascertained to ensure the correctness of the algorithm.

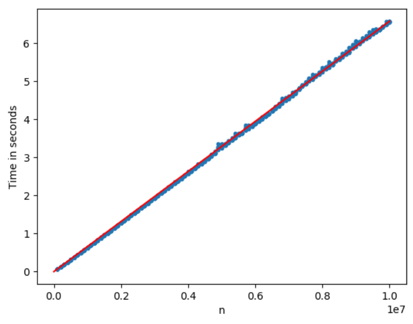
**Algorithm 2 Update_root**$(Sign)$: Updates the Root_remains buffer in main algorithm(Algo. 1).

```
1: $Sign$ ∈{positive,negative,null} // Sign of the derivative used to update the presence of roots on
   $(\alpha_j, \alpha_{j+1})$
2: Root_remains ← true // flag indicating whether a root must be sought on $(\alpha_j, \alpha_{j+1})$
3: if $Sign$ is negative then
4:     Root_remains ← true
5: else if $Sign$ is positive then
6:     Root_remains ← false
7: else if $Sign$ is null then
8:     Root_remains ← false
```

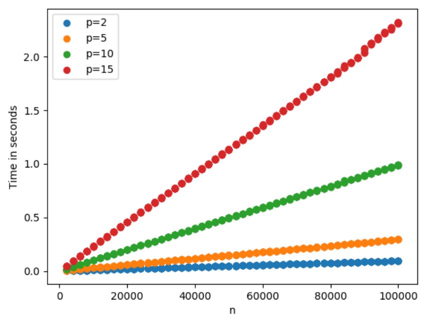
**Algorithm 3 Find_root**$(\alpha_{i-1}, \alpha_i)$: generic algorithm for $p > 5$.

```
1: $\alpha_{i-1}, \alpha_i$: the left and right endpoints of the initial interval
2: $\tau$: Threshold to stop binary search if interval is small enough
3: $c$: Center of interval $g$
4: $\theta_l \leftarrow \alpha_{i-1}, \theta_r \leftarrow \alpha_i$ // Interval being bisected
5: while Interval_too_wide($\theta_l, \theta_r$) do
6:     $c \leftarrow \theta_l + (\theta_r - \theta_l)/2$
7:     $S \leftarrow$ Sign($F_p^{'}(c)$)
8:     if $S$ is positive then
9:         $\theta_r \leftarrow c$
10:    else if $S$ is negative then
11:        $\theta_l \leftarrow c$
12:    else if $S$ is null then
13:        $\theta_r \leftarrow c$
14:        $\theta_l \leftarrow c$
15: $\theta_c \leftarrow \theta_l + (\theta_r - \theta_l)/2$
```

## A.2 Results



$p = 2$, $nmax = 10e^7$



$p \in \{2, 5, 10, 15\}$, $nmax = 10e^5$

**Figure 8 Fréchet mean: computation time depending as a function of $n$ and $p$.** The samples of size $n$ are generated at random angles at random in $[0, 2\pi)$. **(Left)** The red line joins $0, 0$ to the average time of the largest point sets($nmax = 10e^7$). **(Right)** Each color corresponds to a value of $p \in \{2, 5, 10, 15\}$.

# Multi-Level Weighted Additive Spanners

**Reyan Ahmed**
University of Arizona, Tucson, AZ, USA

**Greg Bodwin**
University of Michigan, Ann Arbor, MI, USA

**Faryad Darabi Sahneh**
University of Arizona, Tucson, AZ, USA

**Keaton Hamm**
University of Texas at Arlington, TX, USA

**Stephen Kobourov**
University of Arizona, Tucson, AZ, USA

**Richard Spence**
University of Arizona, Tucson, AZ, USA

## Abstract

Given a graph $G = (V, E)$, a subgraph $H$ is an *additive* $+\beta$ *spanner* if $\mathrm{dist}_H(u, v) \leq \mathrm{dist}_G(u, v) + \beta$ for all $u, v \in V$. A *pairwise spanner* is a spanner for which the above inequality is only required to hold for specific pairs $P \subseteq V \times V$ given on input; when the pairs have the structure $P = S \times S$ for some $S \subseteq V$, it is called a *subsetwise spanner*. Additive spanners in unweighted graphs have been studied extensively in the literature, but have only recently been generalized to weighted graphs.

In this paper, we consider a multi-level version of the subsetwise additive spanner in weighted graphs motivated by multi-level network design and visualization, where the vertices in $S$ possess varying level, priority, or quality of service (QoS) requirements. The goal is to compute a nested sequence of spanners with the minimum total number of edges. We first generalize the +2 subsetwise spanner of [Pettie 2008, Cygan et al., 2013] to the weighted setting. We experimentally measure the performance of this and several existing algorithms by [Ahmed et al., 2020] for weighted additive spanners, both in terms of runtime and sparsity of the output spanner, when applied as a subroutine to multi-level problem.

We provide an experimental evaluation on graphs using several different random graph generators and show that these spanner algorithms typically achieve much better guarantees in terms of sparsity and additive error compared with the theoretical maximum. By analyzing our experimental results, we additionally developed a new technique of changing a certain initialization parameter which provides better spanners in practice at the expense of a small increase in running time.

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

**Keywords and phrases** multi-level, graph spanner, approximation algorithms

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.16

**Related Version** *Full Version*: https://arxiv.org/abs/2102.05831

**Supplementary Material** All algorithms, implementations, the ILP solver, experimental data and analysis are available on Github:
*Software*: https://github.com/abureyanahmed/multi_level_weighted_additive_spanners
archived at swh:1:dir:95e49892297d01930d1de3c40e2bb2c39ccdd658

**Funding** The research for this paper was partially supported by NSF grants CCF-1740858, CCF1712119, and DMS-1839274.

**Acknowledgements** The authors wish to thank the anonymous reviewers for their comments.

## 1 Introduction

Given an undirected graph, a *spanner* is a sparse subgraph with approximately the same distance metric as the original graph. Spanners are used as a primitive for many algorithmic tasks involving the analysis of distances or shortest paths in enormous input graphs; it is often advantageous to first replace the graph with a spanner, which can be analyzed much more quickly and stored in much smaller space, at the price of a small amount of error. See the recent survey [5] for more details on these applications.

Spanners were first studied with multiplicative error, where for an input graph $G$ and an error ("stretch") parameter $k$, the spanner $H$ must satisfy $\text{dist}_H(s,t) \leq k \cdot \text{dist}_G(s,t)$ for all vertices $s, t$, where $\text{dist}_G(s,t)$ denotes the distance in $G$ between $s$ and $t$. This setting was quickly resolved in a seminal paper by Althöfer, Das, Dobkin, Joseph, and Soares [9], where the authors proved that for all positive integers $k$, all $n$-vertex graphs have spanners on $O(n^{1+1/k})$ edges with stretch $2k-1$, and that this tradeoff is the best possible. Thus, as expected, one can trade off error for spanner sparsity, increasing the stretch $k$ to pay more and more error for sparser and sparser spanners.

For very large graphs, additive error is arguably a much more appealing paradigm. Given $\beta > 0$, a $+\beta$ *spanner* of an $n$-vertex graph $G$ is a subgraph $H$ such that $\text{dist}_H(s,t) \leq \text{dist}_G(s,t) + \beta$ for all vertices $s, t$. Thus, for additive error the excess distance in $H$ is independent of the graph size and of $\text{dist}_G(s,t)$, which can be large when $n$ is large. Additive spanners were introduced by Liestman and Shermer [29], and followed by three landmark theoretical results on the sparsity of additive spanners in unweighted graphs: Aingworth, Chekuri, Indyk, and Motwani [8] showed that all graphs have $+2$ spanners on $O(n^{3/2})$ edges, Chechik [14, 18] showed that all graphs have $+4$ spanners on $O(n^{7/5})$ edges, and Baswana, Kavitha, Mehlhorn, and Pettie [12] showed that all graphs have $+6$ spanners on $O(n^{4/3})$ edges.

Despite the inherent appeal of additive error, spanners with multiplicative error remain much more commonly used in practice. There are two reasons for this.

1. First, while the multiplicative spanner of Althöfer et al [9] works without issue for weighted graphs, the previous additive spanner constructions hold only for unweighted graphs, whereas the metrics that arise in applications often require edge weights. Addressing this, recent work of the authors [3] and in two papers of Elkin, Gitlitz, and Neiman [23, 24] gave natural extensions of the classic additive spanner constructions to weighted graphs. For example, the $+2$ spanner bound becomes the following statement: for all $n$-vertex weighted graphs $G$, there is a subgraph $H$ satisfying $\text{dist}_H(s,t) \leq \text{dist}_G(s,t) + 2W(s,t)$, where $W(s,t)$ denotes the maximum edge weight along an arbitrary $s \rightsquigarrow t$ shortest path in $G$. The $+4$ spanner generalizes similarly, and the $+6$ spanner does as well with the small exception that the error increases to $+(6+\varepsilon)W(s,t)$, for arbitrarily small $\varepsilon > 0$ which trades off with the implicit constant in the spanner size.
2. Second, $\text{poly}(n)$ factors in spanner size can be quite serious in large graphs, and so applications often require spanners of near-linear size, say $O(n^{1.01})$ edges for an $n$-vertex input graph. The worst case spanner sizes of $O(n^{4/3})$ or greater for additive spanner constructions are thus undesirable, and unfortunately, there is a theoretical barrier to improving them: Abboud and Bodwin [1] proved that one *cannot* generally trade off more additive error for sparser spanners, as one can in the multiplicative setting. Specifically, for any constant $c > 0$, there is no general construction of $+c$ spanners for $n$-vertex input graphs on $O(n^{4/3-0.001})$ edges. However, the lower bound construction is rather pathological, and it is not likely to arise in practice. It is known that for many practical

graph classes, e.g., those with good expansion, near-linear additive spanners always exist [12]. Thus, towards applications of additive error, it is currently an important open question whether modern additive spanner constructions on *practical* graphs of interest tend to exhibit performance closer to the worst-case bounds from [1], or bounds closer to the best ones available for the given input graphs We remark here that there are strong computational barriers to designing algorithms that achieve the sparsest possible $+c$ spanners directly, or which even closely approximate this quantity in general [19].

The goal of this work is to address the second point, by measuring the experimental performance of the state-of-the-art constructions for weighted additive spanners on graphs generated from various popular random models and measuring their performance. We consider both $+cW$ spanners (where $W = \max_{uv \in E} w(uv)$ is the maximum edge weight) and $+cW(\cdot,\cdot)$ spanners, whose additive error is $+cW(s,t)$ for each pair $s,t \in V$. We are interested both in runtime and in the ratio of output spanner size to the size of the sparsest possible spanner (which we obtain using an ILP, discussed in Section 4). We specifically consider generalizations of the three staple constructions for weighted additive spanners mentioned above, in which the spanner distance constraint only needs to be satisfied for given pairs of vertices.

In particular, the following extensions are considered. A *pairwise spanner* is a subgraph that must satisfy the spanner error inequality for a given set of vertex pairs $P$ taken on input, and a *subsetwise spanner* is a pairwise spanner with the additional structure $P = S \times S$ for some vertex subset $S$. See e.g., [13, 15, 16, 21, 22, 27, 28, 32] for recent prior work on pairwise and subsetwise spanners, or the survey [5]. We then discuss a multi-level version of the subsetwise additive spanner where we have an edge-weighted graph $G = (V, E)$, a nested sequence of terminals $S_\ell \subseteq S_{\ell-1} \subseteq \cdots \subseteq S_1 \subseteq V$ and a real number $c \geq 0$ as input. We want to compute a nested sequence of subgraphs $H_\ell \subseteq H_{\ell-1} \subseteq \cdots \subseteq H_1$ such that $H_i$ is a $+cW$ subsetwise spanner of $G$ over $S_i$. The objective is to minimize the total number of edges in all subgraphs. Similar generalizations have been studied for the Steiner tree problem under various names including Multi-level Network Design [10], Quality of Service Multicast Tree (QoSMT) [17, 26], Priority Steiner Tree [20], Multi-Tier Tree [30], and Multi-level Steiner Tree [2, 7]. However, multi-level or QoS generalizations of spanner problems appear to have been much less studied in literature. These types of problems have applications in multi-level graph visualization and other network design problems where vertices may require different level or QoS requirements. Section 2 generalizes the clustering-based +2 subsetwise spanner [22] to weighted graphs, and Section 3 generalizes to the multi-level setting. Section 5 contains an experimental comparison between several different spanner algorithms given here and in [3] to infer that many of these spanner constructions typically achieve better error or sparsity bounds than the theoretical worst-case bounds. This comparison also suggests that spanners constructed by a certain light initialization technique given in [3] (Section 5.2.9) tend to outperform spanners constructed by clustering and path buying in terms of the sparsity; further, by varying an initialization parameter and selecting the sparsest spanner, we can compute much sparser additive spanners in random graphs at the expense of a logarithmic factor in running time.

## 2 Subsetwise spanners

All unweighted graphs have polynomially constructible +2 subsetwise spanners over $S \subseteq V$ on $O(n\sqrt{|S|})$ edges [22, 32]. For weighted graphs, Ahmed et al. [3] recently give a $+4W$ subsetwise spanner construction, also using $O(n\sqrt{|S|})$ edges. In this section we show how to generalize the $+2$ subsetwise construction [22, 32] to the weighted setting by giving a construction which produces a subsetwise $+2W$ spanner of a weighted graph (with integer edge weights in $[1, W]$) on $O(nW\sqrt{|S|})$ edges.

A *clustering* $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ is a set of disjoint subsets of vertices. Initially, every vertex is unclustered. The subsetwise $+2W$ construction has two steps: the clustering phase and the path buying phase. The clustering phase is exactly the same as that of [22, 32] in which we construct a cluster subgraph $G_{\mathcal{C}}$ as follows: set $\beta = \log_n \sqrt{|S|W}$, and while there is a vertex $v$ with at least $\lceil n^\beta \rceil$ unclustered neighbors, we add a cluster $C$ to $\mathcal{C}$ containing exactly $\lceil n^\beta \rceil$ unclustered neighbors of $v$ (note that $v \notin C$). We add to $G_{\mathcal{C}}$ all edges $vx$ ($x \in C$) and $xy$ ($x, y \in C$). When there are no more vertices with at least $\lceil n^\beta \rceil$ unclustered neighbors, we add all the unclustered vertices and their incident edges to $G_{\mathcal{C}}$.

In the second (path-buying) phase, we start with a clustering $\mathcal{C}$ and a cluster subgraph $G_0 := G_{\mathcal{C}}$. There are $z := \binom{|S|}{2}$ unordered pairs of vertices in $S$; let $\pi_1, \pi_2, \ldots, \pi_z$ denote the shortest paths between these vertex pairs where $\pi_i = \pi(u_i, v_i)$ has endpoints $\{u_i, v_i\}$. As in [22], we iterate from $i = 1$ to $i = z$ and determine whether to add path $\pi_i$ to the spanner. Define the cost and value of a path $\pi_i$ as follows:

$$\begin{aligned}
\text{cost}(\pi_i) &:= \#\text{ edges of } \pi_i \text{ which are absent in } G_{i-1} \\
\text{value}(\pi_i) &:= \#\text{ pairs } (x, C) \text{ where } x \in \{u_i, v_i\}, C \in \mathcal{C}, \\
&\qquad C \text{ contains at least one vertex in } \pi_i, \\
&\qquad \text{and } \text{dist}_{\pi_i}(x, C) < \text{dist}_{G_{i-1}}(x, C)
\end{aligned}$$

If $\text{cost}(\pi_i) \leq (2W + 1)\text{value}(\pi_i)$, then we add ("buy") $\pi_i$ to the spanner by letting $G_i = G_{i-1} \cup \pi_i$. Otherwise, we do not add $\pi_i$, and let $G_i = G_{i-1}$. The final spanner returned is $H = G_z$.

The unweighted $+2$ subsetwise spanner [22] and corresponding cluster subgraph $G_{\mathcal{C}}$ rely on the following properties:

- *Missing-edge property*: if an edge $uv \in E$ is absent in $G_{\mathcal{C}}$, then $u$ and $v$ belong to two different clusters
- *Cluster-diameter property*: the distance in $G_{\mathcal{C}}$ between two vertices in the same cluster is at most 2 ($2W$ for weighted graphs)

Using these properties, a lemma in [22] states that if a shortest $u$-$v$ path $\pi(u, v)$ contains $t \geq 1$ edges which are absent in $G_{\mathcal{C}}$, then there are at least $t/2$ clusters in $\mathcal{C}$ which contain at least one vertex on $\pi(u, v)$. This lemma does not quite hold in weighted graphs since a shortest path can pass through the same cluster many times; we instead prove the following generalization:

▶ **Lemma 1.** *Let $G$ be a weighted graph with edge weights in $[1, W]$ and let $\pi(u, v)$ be a shortest path which contains $t$ edges which are absent from $G_{\mathcal{C}}$. Then there are at least $t/(W + 1)$ clusters of $\mathcal{C}$ which contain at least one vertex on $\pi(u, v)$.*

**Proof.** Consider pairs $(x, e)$ where $e$ is an edge of $\pi(u, v)$ absent in $G_{\mathcal{C}}$ and $x$ is one of the endpoints of $e$. There are $t$ missing edges, so there are $2t$ such pairs. A cluster $C \in \mathcal{C}$ is said to *own* the pair $(x, e)$ if $x \in C$. By the missing-edge property, every missing edge is incident to two different clusters, so each pair $(x, e)$ is owned by some cluster.

Consider some cluster $C \in \mathcal{C}$ such that $\pi(u, v)$ passes through some vertex in $C$. By the cluster-diameter property and using the fact that all edges have weight at least 1, $\pi(u, v)$ cannot pass through more than $2W$ vertices in $C$. Using this we can show that $C$ owns at most $2W + 2$ pairs $(x, e)$. Since there are exactly $2t$ vertex-edge pairs and each cluster passing through some point in $\pi(u, v)$ owns at most $2W + 2$ pairs, we conclude that there are at least $\frac{2t}{2W+2} = \frac{t}{W+1}$ clusters which contain at least one vertex in $\pi(u, v)$. ◀

▶ **Lemma 2.** *For any $u_i, v_i \in S$, we have* $\text{dist}_H(u_i, v_i) \leq \text{dist}_G(u_i, v_i) + 2W$.

▶ **Lemma 3.** *For $\beta = \log_n \sqrt{|S|W}$, the $+2W$ subsetwise spanner $H$ has $O(nW\sqrt{|S|})$ edges.*

The proofs of these lemmas are largely the same as in [22] except we incur an additional $W$ in the size bound due to the cost vs. value when considering when to buy the path $\pi_i$ to the spanner.

▶ **Corollary 4.** *Let $G$ be a weighted graph with integer edge weights in $[1, W]$. Then $G$ has a $+6W$ pairwise spanner on $O(Wn|P|^{1/4})$ edges.*

This follows from applying the $+8W$ construction of Ahmed et al. [3] (Appendix A, Algorithm 3), except we use the above $+2W$ subsetwise spanner instead of the $+4W$ subsetwise spanner construction given in [3] as a subroutine.

## 3 Multi-level spanners

Here we study a multi-level variant of graph spanners. We first define the problem:

▶ **Definition 5** (Multi-level weighted additive spanner)**.** *Given a weighted graph $G(V, E)$ with maximum weight $W$, a nested sequence of subsets of vertices $S_\ell \subseteq S_{\ell-1} \subseteq \ldots \subseteq S_1 \subseteq V$, and $c \geq 0$, a multi-level $+cW$ spanner is a nested sequence of subgraphs $H_\ell \subseteq H_{\ell-1} \subseteq \ldots \subseteq H_1 \subseteq G$, where $H_i$ is a subsetwise $+cW$ spanner over $S_i$.*

Observe that Definition 5 generalizes the subsetwise spanner, which is a special case where $\ell = 1$. We define the *sparsity* of a multi-level spanner by $\text{sparsity}(\{H_i\}_{i=1}^{\ell}) := \sum_{i=1}^{\ell} |E(H_i)|$, where lower sparsity is more desirable. In the following sections, we also measure the quality of a multi-level spanner in terms of the ratio of its sparsity to the minimum possible sparsity over all candidate multi-level spanners (denoted by OPT).

The multi-level spanner can equivalently be phrased in terms of priorities and rates: each vertex $v \in S_1$ has a priority $P(v)$ between 1 and $\ell$ (namely, $P(v) = \max\{i : v \in S_i\}$), and we wish to compute a single subgraph containing edges of different rates such that for all $u, v \in S_1$, there is a $+cW$ spanner path consisting of edges of rate at least $\min\{P(u), P(v)\}$. With this, we will typically refer to the *priority* of $v$ to denote the highest $i$ such that $v \in S_i$, or 0 if $v \notin S_1$. In this section, we show that the multi-level version is not significantly harder than the ordinary "single-level" version: a subroutine which can compute an additive spanner can be used to compute a multi-level spanner whose sparsity is comparably good.

We first describe a simple rounding-up approach based on an algorithm by Charikar et al. [17] for the QoSMT problem, a similar generalization of the Steiner tree problem. For this approach, assume we have a subroutine which computes an exact or approximate single-level subsetwise spanner. Given $v \in S_1$, let $P(v) \in [1, \ell]$ denote the priority of $v$. The rounding-up approach is as follows: for each $v$, round $P(v)$ up to the nearest power of 2. This effectively constructs a "rounded-up" instance where all vertices in $S_1$ have priority 1, 2, 4, ..., or $2^{\lceil \log_2 \ell \rceil}$. The sparsity of the optimum solution in the rounded-up instance is at most 2 OPT; given the optimum solution to the original instance with sparsity OPT, a feasible solution to the rounded-up instance with sparsity at most 2 OPT can be obtained by rounding up the rate of each edge to the nearest power of 2.

For each $i \in \{1, 2, 4, \ldots, 2^{\lceil \log_2 \ell \rceil}\}$, use the subroutine to compute a level-$i$ subsetwise spanner over all vertices whose rounded-up priority is at least $i$. The final multi-level additive spanner is obtained by taking the union of these computed spanners, by keeping an edge at the highest level it appears in. This requires $O(\log \ell)$ calls to the single-level subroutine.

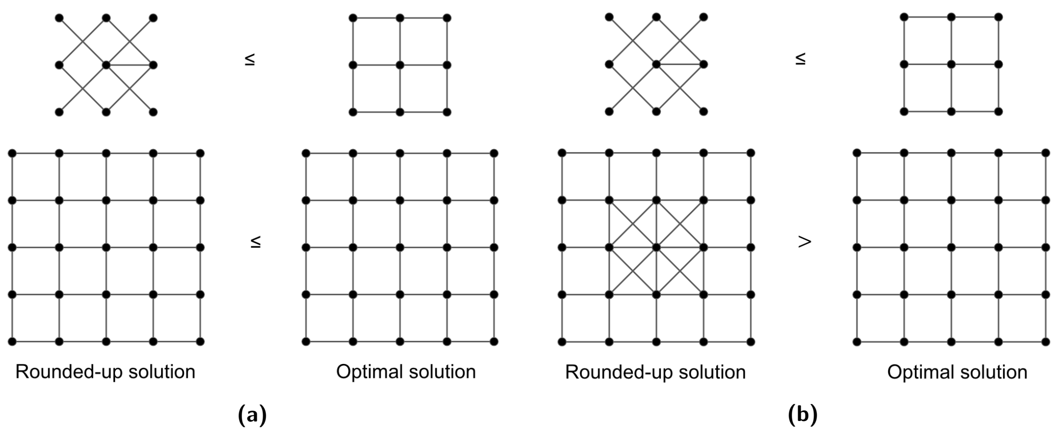


**Figure 1** (a) The rounding-up approach computes an optimal spanner at each level (assuming an exact subroutine), so the sizes of the spanners on each level are at most that of the optimal solution ($9+40$ edges vs. $12+40$). (b) However, when an edge is present in a top-level solution, it must be present in lower-level solutions. The rounding-up approach takes the union of the spanners in the bottom level; in this case, the sparsity of the rounded-up solution ($9+48$ vs. $12+40$) is greater than that of the optimum.

▶ **Theorem 6.** *Assuming an exact subsetwise spanner subroutine, the solution computed by the rounding-up approach has sparsity at most* $4 \cdot \mathrm{OPT}$.

This is proved using the same ideas as the $4\rho$-approximation for QoSMT [17]. As mentioned earlier, in practice we use an approximation algorithm to compute the subsetwise spanner instead of computing the minimum spanner.

▶ **Theorem 7.** *There exists a* $\tilde{O}(n/\sqrt{|S_1|})$*-approximation algorithm to compute multi-level* $+2W$ *spanners when* $W = O(\log n)$.

This follows from using the $+2W$ subsetwise construction in Section 2. The approximation ratio of this subsetwise spanner algorithm is $O(nW/\sqrt{|S|})$ as the construction produces a spanner of size $O(nW\sqrt{|S|})$, while the sparsest additive spanner trivially has at least $|S|-1 = \Omega(|S|)$ edges.

We also show that, under certain conditions, if we have a subroutine which computes a subsetwise spanner of $G$, $S$ of size $O(n^a|S|^b)$ where $a$ and $b$ are absolute constants, a very naïve algorithm can be used to obtain a multi-level spanner also with sparsity $O(n^a|S_1|^b)$.

▶ **Theorem 8.** *Suppose there is an absolute constant* $0 < \alpha < 1$ *such that* $|S_i| \le \alpha|S_{i-1}|$ *for all* $i \in \{1, \ldots, \ell\}$. *Then we can compute a multi-level spanner with sparsity* $O(n^a|S_1|^b)$.

**Proof.** Consider the following simple construction: for each $i \in \{1, 2, 3, \ldots, \ell\}$, compute a level-$i$ subsetwise spanner of size $O(n^a|S_i|^b)$. Consider the union of these spanners, by keeping each edge at the highest level it appears. The sparsity of the returned multi-level spanner is at most

$$\begin{aligned}\mathrm{sparsity}(\{H_i\}) &= O(n^a|S_1|^b + 2n^a|S_2|^b + 3n^a|S_3|^b + \ldots + \ell n^a|S_\ell|^b)\\ &\le O(n^a|S_1|^b(1 + 2\alpha^b + 3\alpha^{2b} + \ldots + \ell\alpha^{(\ell-1)b}))\\ &= O(n^a|S_1|^b)\end{aligned}$$

where we used the arithmetico-geometric series $1 + 2(\alpha^b) + 3(\alpha^b)^2 + \ldots = \frac{1}{(1-\alpha^b)^2}$ which is constant for fixed $\alpha$, $b$. Note that $0 < \alpha < 1$ and $b > 0$, which implies $0 < \alpha^b < 1$. ◀

The assumption that $|S_i| \leq \alpha |S_{i-1}|$ for some constant $\alpha$ is fairly natural, as many realistic networks tend to have significantly fewer hubs than non-hubs.

▶ **Corollary 9.** *Under the assumption $|S_i| \leq \alpha|S_{i-1}|$ for all $i \in \{2, \ldots, \ell\}$, there exists a poly-time algorithm which computes a multi-level $+2$ spanner of sparsity $O(n\sqrt{|S_1|})$.*

**Proof.** This follows by using the $+2$ construction by Cygan et al. [22] on $O(n\sqrt{|S|})$ edges as the subroutine. ◀

## 4 Integer programming formulation

To compute a minimum size $+cW$ spanner over vertex pairs $P$, we utilize a slight modification of the ILP in [5, Section 9], wherein we choose the specific distortion function $f(t) = t + cW$ and minimize the sparsity rather than total weight of the spanner. For completeness, we present the full ILP for computing a single-level additive subsetwise spanner below along with a brief description of the multi-level extension. Here $E'$ represents the bidirected edge set, obtained by adding directed edges $(u, v)$ and $(v, u)$ for each edge $uv \in E$. The binary variable $x^{uv}_{(i,j)}$ is 1 if edge $(i, j)$ is included on the selected $u$-$v$ path and 0 otherwise, and $w(e)$ is the weight of edge $e$.

$$\text{Minimize} \sum_{e \in E} x_e \text{ subject to} \tag{1}$$

$$\sum_{(i,j)\in E'} x^{uv}_{(i,j)} w(e) \leq \text{dist}_G(u, v) + cW \qquad \forall (u, v) \in P;\ e = ij \tag{2}$$

$$\sum_{(i,j)\in Out(i)} x^{uv}_{(i,j)} - \sum_{(j,i)\in In(i)} x^{uv}_{(j,i)} = \begin{cases} 1 & i = u \\ -1 & i = v \\ 0 & \text{else} \end{cases} \qquad \forall (u, v) \in P;\ \forall i \in V \tag{3}$$

$$\sum_{(i,j)\in Out(i)} x^{uv}_{(i,j)} \leq 1 \qquad \forall (u, v) \in P;\ \forall i \in V \tag{4}$$

$$x^{uv}_{(i,j)} + x^{uv}_{(j,i)} \leq x_e \qquad \forall (u, v) \in P;\ \forall e = ij \in E \tag{5}$$

$$x_e, x^{uv}_{(i,j)} \in \{0, 1\} \tag{6}$$

Inequalities (3)–(4) enforce that for each $(u, v) \in P$, the selected edges corresponding to $u$, $v$ form a path; inequality (2) enforces that the length of this path is at most $\text{dist}_G(u, v) + cW$ (note that $W$ may be replaced with $W(u, v)$). Inequality (5) ensures that if $x^{uv}_{(i,j)} = 1$ or $x^{uv}_{(i,j)} = 1$, then edge $ij$ is taken.

To generalize the ILP formulation to the multi-level problem, we take a similar set of variables for every level. The rest of the constraints are similar, except we define $x^k_e = 1$ if edge $e$ is present on level $k$ and the variables $x^{uv}_{(i,j)}$ are also indexed by level. We add the constraint $x^k_e \leq x^{k-1}_e$ for all $k \in \{2, \ldots, \ell\}$ which enforces that if edge $e$ is present on level $k$, it is also present on all lower levels. Finally, the objective is to minimize the sparsity $\sum_{k=1}^{\ell} \sum_{e\in E} x^k_e$.

## 5 Experiments

In this section, we provide experimental results involving the rounding-up framework described in Section 3. This framework needs a single level subroutine; we use the $+2W$ subsetwise construction in Section 2 and the three pairwise $+2W(\cdot, \cdot)$, $+4W(\cdot, \cdot)$, $+6W$ constructions provided in [3][1] (see Appendix A). We generate multi-level instances and solve the instances using the ILP and the four approximation algorithms. We consider natural questions about how the number of levels $\ell$, number of vertices $n$, and decay rate of terminals with respect to levels affect the running times and (experimental) approximation ratios, defined as the sparsity of the returned multi-level spanner divided by OPT.

We used CPLEX 12.6.2 as an ILP solver in a high-performance computer for all experiments (Lenovo NeXtScale nx360 M5 system with 400 nodes). Each node has 192 GB of memory. We have used Python for implementing the algorithms and spanner constructions. Since we have run the experiment on several thousand instances, we run the solver for four hours per instance.

## 5.1 Experiment parameters

We run experiments first to test experimental approximation ratio vs. the parameters, and then to test running time vs. parameters. Each set of experiments has several parameters: the graph generator, the number of levels $\ell$, the number of vertices $n$, and how the size of the terminal sets $S_i$ (vertices requiring level or priority at least $i$) decrease as $i$ decreases.

In what follows, we use the Erdős–Rényi (ER) [25], Watts–Strogatz (WS) [33], Barabási–Albert (BA) [11], and random geometric (GE) [31] models. Let $p$ be the edge selection probability. If we set $p = (1+\varepsilon)\frac{\ln n}{n}$, then the generated Erdős–Rényi graph is connected with high probability for $\varepsilon > 0$ [25]. For our experiments we use $\varepsilon = 1$. In the Watts-Strogatz model, we initially create a ring lattice of constant degree $K$. For our experiments we use $K = 6$ and $p = 0.2$. In the Barabási–Albert model, a new vertex is connected to $m$ existing vertices. For our experiments we use $m = 5$. In the random geometric graph model, two vertices are connected to each other if their Euclidean distance is not larger than a threshold $r_c$. For $r_c = \sqrt{\frac{(1+\epsilon)\ln n}{\pi n}}$ with $\epsilon > 0$, the synthesized graph is connected with a high probability [31]. We generate a set of small graphs ($10 \le n \le 40$) and a set of large graphs ($50 \le n \le 500$). We only compute the exact solutions for the small graphs since the ILP has an exponential running time. In this paper, we provide the results of Erdős–Rényi graphs since it is the most popular model. However, the radius[2] of Erdős–Rényi graphs is relatively small; in our dataset, the range of the radius is 2-4. Hence, we also provide the results of random geometric graphs which have larger radius (4-12). The remaining results and the radius distribution of different generators are available at the supplement Github link. We consider number of levels $\ell \in \{1, 2, 3\}$ for small graphs, $\ell \in \{1, \dots, 10\}$ for large graphs, and adopt two methods for selecting terminal sets: *linear* and *exponential*. A terminal set $S_1$ with lowest priority of size $n(1 - \frac{1}{\ell+1})$ in the linear case and $\frac{n}{2}$ in the exponential case is chosen uniformly at random. For each subsequent level, $\frac{1}{\ell+1}$ vertices are deleted at random in the linear case, whereas half the remaining vertices are deleted in the exponential case. Levels/priorities and terminal sets are related via $S_i = \{v \in S_1 : P(v) \ge i\}$. We choose edge weights $w(e)$ independently uniformly at random from $\{1, 2, 3 \dots, 10\}$.

An experimental instance of the multi-level problem here is thus characterized by four parameters: graph generator, number of vertices $n$, number of levels $\ell$, and terminal selection method TSM $\in$ {Linear,Exponential}. As there is randomness involved, we generated five instances for every choice of parameters (e.g., ER, $n = 30$, $\ell = 2$, Linear). For each

[1] Note that, one can show that the $+2W$, $+4W$, $+8W$ spanners in [3] are actually $+2W(.,.)$, $+4W(.,.)$ and $+6W$ spanners respectively by using a tighter analysis [4].

[2] The minimum over all $v \in V$ of $\max_{w \in V} d_G(v, w)$ where $d_G(v, w)$ is the graph distance (by number of edges, not total weight) between $v$ and $w$

instance of the small graphs, we compute the approximate solution using either the $+2W$, $+2W(\cdot,\cdot)$, $+4W$, or $+6W$ spanner subroutine, and the exact solution using the ILP described in Section 4. We compute the experimental approximation ratio ("Ratio") by dividing the sparsity of the approximate solution by the sparsity of the optimum solution (OPT). For large graphs, we only compute the approximate solution.

## 5.2 Results

We consider different spanner constructions as the single level subroutine in the rounding-up approach described in Section 3. We first consider the $+2W$ subsetwise construction (Section 2).

### 5.2.1 Multi-level $+2W$ spanner

We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 2. The average experimental ratio increases as $n$ increases approximately linearly in $n$, which is expected since the theoretical approximation ratio of $\tilde{O}(n/\sqrt{|S_1|})$ is proportional to $n$. The average and minimum experimental ratio does not change significantly as the number of levels increases; however, the maximum ratio increases. The experimental ratio of the linear terminal selection method is slightly better compared to that of the exponential method.

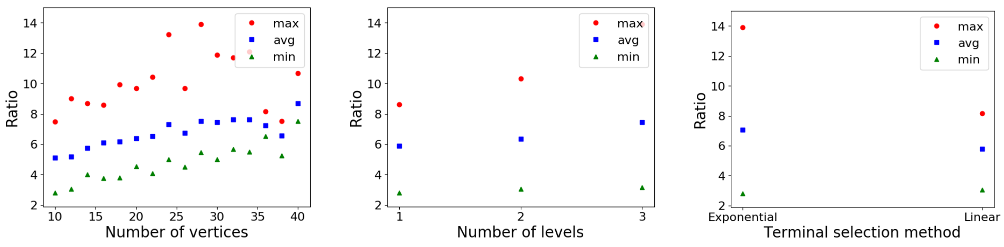


**Figure 2** Performance of the algorithm that uses $+2W$ subsetwise spanner as the single level subroutine on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

### 5.2.2 Multi-level $+2W(\cdot,\cdot)$ spanner

We now consider the $+2W(\cdot,\cdot)$ pairwise construction [3] (Algorithm 1, Appendix A) as a subroutine, with $P = S \times S$. We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 3. The average experimental ratio increases as $n$ increases. This is again expected since the theoretical approximation ratio is proportional to $n$. The average and minimum experimental ratio do not change that much as the number of levels increases, however, the maximum ratio increases. The experimental ratio of the linear terminal selection method is also slightly better compared to that of the exponential method.

### 5.2.3 Comparison between global and local spanners

One major difference between the subsetwise and pairwise construction is the subsetwise construction considers the (global) maximum edge weight $W$ of the graph in the error. On the other hand, the $+cW(\cdot,\cdot)$ spanners consider the (local) maximum edge weight in a shortest path for each pair of vertices $s, t$. We provide a comparison between the global and local settings.

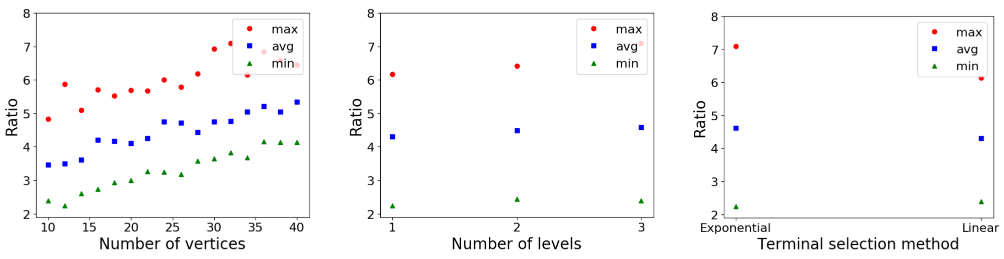


**Figure 3** Performance of the algorithm that uses $+2W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

We describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 4. The average experimental ratio increases as $n$ increases for both global and local settings. However, the ratio of the local setting is smaller compared to that of the global setting. One reason for this difference is the solution to the global exact algorithm is relatively smaller since the global setting considers larger errors. The ratio of the global setting increases as the number of levels increases and for the exponential terminal selection method. For the local setting, the ratio does not change significantly.

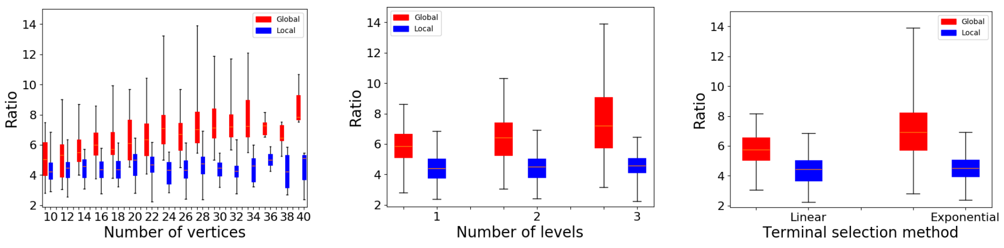


**Figure 4** Performance of the global (subsetwise $+2W$) and local (pairwise $+2W(\cdot,\cdot)$) construction-based algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

## 5.2.4 Multi-level $+4W(\cdot,\cdot)$ spanner

We now consider the $+4W(\cdot,\cdot)$ pairwise construction [3] (Algorithm 2, Appendix A) as a single level subroutine. We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 5. The average experimental ratio increases as $n$ increases. This is expected since the theoretical approximation ratio is proportional to $n$. The average experimental ratio does not change significantly as the number of levels increases; however, the maximum ratio increases. The experimental ratio of the linear terminal selection method is also slightly better compared to that of the exponential method.

## 5.2.5 Comparison between $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ spanners

We now provide a comparison between the pairwise $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ construction-based approximation algorithms. We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 6. The average experimental ratio increases as $n$ increases for both $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ settings. The $+4W(\cdot,\cdot)$ construction-based algorithm slightly outperforms the $+2W(\cdot,\cdot)$ algorithm for $\ell = 3$ and exponential selection method.

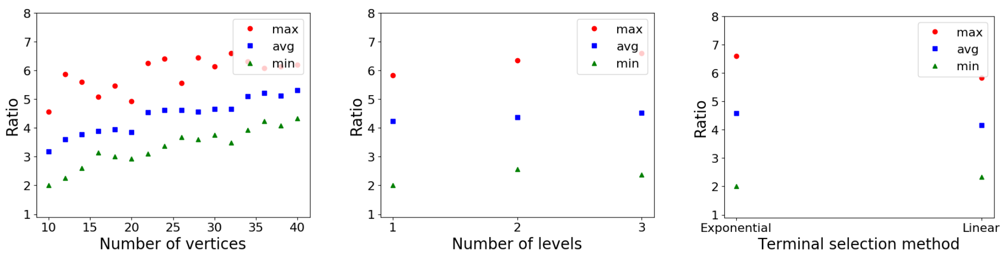


**Figure 5** Performance of the algorithm that uses $+4W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

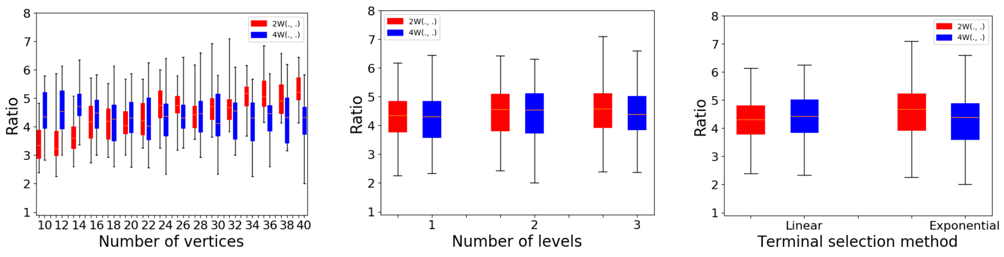


**Figure 6** Performance of the pairwise $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ construction-based algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

### 5.2.6 Multi-level $+6W$ spanner

We now consider the $+6W$ pairwise construction [3] (Algorithm 3, Appendix A) as a single level subroutine. We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 7. The average experimental ratio increases as $n$ increases. This is expected since the theoretical approximation ratio is proportional to $n$. The average experimental ratio does not change significantly as the number of levels increases; however, the maximum ratio increases. The maximum and average experimental ratios of the linear terminal selection method are slightly better compared to that of the exponential method.

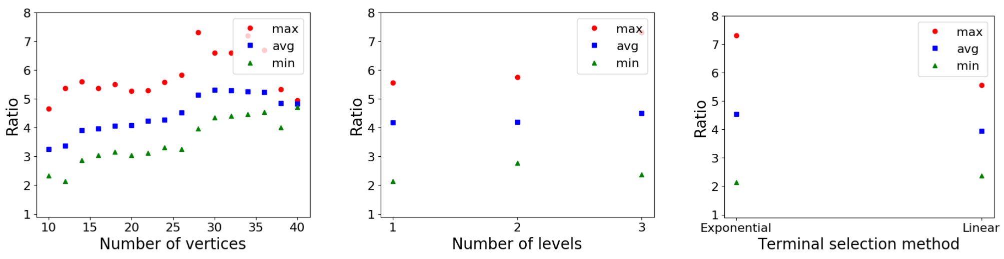


**Figure 7** Performance of the algorithm that uses $+6W$ pairwise spanner as the single level subroutine on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

### 5.2.7 Comparison between $+2W$ and $+6W$ spanners

We now provide a comparison between the pairwise $+2W$ and $+6W$ construction-based approximation algorithms. We first describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 8. The average experimental ratio increases as $n$ increases for both $+2W$ and $+6W$ settings. As the number of vertices increases, the ratio of the $+6W$ construction-based algorithm gets smaller. This is expected since a larger error makes the problem easier to solve. Similarly, as $\ell$ increases, the $+6W$ construction-based algorithm outperforms the $+2W$ algorithm. The average experimental ratio of the $+6W$ construction based algorithm is smaller both in the linear and exponential terminal selection methods.

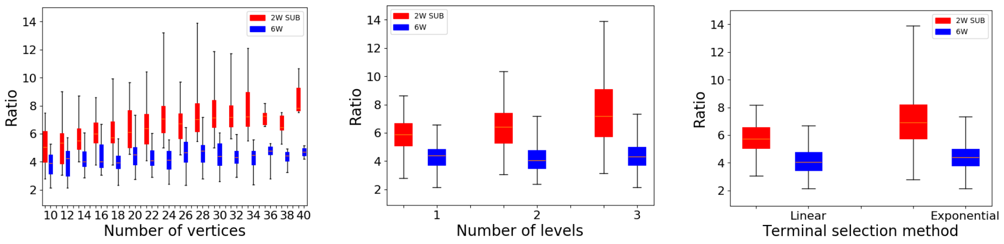


**Figure 8** Performance of the pairwise $+2W$ and $+6W$ construction-based algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

## 5.2.8 Experiment on large graphs

We generate some large instances on up to 500 vertices and run different multi-level spanner algorithms on them. We use $n = \{50, 100, 150, \ldots, 500\}$ and $\ell = \{1, 2, 3, \ldots, 10\}$. We describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 9. We are comparing four multi-level algorithms, namely those using the $+2W$ subsetwise and $+2W(\cdot,\cdot)$, $+4W(\cdot,\cdot)$, $+6W$ pairwise constructions [3] as subroutines with $P = S \times S$. Since computing the optimal solution exactly via ILP is computationally expensive on large instances, we report the ratio in terms of relative sparsity, defined as the sparsity of the multi-level spanner returned by one algorithm divided by the minimum sparsity over the spanners returned by all four. The ratio of the $+6W$ construction based algorithm is lowest and the $+2W$ construction based algorithm is highest. This is expected since a higher additive error generally reduces the number of edges needed. Overall the ratio decreases as $n$ increases. This is because the significance of small additive error reduces as the graph size and distances get larger. The relative ratio for the $+2W$ construction increases as $\ell$ increases, and for the exponential terminal selection method.

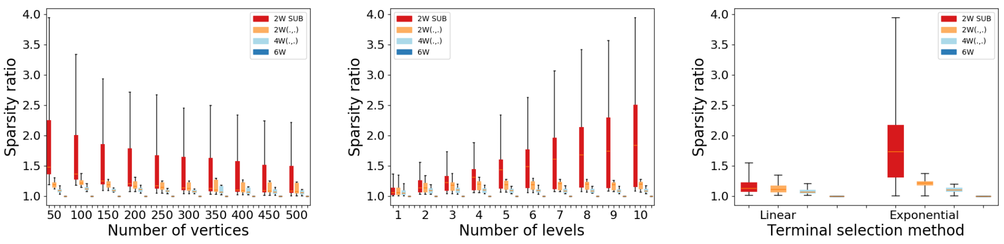


**Figure 9** Performance of different approximation algorithms on large Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

### 5.2.9 Impact of the initialization parameters

It is worth mentioning that the +2 subsetwise spanner [22] and $+2W$ subsetwise spanner (Section 2) begin with a clustering phase, while the algorithms described in Appendix A begin with a $d$-light initialization. In $d$-light initialization, we add the $d$ lightest edges incident to each vertex, where $d \geq 1$ is a parameter specific to the algorithm; these edges tend to be on shortest paths. In practice, there may be relatively few edges which appear on shortest paths and some of these edges might be redundant. Hence, we compute $+2W(\cdot, \cdot)$ spanners with different values of $d$. We describe the experimental results on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 10. We have computed the ratio as described in Section 5.2.8. From the figures, we see that as we reduce the value of $d$ exponentially, the ratio decreases: in particular, the optimal choice of parameter $d$ in practice might be significantly smaller than the optimal value of $d$ in theory. Generally, it could make sense in practical implementations of spanner algorithms to try all values $\{d, d/2, d/4, d/8, \dots\}$, computing $\approx \log_2 d$ different spanners, and then use only the sparsest one. This costs only a $O(\log d)$ factor in the running time of the algorithm, which is typically reasonable.

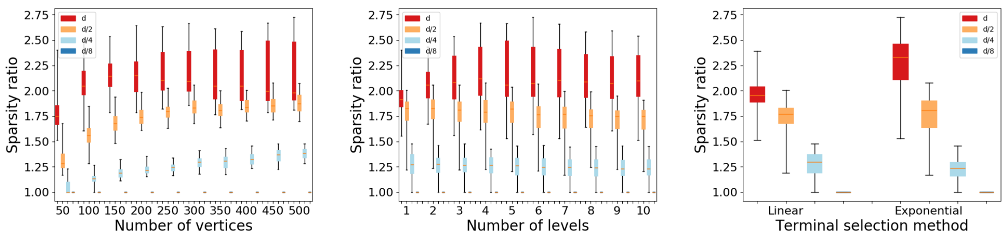


**Figure 10** Impact of different values of $d$ on large Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

## 5.3 Running time

We now provide the running times of the different algorithms. We show the running time of the ILP on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 11. The running time of the ILP increases exponentially as $n$ increases, as expected. The execution time of a single level instance with 45 vertices is more than 64 hours using a 28 core processor. Hence, we kept the number of vertices less than or equal to 40 for our small graphs. The experimental running time should increase as $\ell$ increases, but we do not see that pattern in these plots because some of the instances were not able to finish in four hours.

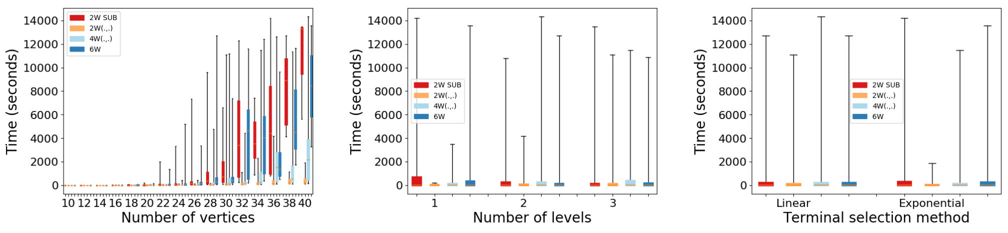


**Figure 11** Running time of all exact algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

We provide the experimental running time of the approximation algorithm on Erdős–Rényi graphs in Figure 12. The running time of the $+2W$ construction-based algorithm is the largest. Overall, the running time increases as $n$ increases. There is no straightforward relation between the running time and $\ell$. Although the number of calls to the single level subroutine increases as $\ell$ increases, it also depends on the size of the subset in a single level. Hence, if the subset sizes are larger, then it may take longer for small $\ell$. The running time of the linear method is larger.

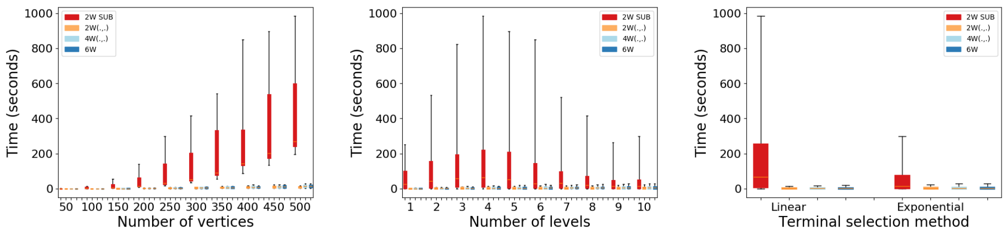


**Figure 12** Running time of all approximation algorithms on large Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

The running times appear reasonable in other settings too; see the supplemental Github repository and the arXiv version [6] of this paper for details and experimental results.

## 5.4 Experimental additive error

Different spanner constructions provide theoretical guarantees on the maximum amount of additive error. For example, a $+2W$ subsetwise spanner over $G$, $S$ ensures that any pair of vertices in $S$ does not have an error of more than $+2W$. Similarly, the $+2W(\cdot,\cdot)$, $+4W(\cdot,\cdot)$, and $+6W$ pairwise construction ensures that the error in the shortest path distance in the spanner is no more than $+2W(\cdot,\cdot)$, $+4W(\cdot,\cdot)$, and $+6W$ respectively. These are theoretical upper bounds that directly appear from the construction. However, in our experiment, we have found that the theoretical upper bound is never achieved, and most vertex pairs contain a less additive error. We define the *error ratio* of an additive spanner $H$ to be the sum of additive errors in $H$ (over all vertex pairs) divided by the maximum possible sum of errors. For example, if $H$ is a $+2W(\cdot,\cdot)$ spanner over vertex pairs $P \subseteq V \times V$, then

$$\text{error ratio} := \frac{\sum_{(u,v)\in P}(d_H(u,v) - d_G(u,v))}{\sum_{(u,v)\in P} 2W(u,v)}.$$

For a multi-level spanner, we define the error ratio similarly, except we additionally sum the numerator (and denominator) over all subgraphs $H_i$ from $i = 1$ to $\ell$. We consider the $+2W(\cdot,\cdot)$ pairwise construction [3] (Algorithm 1) as a single level subroutine and compute the error ratios using Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 13. Figure 13 suggests that the average error ratio is typically less than 0.05; in other words, the $+2W(\cdot,\cdot)$ spanner algorithm outputs a spanner whose average additive error is around 5% of the maximum allowable error. We provide the comparison among all algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 14.

## 5.5 Relative sparsity

In most of the previous figures, we provide the sparsity (number of edges) of the spanner, relative to the optimum spanner. Here we provide a comparison of the spanner sparsities of the four spanner algorithms ($+2W$ subsetwise, $+2W(\cdot,\cdot)$, $+4W(\cdot,\cdot)$, and $+6W$) relative to

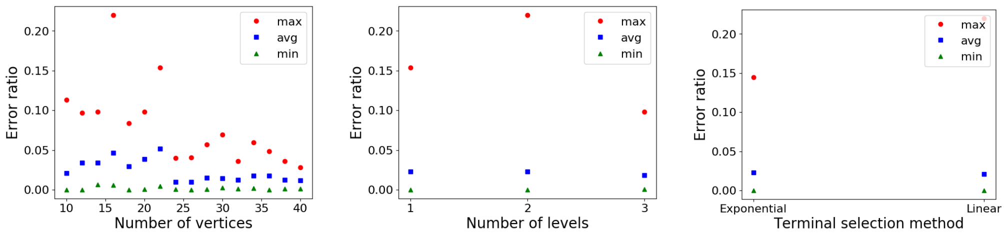


**Figure 13** Average error ratios of the spanners computed using the algorithm that uses $+2W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

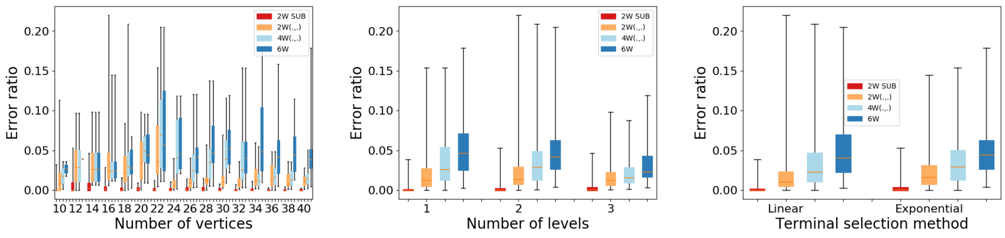


**Figure 14** Average error ratios of the spanners computed using all algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

each other in Figure 15. According to Figure 15, the $+6W$ construction-based approximation algorithm uses the fewest number of edges, and the $+2W$ subsetwise spanner typically outputs a spanner with 50% to 75% more edges than the $+6W$ spanner; this is expected as more additive error generally leads to sparser spanners in practice.

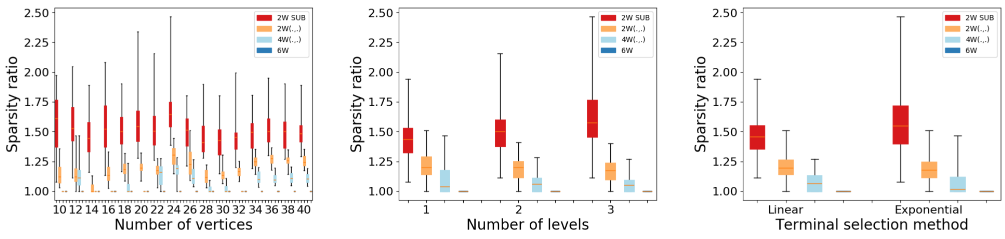


**Figure 15** Sparsity ratio of the spanners computed using all algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

## 5.6 Amount of reduction

One of the major goals of graph spanners is to sparsify the input graph without losing significant information on distances in the original graph. A straightforward way to compute the amount of reduction is to take the ratio of the number of edges removed from the input graph in the spanner to the number of edges in the input graph. Since we are computing multi-level spanners, we define the reduction ratio as the number of edges removed from the input divided by $|E|\ell$ (in other words, $(|E|\ell - \text{sparsity}(\{H_i\}))/(|E|\ell)$). We describe these reduction ratios on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 16. The global $+2W$ construction-based approximation algorithm provides the smallest reduction ratio which is also consistent with the idea that sparser spanners generally take on more additive error. Remember that the $+2W$ construction uses clustering and other construction uses $d$-initialization. This result again indicates that the clustering-based approach performs worse compared to the initialization-based approach.

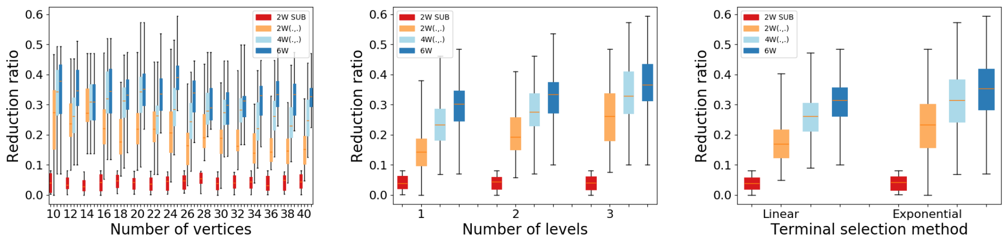


**Figure 16** Reduction ratio of the spanners computed using all algorithms on Erdős–Rényi graphs w.r.t. $n$, $\ell$, and terminal selection method.

## 6 Conclusion

We have provided a framework where we can use different spanner subroutines to compute multi-level spanners of varying additive error. We additionally introduced a generalization of the $+2$ subsetwise spanner [22] to integer edge weights, and illustrate that this can reduce the $+8W$ error in [3] to $+6W$. A natural question is to provide an approximation algorithm that can handle different additive error for different levels. We also provided an ILP to compute the optimum spanner; computing this optimally is very slow, so natural directions include using techniques such as graph reduction to sparsify the input graph before computing a spanner. The experimental results in Section 5 suggest that the $+2W$ clustering-based approach is slower and returns worse spanners than the initialization based approaches. We provided a method of changing the initialization parameter $d$ which reduces the sparsity in practice.

## References

1 Amir Abboud and Greg Bodwin. The 4/3 additive spanner exponent is tight. *Journal of the ACM (JACM)*, 64(4):1–20, 2017.

2 Abu Reyan Ahmed, Patrizio Angelini, Faryad Darabi Sahneh, Alon Efrat, David Glickenstein, Martin Gronemann, Niklas Heinsohn, Stephen Kobourov, Richard Spence, Joseph Watkins, and Alexander Wolff. Multi-level Steiner trees. In *17th International Symposium on Experimental Algorithms, (SEA)*, pages 15:1–15:14, 2018. doi:10.4230/LIPIcs.SEA.2018.15.

3 Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Stephen Kobourov, and Richard Spence. Weighted additive spanners. In Isolde Adler and Haiko Müller, editors, *Graph-Theoretic Concepts in Computer Science*, pages 401–413. Springer, 2020.

4 Reyan Ahmed, Greg Bodwin, Keaton Hamm, Stephen Kobourov, and Richard Spence. Weighted sparse and lightweight spanners with local additive error. *arXiv preprint arXiv:2103.09731*, 2021.

5 Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Mohammad Javad Latifi Jebelli, Stephen Kobourov, and Richard Spence. Graph spanners: A tutorial review. *Computer Science Review*, 37:100253, 2020.

6 Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Keaton Hamm, Stephen Kobourov, and Richard Spence. Multi-level weighted additive spanners. *arXiv preprint arXiv:2102.05831*, 2021.

7 Reyan Ahmed, Faryad Darabi Sahneh, Keaton Hamm, Stephen Kobourov, and Richard Spence. Kruskal-based approximation algorithm for the multi-level Steiner tree problem. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:21, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ESA.2020.4.

8 Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

9 Ingo Althöfer, Gautam Das, David Dobkin, and Deborah Joseph. Generating sparse spanners for weighted graphs. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 26–37, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

10 Anantaram Balakrishnan, Thomas L. Magnanti, and Prakash Mirchandani. Modeling and heuristic worst-case performance analysis of the two-level network design problem. *Management Sci.*, 40(7):846–867, 1994. doi:10.1287/mnsc.40.7.846.

11 Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

12 Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. Additive spanners and ($\alpha$, $\beta$)-spanners. *ACM Transactions on Algorithms (TALG)*, 7(1):5, 2010.

13 Greg Bodwin. Linear size distance preservers. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 600–615. Society for Industrial and Applied Mathematics, 2017.

14 Greg Bodwin. A note on distance-preserving graph sparsification. *arXiv preprint arXiv:2001.07741*, 2020.

15 Greg Bodwin and Virginia Vassilevska Williams. Better distance preservers and additive spanners. In *Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 855–872. Society for Industrial and Applied Mathematics, 2016. URL: http://dl.acm.org/citation.cfm?id=2884435.2884496.

16 Hsien-Chih Chang, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Near-Optimal Distance Emulator for Planar Graphs. In *Proceedings of 26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, pages 16:1–16:17, 2018.

17 M. Charikar, J. Naor, and B. Schieber. Resource optimization in QoS multicast routing of real-time multimedia. *IEEE/ACM Transactions on Networking*, 12(2):340–348, April 2004. doi:10.1109/TNET.2004.826288.

18 Shiri Chechik. New additive spanners. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 498–512. Society for Industrial and Applied Mathematics, 2013.

19 Eden Chlamtáč, Michael Dinitz, Guy Kortsarz, and Bundit Laekhanukit. Approximating spanners and directed Steiner forest: Upper and lower bounds. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 534–553. SIAM, 2017.

20 Julia Chuzhoy, Anupam Gupta, Joseph (Seffi) Naor, and Amitabh Sinha. On the approximability of some network design problems. *ACM Trans. Algorithms*, 4(2):23:1–23:17, 2008. doi:10.1145/1361192.1361200.

21 Don Coppersmith and Michael Elkin. Sparse sourcewise and pairwise distance preservers. *SIAM Journal on Discrete Mathematics*, 20(2):463–501, 2006.

22 Marek Cygan, Fabrizio Grandoni, and Telikepalli Kavitha. On pairwise spanners. In *Proceedings of 30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20, pages 209–220, 2013.

23 Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Almost shortest paths and PRAM distance oracles in weighted graphs. *arXiv preprint arXiv:1907.11422*, 2019.

24 Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Improved weighted additive spanners. *arXiv preprint arXiv:2008.09877*, 2020.
25 Paul Erdős and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
26 Marek Karpinski, Ion I. Mandoiu, Alexander Olshevsky, and Alexander Zelikovsky. Improved approximation algorithms for the quality of service multicast tree problem. *Algorithmica*, 42(2):109–120, 2005. doi:10.1007/s00453-004-1133-y.
27 Telikepalli Kavitha. New pairwise spanners. *Theory of Computing Systems*, 61(4):1011–1036, 2017.
28 Telikepalli Kavitha and Nithin M. Varma. Small stretch pairwise spanners and approximate $d$-preservers. *SIAM Journal on Discrete Mathematics*, 29(4):2239–2254, 2015.
29 Arthur Liestman and Thomas Shermer. Additive graph spanners. *Networks*, 23:343–363, July 1993. doi:10.1002/net.3230230417.
30 Prakash Mirchandani. The multi-tier tree problem. *INFORMS J. Comput.*, 8(3):202–218, 1996.
31 Mathew Penrose. *Random geometric graphs*. Number 5. Oxford University Press, 2003.
32 Seth Pettie. Low distortion spanners. *ACM Transactions on Algorithms (TALG)*, 6(1):7, 2009.
33 Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440, 1998.

## A Pairwise spanner constructions [3]

Here, we provide pseudocode (Algorithms 1–3) describing the $+2W$, $+4W$, and $+8W$ pairwise spanner constructions[3] by Ahmed et al. [3]. These spanner constructions have a similar theme: first, construct a *d-light initialization*, which is a subgraph $H$ obtained by adding the $d$ lightest edges incident to each vertex (or all edges if the degree is at most $d$). Then for each pair $(s,t) \in P$, consider the number of edges in $\pi(s,t)$ which are absent in the current subgraph $H$. Add $\pi(s,t)$ to $H$ if the number of missing edges is at most some threshold $\ell$, or otherwise randomly sample vertices and either add a shortest path tree rooted at these vertices, or construct a subsetwise spanner among them.

**Algorithm 1** $+2W$ pairwise spanner [3].

```
1:  $d = |P|^{1/3}$, $\ell = n/|P|^{2/3}$
2:  $H = d$-light initialization
3:  let $m'$ be the number of missing edges needed for a valid construction
4:  while $m' > nd$ do
5:      for $(s,t) \in P$ do
6:          $x = |E(\pi(s,t)) \setminus E(H)|$
7:          if $x \le \ell$ then
8:              add $\pi(s,t)$ to $H$
9:      $R =$ random sample of vertices, each with probability $1/(\ell d)$
10:     for $r \in R$ do
11:         add a shortest path tree rooted at $r$ to each vertex
12: add the $m'$ missing edges
13: return $H$
```

[3] Using a tighter analysis or the above $+2W$ subsetwise construction in place of the $+4W$ construction in Algorithm 3, the additive error can be improved to $+2W(\cdot,\cdot)$, $+4W(\cdot,\cdot)$, and $+6W$ for integer edge weights.

**Algorithm 2** $+4W$ pairwise spanner [3].

```
1:  $d = |P|^{2/7}$, $\ell = n/|P|^{5/7}$
2:  $H$ = $d$-light initialization
3:  let $m'$ be the number of missing edges needed for a valid construction
4:  while $m' > nd$ do
5:      for $(s,t) \in P$ do
6:          $x = |E(\pi(s,t)) \setminus E(H)|$
7:          if $x \leq \ell$ then
8:              add $\pi(s,t)$ to $H$
9:          else if $x \geq n/d^2$ then
10:             $R_1$ = random sample of vertices, each w.p. $d^2/n$
11:             add a shortest path tree rooted at each $r \in R_1$
12:         else
13:             add first $\ell$ and last $\ell$ missing edges of $\pi(s,t)$ to $H$
14:             $R_2$ = i.i.d. sample of vertices, w.p. $1/(\ell d)$
15:             for each $r, r' \in R_2$ do
16:                 if exists $r \to r'$ path missing $\leq n/d^2$ edges then
17:                     add to $H$ a shortest $r \to r'$ path among paths missing $\leq n/d^2$ edges
18: add the $m'$ missing edges
19: return $H$
```

**Algorithm 3** $+8W$ pairwise spanner [3].

```
1:  $d = |P|^{1/4}$, $\ell = n/|P|^{3/4}$
2:  $H$ = $d$-light initialization
3:  let $m'$ be the number of missing edges needed for a valid construction
4:  while $m' > nd$ do
5:      for $(s,t) \in P$ do
6:          $x = |E(\pi(s,t)) \setminus E(H)|$
7:          if $x \leq \ell$ then
8:              add $\pi(s,t)$ to $H$
9:          else
10:             add first $\ell$ and last $\ell$ missing edges of $\pi(s,t)$ to $H$
11:             $R$ = random sample of vertices, each w.p. $1/(\ell d)$
12:             $H'$ = $+4W$ subsetwise $(R \times R)$-spanner [3]
13:             add $H'$ to $H$
14: add the $m'$ missing edges
15: return $H$
```

## B Experiments

In the main paper, we mostly discussed the experimental results of Erdős–Rényi graphs. In this section, we provide the results of random geometric graphs. The plots of Watts–Strogatz and Barabási–Albert graphs are available in the Github repository.

## B.1 Multi-level $+2W$ spanner

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection methods in Figure 17. In both cases the average ratio increases as $n$ and $\ell$ increases. The average ratio is relatively lower for the linear terminal selection method.

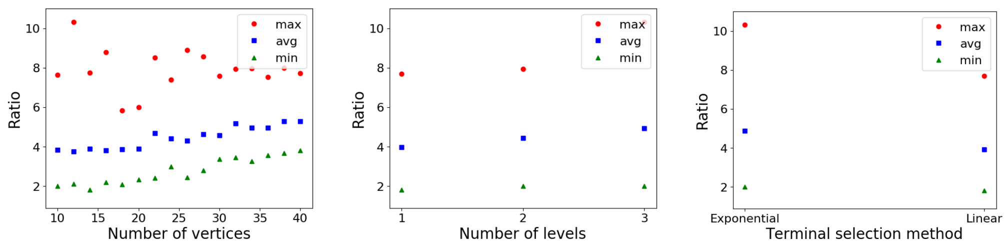


**Figure 17** Performance of the algorithm that uses $+2W$ subsetwise spanner as the single level subroutine on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.2 Multi-level $+2W(\cdot,\cdot)$ spanner

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 18. The average experimental ratio increases as $n$ increases. The maximum ratio increases as $\ell$ increases. Again, the experimental ratio of the linear terminal selection method is relatively smaller compared to the exponential method.

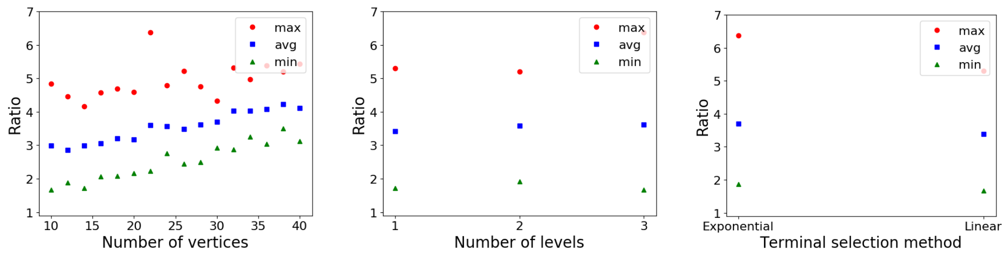


**Figure 18** Performance of the algorithm that uses $+2W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.3 Comparison between global and local error

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 19. The ratio of the local setting is smaller compared to the global setting.

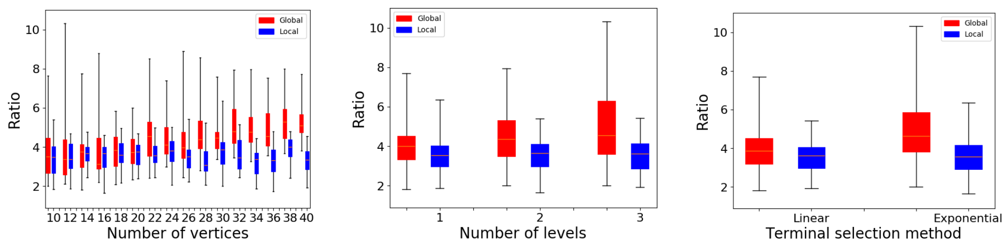


**Figure 19** Performance of the global and local construction-based algorithms on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.4 Multi-level $+4W(\cdot,\cdot)$ spanner

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 20. The experimental ratio increases as the number of vertices increases. The maximum ratio increases as the number of levels increases. Again, the experimental ratio of the linear terminal selection method is relatively smaller compared to the exponential method.

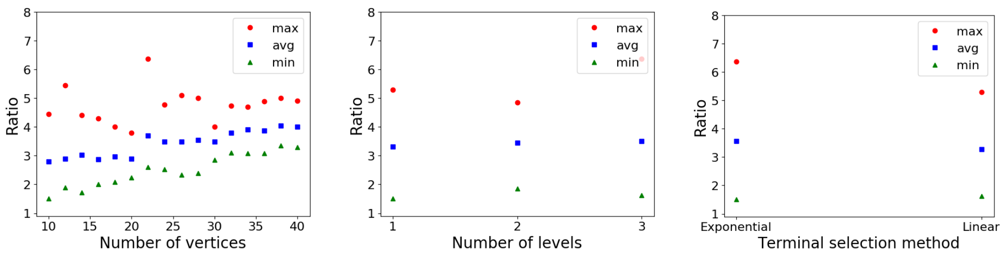


**Figure 20** Performance of the algorithm that uses $+4W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.5 Comparison between $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ setups

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 21. As $n$ increases the average ratio of $+4W(\cdot,\cdot)$-based approximation algorithm becomes smaller compared to the $+2W(\cdot,\cdot)$-based algorithm. The average ratio of $+4W(\cdot,\cdot)$ is relatively smaller for the exponential terminal selection method.

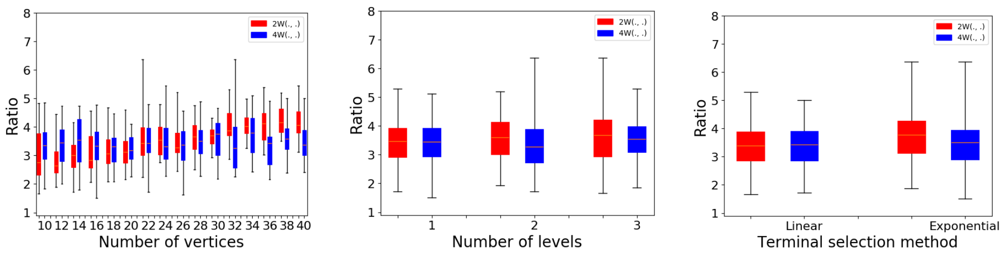


**Figure 21** Performance of the pairwise $+2W(\cdot,\cdot)$ and $+4W(\cdot,\cdot)$ construction-based algorithms on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.6 Multi-level $+6W$ spanner

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method in Figure 22. The experimental ratio increases as the number of vertices increases. The maximum ratio increases as the number of levels increases. Again, the experimental ratio of the linear terminal selection method is relatively smaller compared to the exponential method.

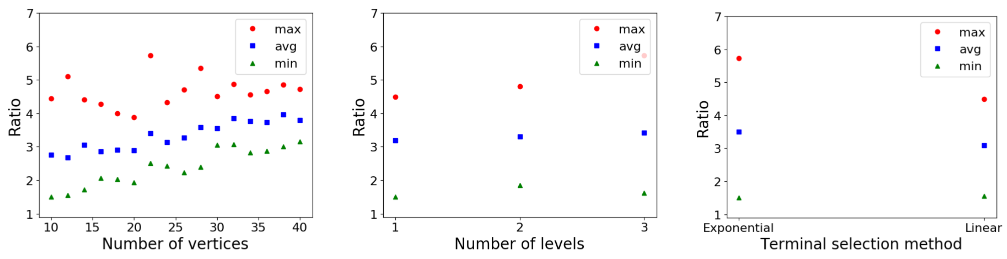


**Figure 22** Performance of the algorithm that uses $+6W(\cdot,\cdot)$ pairwise spanner as the single level subroutine on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.7 Comparison between $+2W$ and $+6W$ setups

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 23. We can see that as $n$ gets larger the ratio of $+6W$ gets smaller. The situation is similar when $\ell$ increases.

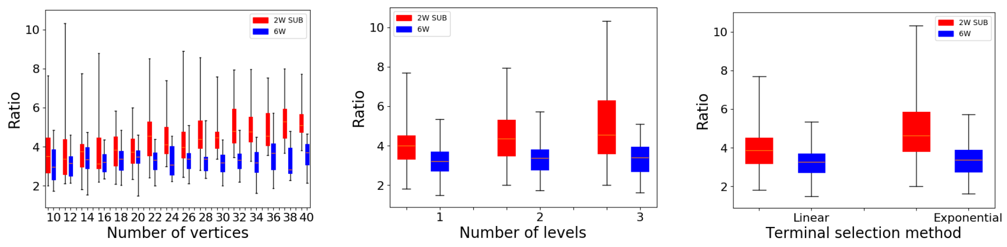


**Figure 23** Performance of the pairwise $+2W$ and $+6W$ construction-based algorithms on random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.8 Experiment on large graphs

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 24.

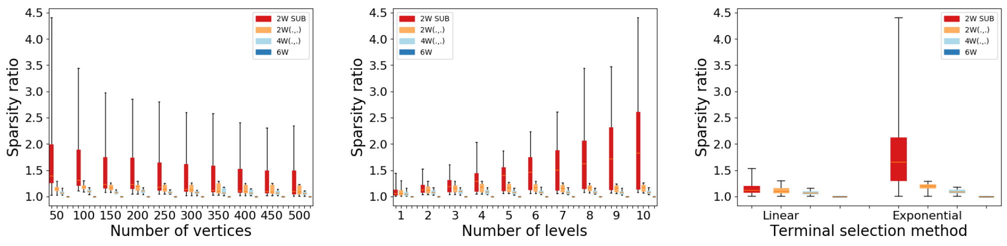


**Figure 24** Performance of different approximation algorithms on large random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

## B.9 Impact of the initialization parameters

We describe the experimental results on random geometric graphs w.r.t. $n$, $\ell$, and the terminal selection method in Figure 25. Again, the experiment suggests that we can exponentially reduce the value of $d$ and take the solution that has a minimum number of edges, with an additional cost of $O(\log d)$ running time.

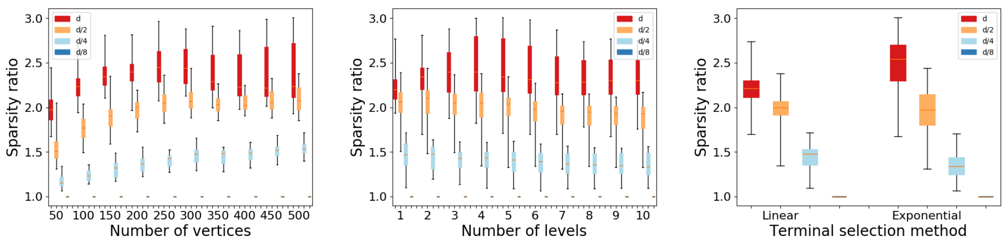


**Figure 25** Impact of different values of $d$ on large random geometric graphs w.r.t. $n$, $\ell$, and terminal selection method.

# Targeted Branching for the Maximum Independent Set Problem

**Demian Hespe**
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

**Sebastian Lamm**
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

**Christian Schorr**
Karlsruhe Institute of Technology, Institute for Theoretical Informatics, Germany

## Abstract

Finding a maximum independent set is a fundamental NP-hard problem that is used in many real-world applications. Given an unweighted graph, this problem asks for a maximum cardinality set of pairwise non-adjacent vertices. In recent years, some of the most successful algorithms for solving this problem are based on the branch-and-bound or branch-and-reduce paradigms. In particular, branch-and-reduce algorithms, which combine branch-and-bound with reduction rules, have been able to achieve substantial results, solving many previously infeasible real-world instances. These results were to a large part achieved by developing new, more practical reduction rules. However, other components that have been shown to have a significant impact on the performance of these algorithms have not received as much attention. One of these is the branching strategy, which determines what vertex is included or excluded in a potential solution. Even now, the most commonly used strategy selects vertices solely based on their degree and does not take into account other factors that contribute to the performance of the algorithm.

In this work, we develop and evaluate several novel branching strategies for both branch-and-bound and branch-and-reduce algorithms. Our strategies are based on one of two approaches which are motivated by existing research. They either (1) aim to decompose the graph into two or more connected components which can then be solved independently, or (2) try to remove vertices that hinder the application of a reduction rule which can lead to smaller graphs. Our experimental evaluation on a large set of real-world instances indicates that our strategies are able to improve the performance of the state-of-the-art branch-and-reduce algorithm by Akiba and Iwata. To be more specific, our reduction-based packing branching rule is able to outperform the default branching strategy of selecting a vertex of highest degree on 65% of all instances tested. Furthermore, our decomposition-based strategy based on edge cuts is able to achieve a speedup of 2.29 on sparse networks (1.22 on all instances).

**2012 ACM Subject Classification** Mathematics of computing → Graph algorithms; Theory of computation → Branch-and-bound; Mathematics of computing → Combinatorial optimization

**Keywords and phrases** Graphs, Combinatorial Optimization, Independent Set, Vertex Cover, Clique, Branch-and-Reduce, Branch-and-Bound, Data Reduction

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.17

**Related Version** *Full Version*: `https://arxiv.org/abs/2102.01540`

**Supplementary Material** *Software (Source Code)*: `https://github.com/Hespian/CutBranching`
archived at `swh:1:dir:ba9fbe7f94ffb2b8563f5057a97481404d236781`

## 1 Introduction

An *independent set* of a graph $G = (V, E)$ is a set of vertices $I \subseteq V$ of $G$ such that no two vertices in this set are adjacent. The problem of finding such an independent set of maximum cardinality, the *maximum independent set problem*, is a fundamental NP-hard problem [15].

Its applications cover a wide variety of fields including computer graphics [33], network analysis [31], route planning [24] and computational biology [4, 8]. In computer graphics for instance, large independent sets can be used to optimize the traversal of mesh edges in a triangle mesh. Further applications stem from its complementary problems minimum vertex cover and maximum clique.

One of the best known techniques for finding maximum independent sets, both in theory [39, 7] and practice [1], are *data reduction algorithms*. These algorithms apply a set of reduction rules to decrease the size of an instance while maintaining the ability to compute an optimal solution afterwards. A recently successful type of data reduction algorithm is so-called *branch-and-reduce algorithms* [1, 19], which exhaustively apply a set of reduction rules to compute an irreducible graph. If no further rule can be applied, the algorithm branches into (at least) two smaller subproblems, which are then solved recursively. To make them more efficient in practice, these algorithms also make use of problem-specific upper and lower bounds to quickly prune the search space.

Due to the practical impact of data reduction, most of the research aimed at improving the performance of branch-and-reduce algorithms so far has been focused on either proposing more practically efficient special cases of already existing rules [6, 9], or maintaining dependencies between reduction rules to reduce unnecessary checks [2, 20]. However, improving other aspects of branch-and-reduce has been shown to benefit its performance [30]. The branching strategy in particular has been shown to have a significant impact on the running time [1]. Up to now, the most frequently used branching strategy employed in many state-of-the-art solvers selects branching vertices solely based on their degree. Other factors, such as the actual reduction rules used during the algorithm are rarely taken into account. However, recently there have been some attempts to incorporate such branching strategies for other problems such as finding a maximum $k$-plex [14].

## 1.1 Contribution

In this paper, we propose and examine several novel strategies for selecting branching vertices. These strategies follow two main approaches that are motivated by existing research: (1) Branching on vertices that decompose the graph into several connected components that can be solved independently. Solving components individually has been shown to significantly improve the performance of branch-and-reduce in practice, especially when the size of the largest component is small [2]. (2) Branching on vertices whose removal leads to reduction rules becoming applicable again. In turn, this leads to a smaller reduced graph and thus improved performance. For each approach we present several concrete strategies that vary in their complexity. Finally, we evaluate their performance by comparing them to the aforementioned default strategy used in the state-of-the-art solver by Akiba and Iwata [1]. For this purpose we make use of a wide spectrum of instances from different graph classes and applications. Our experiments indicate that our strategies are able to find an optimal solution faster than the default strategy on a large set of instances. In particular, our reduction-based packing rule is able to outperform the default strategy on 65% of all instances. Furthermore, our decomposition-based strategies achieve a speedup of 1.22 (over the default strategy) over all instances. A more detailed explanation of a previous version of this work can be found in Schorr's Bachelor's thesis [35].

## 2 Preliminaries

Let $G = (V, E)$ be an undirected graph, where $V = \{0, \dots, n-1\}$ is a set of $n$ vertices and $E \subseteq \{\{u, v\} \mid u, v \in V\}$ is a set of $m$ edges. We assume that $G$ is *simple*, i. e., it has no self loops or multi-edges. The *(open) neighborhood* of a vertex $v \in V$ is denoted by $N(v) = \{u \mid \{v, u\} \in E\}$. Furthermore, we denote the *closed neighborhood* of a vertex by $N[v] = N(v) \cup \{v\}$. We define the open and closed neighborhood of a set of vertices $U \subseteq V$ as $N(U) = \cup_{u \in U} N(u) \setminus U$ and $N[U] = N(U) \cup U$, respectively. The *degree* of a vertex $v \in V$ is the size of its neighborhood $d(v) = |N(v)|$ and $\Delta = \max_{v \in V}\{d(v)\}$. For a vertex $v \in V$, we further define $N^2(v) = N(N(v))$.

For a subset of vertices $V_S \subseteq V$, the *(vertex-)induced subgraph* $G[V_S] = (V_S, E_S)$ is given by restricting the edges of $G$ to vertices of $V_S$, i. e., $E_S = \{\{u, v\} \in E \mid u, v \in V_S\}$. Likewise, for a subset of edges $E_S \subseteq E$, the *edge-induced subgraph* $G[E_S] = (V_S, E_S)$ is given by restricting the vertices of $G$ to the endpoints of edges in $E_S$, i. e., $V_S = \{u, v \in V \mid \{u, v\} \in E_S\}$. For a subset of vertices $U \subset V$, we further define $G - U$ as the induced subgraph $G[V \setminus U]$.

A *path* $P = (v_1, \dots, v_k)$ of length $k$ is a sequence of $k$ distinct vertices in $G$ such that $\{v_i, v_{i+1}\} \in E$ for all $i \in \{1, \dots, k-1\}$. A subgraph of $G$ induced by a maximal subset of vertices that are connected by a path is called a *connected component*. Furthermore, a graph that only contains one connected component is called *connected*. Likewise, a graph with more than one connected component is called *disconnected*. A subset $S \subset V$ of a connected graph $G$ is called a *vertex separator* if the removal of $S$ from $G$ makes the graph disconnected.

An *independent set* of a graph is a subset of vertices $I \subseteq V$ such that no two vertices of $I$ are adjacent. A *maximum independent set* (MIS) is an independent set of maximum cardinality. Closely related to independent set are vertex covers and cliques. A *vertex cover* is a set of vertices $C \subseteq V$ such that for each edge $\{u, v\} \in E$ either $u$ or $v$ is contained in $C$. The complement of a (maximum) independent set of a graph $G$ is a *(minimum) vertex cover* (MVC) of $G$. A *clique* is a subset of vertices $K \subseteq V$ such that all vertices of $K$ are adjacent to each other, i. e., $\forall u, v \in K : \{u, v\} \in E$. Finally, a (maximum) independent set of a graph $G$ is a *(maximum) clique* (MC) in the complement graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{\{u, v\} \mid \{u, v\} \notin E\}$.

## 3 Related Work

The most commonly used branching strategy for MIS and MVC is to select a vertex of maximum degree. Fomin et al. [13] show that using a vertex of maximum degree that also minimizes the number of edges between its neighbors is optimal with respect to their complexity measure. The algorithm by Akiba and Iwata [1] (which we augment with our new branching rules) also uses this strategy. Akiba and Iwata also compare this strategy to branching on a vertex of minimum degree and a random vertex. They show that both of these perform significantly worse than branching on a maximum degree vertex.

Xiao and Nagamochi [39] also use this strategy in most cases. For dense subgraphs, however, they use an edge branching strategy: They branch on an edge $\{u, v\}$ where $|N(u) \cap N(v)|$ is sufficiently large (depending on the maximum degree of the graph) by excluding both $u$ and $v$ in one branch and applying the alternative reduction (see Section 5.2) to $\{u\}$ and $\{v\}$ in the other branch.

Bourgeois et al. [3] use maximum degree branching as long as there are vertices of degree at least five. Otherwise, they utilize specialized algorithms to solve subinstances with an average degree of three or four. Those algorithms perform a rather complex case analysis to find a suitable branching vertex. The analysis is based on exploiting structures that contain 3- or 4-cycles. Branching on specific vertices in such structures often enables further reduction rules to be applied.

Chen et al. [7] use a notion of *good pairs* that are advantageous for branching. They chose these good pairs by a set of rules which are omitted here. They combine these with so-called *tuples* of a set of vertices and the number of vertices from this set that have to be included in an MIS. This information can be used when branching on a vertex contained in that set to remove further vertices from the graph. Akiba and Iwata [1] use the same concept in their *packing* rule. Chen et al. combine good pairs, tuples and high degree vertices for their branching strategy.

Most algorithms for MC (e.g. [36, 37]) compute a greedy coloring and branch on vertices with a high coloring number. More sophisticated MC algorithms use MaxSAT encodings to prune the set of branching vertices [26, 27, 29]. Li et al. [28] combine greedy coloring and MaxSAT reasoning the further reduce the number of branching vertices.

Another approach used for MC is using the *degeneracy order* $v_1 < v_2 < \cdots < v_n$ where $v_i$ is a vertex of smallest degree in $G - \{v_1, \ldots v_{i-1}\}$. Carraghan and Pardalos [5] present an algorithm that branches in descending degeneracy order. Li et al. [26] introduce another vertex ordering using iterative maximum independent set computations (which might be easier than MC on some graphs) and breaking ties according to the degeneracy order.

The algorithm by Akiba and Iwata [1] is a so-called *branch-and-reduce* algorithm: It repeatedly reduces the instance size by a set of polynomial-time reduction rules and then branches on a vertex once no more reduction rules can be applied. Since branching removes at least one vertex from the graph, more reduction rules might be applicable afterwards. The set of reductions used in their algorithm is relatively large and not covered completely here. However, some reduction rules are explained in Section 5 where we show how to target particular reduction rules when branching. Akiba and Iwata apply the reduction rules in a predefined order. For each rule, their algorithm iterates over all vertices in the graph and checks whether the rule can be applied. If a rule is applied successfully, this process is restarted from the first reduction rule. In order to prune the search space, bounds on the largest possible independent set of a branch are computed. They implement three different methods for determining upper bounds: clique cover, LP relaxation and cycle cover. Additionally, they employ special reduction rules that can be applied during branching. Another optimization done by their algorithm is to solve connected components separately. We utilize this in Section 4 where we introduce branching rules that decompose the graph into connected components. We use this algorithm as the base implementation to test our new branching strategies.

## 4 Decomposition Branching

Our first approach to improve the default branching strategy found in many state-of-the-art algorithms (including that of Akiba and Iwata [1]) is to decompose the graph into several connected components. Subsequently, processing these components individually has been shown to improve the performance of branch-and-reduce in practice [2]. To this end, we now present three concrete strategies with varying computational complexity: articulation points, edge cuts and nested dissections.

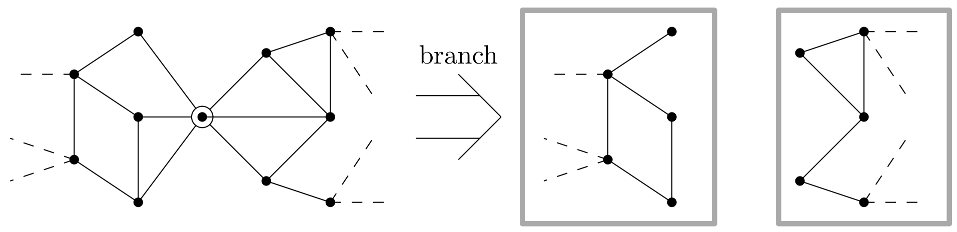


**Figure 1** Branching on an articulation point (circled vertex) decomposes the graph into two connected components (gray boxes) that can be solved independently. The graphic shows the branch in which the vertex is excluded from the independent set.

## 4.1 Articulation Points

First, we are concerned with finding single vertices that are able to decompose a graph into at least two separated components. Such points are called *articulation points* (or cut vertices). Articulation points can be computed in linear time $\mathcal{O}(n+m)$ using a simple depth-first search (DFS) algorithm (see Hopcroft and Tarjan [21] for a detailed description). In particular, a vertex $v$ is an articulation point if it is either the root of the DFS tree and has at least two children or any non-root vertex that has a child $u$, such that no vertex in the subtree rooted at $u$ has a back edge to one of the ancestors of $v$.

For our first branching strategy we maintain a set of articulation points $A \subseteq V$. When selecting a branching vertex, we first discard all invalid vertices from $A$, i. e., vertices that were removed from the graph by a preceding data reduction step. If this results in $A$ becoming empty, a new set of articulation points is computed on the current graph in linear time. However, if no articulation points exist, we select a vertex based on the default branching strategy. Otherwise, if $A$ contains at least one vertex, an arbitrary one from $A$ is selected as the branching vertex. Figure 1 illustrates branching on an articulation point.

Even though this strategy introduces only a small (linear) overhead, finding articulation points can be rare depending on the type of graph. This results in the default branching strategy being selected rather frequently. Furthermore, our preliminary experiments indicate that articulation points are rarely found at higher depth. However, due to their low overhead, we can justify searching for them whenever $A$ becomes empty.

## 4.2 Edge Cuts

To alleviate the restrictive nature of finding articulation points, we now propose a more flexible branching strategy based on *(minimal) edge cuts*. In general, we aim to find small vertex separators, i. e., a set of vertices whose removal disconnects the graph. We do so by making use of minimum edge cuts.

A *cut* $(S, T)$ is a partitioning of $V$ into two sets $S$ and $T = V \setminus S$. Furthermore, a cut is called minimum if its *cut set* $C = \{\{u, v\} \in E \mid u \in S, v \in T\}$ has minimal cardinality. However, in practice, finding minimum cuts often yields trivial cuts with either $S$ or $T$ only consisting of a single vertex with minimum degree. Thus, we are interested in finding *$s$-$t$-cuts*, i. e., cuts where $S$ and $T$ contain specific vertices $s, t \in V$. Finding these cuts can be done efficiently in practice, e. g., using a preflow push algorithm [17]. However, selecting the vertices $s$ and $t$ to ensure reasonably balanced cuts can be tricky. Natural choices include random vertices, as well as vertices that are far apart in terms of their shortest path distance. However, our preliminary experiments indicate that selecting random vertices of maximum degree for $s$ and $t$ seems to produce the best results. Finally, to derive a vertex separator from a cut, one can compute an MVC on the bipartite graph induced by the cut set, e. g., using the Hopcroft-Karp algorithm [22]. This separator can then be used to select branching vertices. In particular, we continuously branch on vertices from the separator.

Overall, our second strategy works similar to the first one: We maintain a set of possible branching vertices that were selected by computing a minimum $s$-$t$-cut and turning it into a vertex separator. Vertices that were removed by data reduction are discarded from this set and once it is empty a new cut computation is started. However, in contrast to the first strategy, finding a set of suitable branching vertices is much more likely. In order to avoid separators that contain too many vertices, and thus would require too many branching steps to disconnect the graph, we only keep those that do not exceed a certain size and balance threshold. The specific values for these threshold are presented in Section 6.2. Finally, if no suitable separator is found, we use the default branching strategy. Furthermore, in this case we do not try to find a new separator for a fixed number of branching steps as finding one is both unlikely and costly.

### 4.3 Nested Dissection

Both of our previous strategies dynamically maintain a set of branching vertices. Even though this comes at the advantage that most of the computed vertices remain viable candidates for some branching steps, it introduces a noticeable overhead. To alleviate this, our last strategy uses a static ordering of possible branching vertices that is computed once at the beginning of the algorithm. For this purpose we make use of a *nested dissection ordering* [16].

A nested dissection ordering of the vertices of a graph $G$ is obtained by recursively computing balanced bipartitions $(A, B)$ and a vertex separator $S$, that separates $A$ and $B$. The actual ordering is then given by concatenating the orderings of $A$ and $B$ followed by the vertices of $S$. Thus, if we select branching vertices based on the reverse of a nested dissection ordering, we continuously branch on vertices that disconnect the graph into balanced partitions. We compute such an ordering once, after finishing the initial data reduction phase.

There are two main optimizations that we use when considering the nested dissection ordering. First, we limit the number of recursive calls during the nested dissection computation, because we noticed that vertices at the end of the ordering seldom lead to a decomposition of the graph. This is due to the graph structure being changed by data reduction which can lead to separators becoming invalid. Furthermore, similar to the edge-cut-based strategy, we limit the size of separators considered during branching using a threshold. Again, this is done to ensure that we do not require too many branching steps to decompose the graph. The specific value for this size threshold is given in Section 6.2. If any separator in the nested dissection exceeds this threshold, we use the default branching strategy.

## 5 Reduction Branching

Our second approach to selecting good branching vertices is to choose a vertex whose removal will enable the application of new reduction rules. During every reduction step we find a list of candidate vertices to branch on. The following sections will demonstrate how we identify such branching candidate vertices with little computational overhead in practice. To be self contained we will also repeat the reduction rules used here but omit any proofs that can be found in the original publications. Out of the candidates found we then select a vertex of maximum degree. If the degree of all candidate vertices lies below a threshold (defined in Section 6.2) or no candidate vertices were found, we fall back to branching on a vertex of

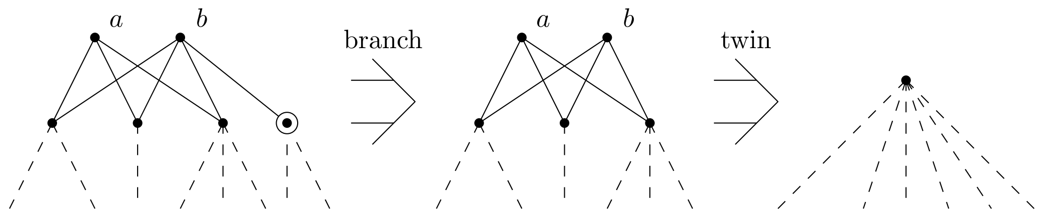


**Figure 2** Vertices $a$ and $b$ are almost twins. After branching on the circled vertex they become twins (in the excluding branch) and can be reduced.

maximum degree. The rational here is that a vertex of large degree changes the structure of the graph more than a vertex of small degree even if that vertex is guaranteed to enable the application of a reduction rule. Also, our current strategies (except the packing-based rule in Section 5.4) only enable the application of the targeted reduction rule in the branch that excludes the vertex from the independent set, the *excluding branch*. However, in the case that includes it into the independent set (*including branch*) all neighbors are removed from the graph as well because they already have an adjacent vertex in the solution. Thus, in both branches multiple vertices are removed.

We also performed some preliminary experiments with storing the candidate vertices in a priority queue without resetting after every branch. However, changes were too frequent for this approach to be faster because of the high amount of priority queue operations.

## 5.1 Almost Twins

The first reduction we target is the *twin* reduction by Xiao and Nagamochi [38]:

▶ **Definition 1.** *(Twins [38]) In a graph $G = (V, E)$ two vertices $u$ and $v$ are called twins if $N(u) = N(v)$ and $d(u) = d(v) = 3$.*

▶ **Theorem 2.** *(Twin Reduction [38]) In a graph $G = (V, E)$ let vertices $u$ and $v$ be twins. If there is an edge among $N(u)$, then there is always an MIS that includes $\{u, v\}$ and therefore excludes $N(u)$. Otherwise, let $G' = (V', E')$ be the graph with $V' = (V \setminus N[\{u, v\}]) \cup \{w\}$ where $w \notin V$ and $E' = (E \cap \binom{V'}{2}) \cup \{\{w, x\} \mid x \in N^2(u)\}$ and let $I'$ be an MIS in $G'$. Then,* $I = \begin{cases} I' \cup \{u, v\} & \text{, if } w \notin I' \\ (I' \setminus \{w\}) \cup N(u) & \text{, else} \end{cases}$ *is an MIS in $G$.*

We now define *almost twins* as follows:

▶ **Definition 3.** *(Almost Twins) In a graph $G = (V, E)$ two non adjacent vertices $u$ and $v$ are called almost twins if $d(u) = 4$, $d(v) = 3$ and $N(v) \subseteq N(u)$ (i. e., $N(u) = N(v) \cup \{w\}$).*

Clearly, after removing $w$, $u$ and $v$ are twins so we can apply the twin reduction. Finding almost twins can be done while searching for twins: The original algorithm checks for each vertex $v$ of degree-3 whether there is a vertex $u \in N^2(v)$ with $d(u) = 3$ and $N(u) = N(v)$. We augment this algorithm by simultaneously also searching for $u \in N^2(v)$ with $d(u) = 4$ and $N(v) \subset N(u)$. This induces about the same computational cost for degree-4 vertices in $N^2(v)$ as for degree 3 vertices. While there might be instances where this causes high overhead, we expect the practical slowdown to be small. Figure 2 illustrates branching for almost twins.

## 5.2 Almost Funnels

Next, we consider the *funnel* reduction which is a special case of the *alternative* reduction by Xiao and Nagamochi [38]:

▶ **Definition 4.** *(Alternative Sets [38]) In a graph $G = (V, E)$ two non empty, disjoint subsets $A, B \subseteq V$ are called alternatives if $|A| = |B|$ and there is an MIS $I$ in $G$ such that $I \cap (A \cup B)$ is either $A$ or $B$.*

▶ **Theorem 5.** *(Alternative Reduction [38]) In a graph $G = (V, E)$ let $A$ and $B$ be alternative sets. Let $G' = (V', E')$ the graph with $V' = V \setminus (A \cup B \cup (N(A) \cap N(B)))$ and $E' = \{\{u, v\} \in E \mid u, v \in V'\} \cup \{\{u, v\} \mid u \in N(A) \setminus N[B], v \in N(B) \setminus N[A]\}$ and let $I'$ be an MIS in $G'$. Then, $I = \begin{cases} I' \cup A & \text{, if } (N(A) \setminus N[B]) \cap I' = \emptyset \\ I' \cup B & \text{, else} \end{cases}$ is an MIS in $G$.*

Note that the alternative reduction adds new edges between existing vertices of the graph which might not be beneficial in every case. To counteract this, the algorithm by Akiba and Iwata [1] only uses special cases, one of which is the funnel reduction:

▶ **Definition 6.** *(Funnel [38]) In a graph $G = (V, E)$ two adjacent vertices $u$ and $v$ are called funnels if $G_{N(v)\setminus\{u\}}$ is a complete graph, i.e, if $N(v) \setminus \{u\}$ is a clique.*

▶ **Theorem 7.** *(Funnel Reduction [38]) In a graph $G = (V, E)$ let $u$ and $v$ be funnels. Then, $\{u\}$ and $\{v\}$ are alternative sets.*

Again, we define a structure that is covered by the funnel reduction after removal of a single vertex:

▶ **Definition 8.** *(Almost Funnel) In a graph $G = (V, E)$ two adjacent vertices $u$ and $v$ are called almost funnels if $u$ and $v$ are not funnels and there is a vertex $w$ such that $N(v) \setminus \{u, w\}$ induces a clique.*

By removing $w$, $u$ and $v$ become funnels. The original funnel algorithm checks whether $u$ and $v$ are funnels by iterating over the vertices in $N(v) \setminus \{u\}$ and checking whether they are adjacent to *all* previous vertices. Once a vertex is found that is not adjacent to all previous vertices, the algorithm concludes that $u$ and $v$ are not funnels and terminates. We augment this algorithm by not immediately terminating in this case. Instead, we consider the following two cases: Either the current vertex $w$ is not adjacent to at least two of the previous vertices. In this case, we can check whether $N(v) \setminus \{u, w\}$ induces a clique. In the second case, $w$ is adjacent to all but one previous vertex $w'$. In this case, both $w$ and $w'$ might be candidate branching vertices. Thus, we check whether $N(v) \setminus \{u, w\}$ or $N(v) \setminus \{u, w'\}$ induce a clique. This adds up to two additional clique checks (of slightly smaller size) to the one clique check in the original algorithm.

## 5.3 Almost Unconfined

The core idea of the *unconfined* reduction by Xiao and Nagamochi [38] is to detect vertices not required for an MIS that can therefore be removed from the graph by algorithmically contradicting the assumption that every MIS contains the vertex.

▶ **Definition 9.** *(Child, Parent [38]) In a graph $G = (V, E)$ with an independent set $I$, a vertex $v$ is called a child of $I$ if $|N(v) \cap I| = 1$ and the unique neighbor of $v$ in $I$ is called the parent of $v$.*

**Algorithm 1** Unconfined – Xiao and Nagamochi [38].

```
Input: A graph $G$, a vertex $v$
Unconfined($G$, $v$) begin
    $S \leftarrow \{v\}$
    while $S$ has child $u$ with $|N(u) \setminus N[S]| \leq 1$ do
        if $|N(u) \setminus N[S]| = 0$ then
            return true
        else
            $\{w\} \leftarrow N(u) \setminus N[v]$        // by assumption $w$ also has to
            $S \leftarrow S \cup \{w\}$                   // be contained in every MIS
    return false
Output: true if $v$ is unconfined, false otherwise
```

Algorithm 1 shows the algorithm used by Akiba and Iwata [1] to detect so called *unconfined* vertices.

▶ **Theorem 10.** *(Unconfined Reduction [38]) In a graph $G = (V, E)$, if Algorithm 1 returns true for an unconfined vertex $v$, then there is always an MIS that does not contain $v$.*

Again, we define a vertex to be almost unconfined:

▶ **Definition 11.** *(Almost Unconfined) In a graph $G = (V, E)$ a vertex $v$ is called almost unconfined if $v$ is not unconfined but there is a vertex $w$ such that $v$ is unconfined in $G - \{w\}$.*

Here, we only present an augmentation that detects *some* almost unconfined vertices. In particular, if at any point during the algorithm there is only *one* extending child, i.e. a child $u$ of $S$ with $N(u) \setminus N[S] = \{w\}$, then removal of $w$ makes $v$ unconfined. During Algorithm 1 we collect all these vertices $w$ and add them to the set of candidate branching vertices if the algorithm cannot already remove $v$. This only adds the overhead of temporarily storing the potential candidates and adding them to the actual candidate list if $v$ is not removed.

## 5.4 Almost Packing

The core idea behind the packing rule by Akiba and Iwata [1] is that when the exluding branch of a vertex $v$ is selected, one can assume that no maximum independent set contains $v$. Otherwise, if there is a maximum independent set that contains $v$, the algorithm finds it in the including branch of $v$. Based on the assumption that no maximum independent set includes a vertex $v$, constraints for the remaining vertices can be derived. For example, a maximum independent set that does not contain $v$ has to include at least two neighbors of $v$. The corresponding constraint is $\sum_{u \in N(v)} x_u \geq 2$, where $x_u$ is a binary variable that indicates whether a vertex is included in the current solution. Otherwise, we will find a solution of the same size in the branch including $v$. The algorithm creates such constraints when branching or reducing, and updates them accordingly during the data reductions and branching steps. When a vertex $v$ is eliminated from the graph, $x_v$ gets removed from all constraints. If $v$ is included into the current solution, the corresponding right sides are also decreased by one.

A constraint $\sum_{v \in S \subset V} x_v \geq k$ can be utilized in two reductions. Firstly, if $k$ is equal to the number of variables $|S|$, all vertices from $S$ have to be included into the current solution. If there are edges between vertices from $S$, then no valid solution can include all vertices from $S$, so the branch is pruned. Secondly, if there is a vertex $v$ such that $|S| - |N(v) \cap S| < k$, then $v$ has to be excluded from the current solution. If $k > |S|$, the constraint can not be fulfilled and the current branch is pruned.

In our branching strategy we target both reductions. If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$, where $|S| = k + 1$, excluding any vertex of $S$ from the solution or including a vertex of $S$ that has one neighbor in $S$ enables the first reduction. Thus, we consider all vertices in $S$ for branching. Note that including a vertex from $S$ that has more than one neighbor in $S$ makes the constraint unfulfillable and the branch is pruned.

If there is a constraint $\sum_{v \in S \subset V} x_v \geq k$ and a vertex $v$, such that $k = |S| - |N(v) \cap S|$, excluding any vertex of $S \setminus N(v)$ from the solution or including a vertex of $S \setminus N(v)$ that has at least one neighbor in $S \setminus N(v)$ enables the second reduction. Thus, we consider all vertices in $S \setminus N(v)$ for branching.

Note that in contrast to our previous reduction-based branching rules, packing reductions can also be applied in the including branch in many cases.

Detecting these branching candidates can be done with small constant overhead whilst performing the packing reduction.

## 6 Experimental Evaluation

In this section, we present the results of our experimental evaluation. Tables and figures here show aggregated results. For detailed results for all of our algorithms across all instances, see Appendix A.

### 6.1 Experimental Environment

We augment a C++-adaptation of the algorithm by Akiba and Iwata [1] with our branching strategies and compile it with g++ 9.3.0 using full optimizations (`-O3`). Our code is publicly available on GitHub[1]. We execute all our experiments on a machine with 4 8-core Intel Xeon E5-4640 CPUs (2.4 GHz) and 512 GiB DDR3-PC1600 RAM running Ubuntu 20.04.1 with Linux Kernel 5.4.0-64. To speed up our experiments we use two identical machines and run at most 8 instances at once on the same machine (using the same machine for all algorithms on a specific instance). All numbers reported are arithmetic means of three runs with a timeout of ten hours.

### 6.2 Algorithm Configuration

We use a C++ adaptation of the implementation by Akiba and Iwata [1] in its default configuration as a basis for our algorithm. During preliminary experiments we found suitable values for the parameters of our techniques. These experiments were run on a subset of our total instance set. We use the geometric mean over all instances of the speedup over the default branching strategy as a basis for the following decisions: for the technique based on edge cuts, we only use cuts that contain at most 25 vertices and where the smaller side of the cut contains at least ten percent of the remaining vertices. If no suitable separator is found, we skip ten branching steps. For computing nested dissections, we use InertialFlowCutter [18] with the KaFFPa [34] backend. The KaFFPa partitioner is configured to use the *strong* preset with a fixed seed of 42. For branching, we use three levels of nested dissections with a minimum balance of at least 40% of the vertices in the smaller part of each dissection. Furthermore, we only use the nested dissection if separators contain at most 50 vertices. For the reduction-based branching rules, we fall back to the default branching strategy if all

[1] `https://github.com/Hespian/CutBranching`

candidates have a degree of less than $\Delta - k$. In the case of twin-, funnel- and unconfined-reduction-based branching strategies we choose $k$ as 2. For the packing-reduction-based branching rule, $k$ is set to 5 and for the combined branching rule, $k$ is set to 4.

## 6.3 Instances

We use instances from several sources: The "easy" instances used for the PACE 2019 Challenge on Minimum Vertex Cover [12]. Complements of Maximum Clique instances from the second DIMACS Implementation Challenge [23] and sparse instances from the Stanford Network Analysis Project (SNAP) [25], the 9th DIMACS Implementation Challenge on Shortest Paths [10] and the Network Data Repository [32]. Detailed instance information can be found in Table 1. Directed instances were converted into undirected graphs by ignoring the direction of edges and removing duplicates. Our original set of instances contained the first 80 PACE instances, 53 DIMACS instances and 34 sparse networks. From these instances, we excluded all instances that (1) required no branches, (2) on which all techniques had a running time of less than 0.1 seconds, or (3) on which no technique was able to find a solution within 10 hours. The remaining set of instances is composed of 48 PACE instances, 37 DIMACS instances and 16 sparse networks.

**Table 1** Number of vertices $|V|$ and edges $|E|$ for each graph.

| PACE [12] instances: | | |
|---|---|---|
| Graph | $\|V\|$ | $\|E\|$ |
| 05 | 200 | 798 |
| 06 | 200 | 733 |
| 10 | 199 | 758 |
| 16 | 153 | 802 |
| 19 | 200 | 862 |
| 31 | 200 | 813 |
| 33 | 4,410 | 6,885 |
| 35 | 200 | 864 |
| 36 | 26,300 | 41,500 |
| 37 | 198 | 808 |
| 38 | 786 | 14,024 |
| 39 | 6,795 | 10,620 |
| 40 | 210 | 625 |
| 41 | 200 | 1,023 |
| 42 | 200 | 952 |
| 43 | 200 | 841 |
| 44 | 200 | 1,147 |
| 45 | 200 | 1,020 |
| 46 | 200 | 812 |
| 47 | 200 | 1,093 |
| 48 | 200 | 1,025 |
| 49 | 200 | 933 |
| 50 | 200 | 1,025 |
| 51 | 200 | 1,098 |
| 52 | 200 | 992 |
| 53 | 200 | 1,026 |
| 54 | 200 | 961 |
| 55 | 200 | 938 |
| 56 | 200 | 1,089 |
| 57 | 200 | 1,160 |
| 58 | 200 | 1,171 |
| 59 | 200 | 961 |
| 60 | 200 | 1,118 |
| 61 | 200 | 931 |
| 62 | 199 | 1,128 |
| 63 | 200 | 1,011 |
| 64 | 200 | 1,042 |
| 65 | 200 | 1,011 |
| 66 | 200 | 866 |
| 67 | 200 | 1,174 |
| 68 | 200 | 961 |
| 69 | 200 | 1,083 |
| 70 | 200 | 860 |
| 71 | 200 | 952 |
| 72 | 200 | 1,167 |
| 73 | 200 | 1,078 |
| 74 | 200 | 805 |
| 77 | 200 | 961 |

| DIMACS [23] instances: | | |
|---|---|---|
| Graph | $\|V\|$ | $\|E\|$ |
| C125.9 | 125 | 787 |
| MANN_a27 | 378 | 702 |
| MANN_a45 | 1,035 | 1,980 |
| brock200_1 | 200 | 5,066 |
| brock200_2 | 200 | 10,024 |
| brock200_3 | 200 | 7,852 |
| brock200_4 | 200 | 6,811 |
| gen200_p0.9_44 | 200 | 1,990 |
| gen200_p0.9_55 | 200 | 1,990 |
| hamming8-4 | 256 | 11,776 |
| johnson16-2-4 | 120 | 1,680 |
| keller4 | 171 | 5,100 |
| p_hat1000-1 | 1,000 | 377,247 |
| p_hat1000-2 | 1,000 | 254,701 |
| p_hat1500-1 | 1,500 | 839,327 |
| p_hat300-1 | 300 | 33,917 |
| p_hat300-2 | 300 | 22,922 |
| p_hat300-3 | 300 | 11,460 |
| p_hat500-1 | 500 | 93,181 |
| p_hat500-2 | 500 | 61,804 |
| p_hat500-3 | 500 | 30,950 |
| p_hat700-1 | 700 | 183,651 |
| p_hat700-2 | 700 | 122,922 |
| san1000 | 1,000 | 249,000 |
| san200_0.7_1 | 200 | 5,970 |
| san200_0.7_2 | 200 | 5,970 |
| san200_0.9_1 | 200 | 1,990 |
| san200_0.9_2 | 200 | 1,990 |
| san200_0.9_3 | 200 | 1,990 |
| san400_0.5_1 | 400 | 39,900 |
| san400_0.7_1 | 400 | 23,940 |
| san400_0.7_2 | 400 | 23,940 |
| san400_0.7_3 | 400 | 23,940 |
| sanr200_0.7 | 200 | 6,032 |
| sanr200_0.9 | 200 | 2,037 |
| sanr400_0.5 | 400 | 39,816 |
| sanr400_0.7 | 400 | 23,931 |

| Sparse networks: | | | |
|---|---|---|---|
| Graph | $\|V\|$ | $\|E\|$ | source |
| as-skitter | 1,696,415 | 11,095,298 | [25] |
| baidu-relatedpages | 415,641 | 2,374,044 | [32] |
| bay | 321,270 | 397,415 | [10] |
| col | 435,666 | 521,200 | [10] |
| fla | 1,070,376 | 1,343,951 | [10] |
| hudong-internallink | 1,984,484 | 14,428,382 | [32] |
| in-2004 | 1,382,870 | 13,591,473 | [32] |
| libimseti | 220,970 | 17,233,144 | [32] |
| musae-twitch_DE | 9,498 | 153,138 | [25] |
| musae-twitch_FR | 6,549 | 112,666 | [25] |
| petster-fs-dog | 426,820 | 8,543,549 | [32] |
| soc-LiveJournal1 | 4,847,571 | 42,851,237 | [25] |
| web-BerkStan | 685,230 | 6,649,470 | [25] |
| web-Google | 875,713 | 4,322,051 | [25] |
| web-NotreDame | 325,730 | 1,090,108 | [25] |
| web-Stanford | 281,903 | 1,992,636 | [25] |

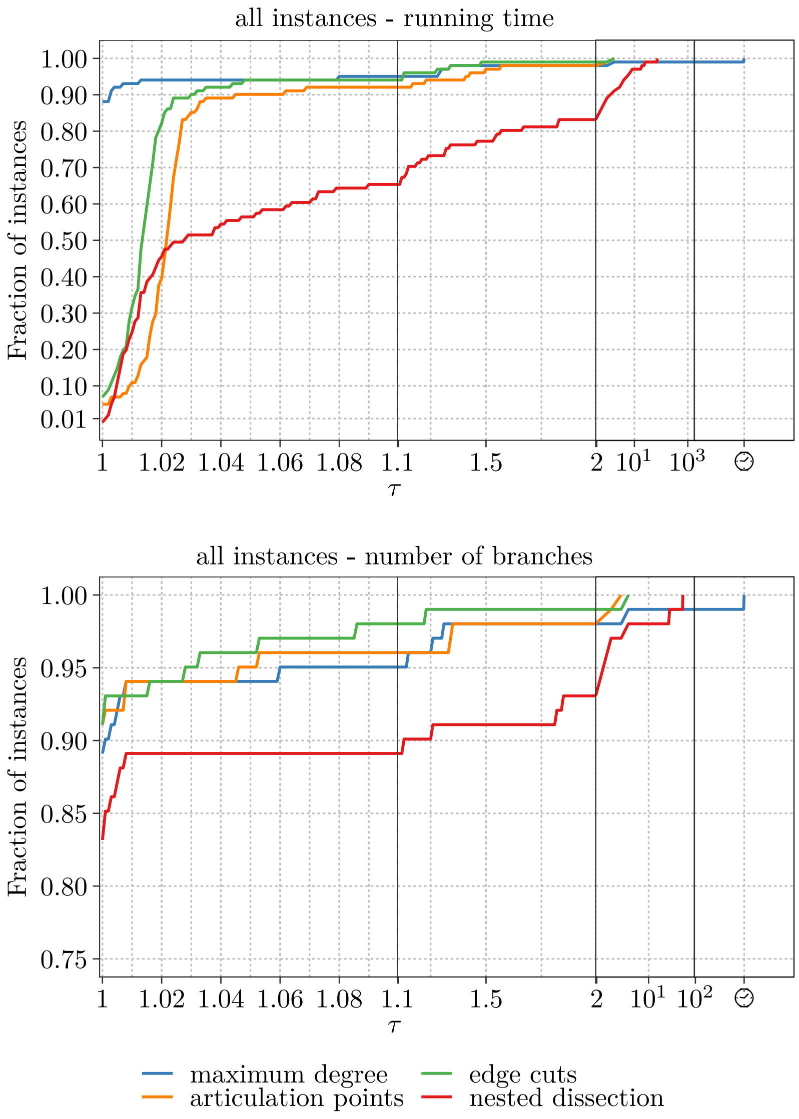


**Figure 3** Performance plots for decomposition-based branching strategies.

**Table 2** Speedup of our decomposition-based techniques over maximum degree branching.

| | PACE | DIMACS | Sparse net. | All Instances |
|---|---|---|---|---|
| articulation points | 0.99 | 0.99 | 2.17 | 1.20 |
| edge cuts | 1.00 | 0.99 | **2.29** | **1.22** |
| nested dissections | 1.00 | 0.99 | 2.15 | 1.21 |

## 6.4 Decomposition Branching

Figure 3 shows performance profiles [11] of the running time and number of branches of our decomposition-based branching strategies: Let $\mathcal{T}$ be the set of all techniques we want to compare, $\mathcal{I}$ the set of instances, and $t_T(I)$ the running time/number of branches of technique $T \in \mathcal{T}$ on instance $I \in \mathcal{I}$. The y-axis shows for each technique $T$ the fraction of instances for which $t_T(I) \leq \tau \cdot \min_{T' \in \mathcal{T}} t_{T'}(I)$, where $\tau$ is shown on the x-axis. For $\tau = 1$, the y-axis shows the fraction of instances on which a technique performs best. Note that these plots compare the performance of a technique relative to the best performing technique and do not show a ranking of all techniques. Instances that were not finished by a technique within the time limit are marked with ⏲.

The running time plot in Figure 3 shows that for most instances, the default strategy of branching on a vertex of maximum degree outperforms our decomposition-based approaches. However, for instances that have suitable candidates for decomposition, such as sparse networks, significant speedups compared to the default strategy can be seen. To be more specific, assigning a time of ten hours (our timeout threshold) for unfinished instances, we achieve a total speedup[2] of 2.15 to 2.29 over maximum degree branching for our decomposition-based techniques on sparse networks (see Table 2). In particular, there is one instance (web-stanford) that causes a timeout with the default strategy but can be solved in 8 (articulation points) to 43 (nested dissections) seconds using a decomposition-based approach. Table 2 shows that overall, our technique using edge cuts seems to be the most beneficial, achieving an overall speedup of 22% over maximum degree. Finally, Figure 3 shows that most running times are only slightly slower than the default strategy with a few instances showing a speedup. This is mainly because the number of branches required to solve the instances does not change in most cases and most of the running time difference is caused by the overhead from searching for branching vertices.

## 6.5 Reduction Branching

Figure 4 shows the performance profiles (see Section 6.4) for our reduction-based branching strategies. Here we see that targeting the packing reduction results in the fastest time for the most number of instances. In fact, targeting the packing reduction performs better than maximum degree branching on all but 3 PACE instances, achieving a speedup of 34% (Table 3) on these instances. On the DIMACS instances, performance is closer to that of maximum degree with an overall speedup of 4%. On sparse networks, packing is only faster than maximum degree branching on 6 out of 16 instances but still achieves an overall speedup of 31% due to being considerably faster on some of the longer running instances. The performance of our packing-based technique might be explained by it's property of enabling a reduction in both the including and the excluding branch, while our other reduction-based techniques only enable a reduction in the excluding branch. Our funnel-based technique is faster than maximum degree branching for all but 4 of the PACE instances, resulting in a speedup of 14% on these instances but only a 2% speedup over all instances due to slightly slower running times on the other instance classes. We also show results for a strategy that targets all reduction rules described in Section 5 (called *combined*). Even though this approach leads to the second lowest number of branches for most instances, the time required to identify candidate vertices for all reduction rules causes too big of an overhead to be competitive. In fact, preliminary experiments showed that the number of branches is still small for a technique that only combines twin-, funnel- and unconfined-based branching. Optimizing the algorithms to identify candidate vertices could lead to making this combined strategy competitive.

[2] calculated by dividing the running times to solve all instances for two algorithms, excluding instances unsolved by both algorithms

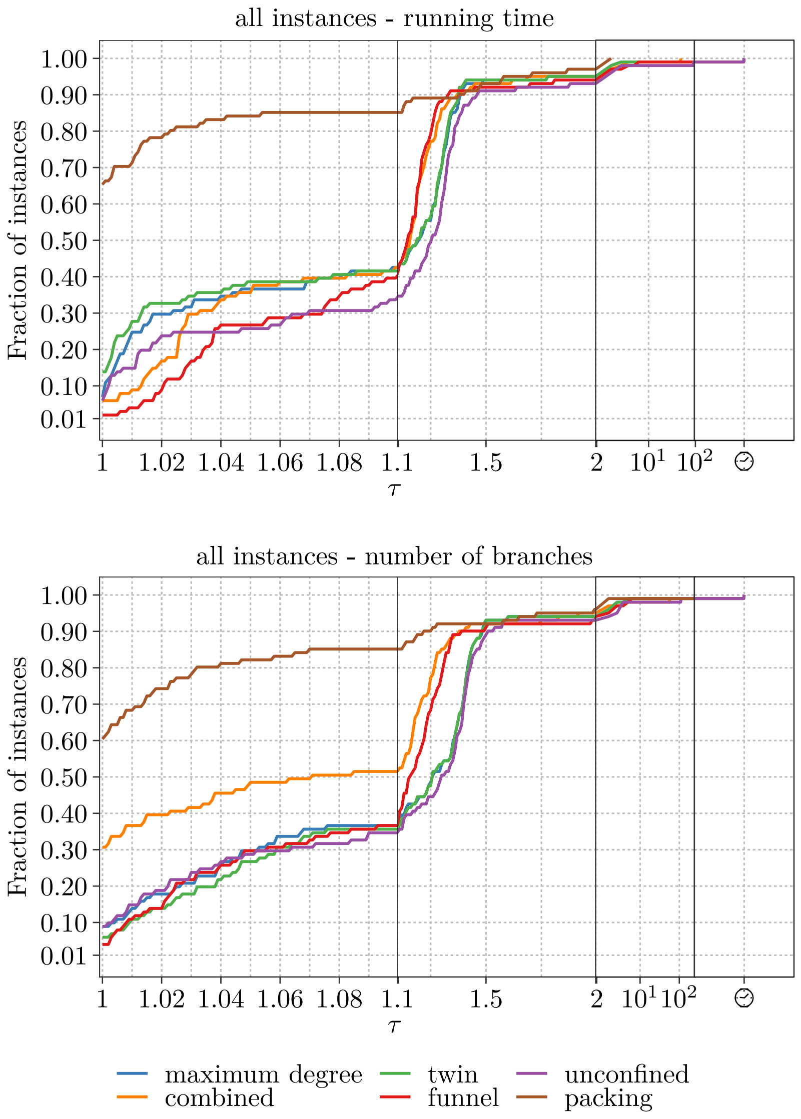


**Figure 4** Performance plots for reduction-based branching strategies.

**Table 3** Speedup of our reduction-based techniques over maximum degree branching.

| | PACE | DIMACS | Sparse net. | All Instances |
|---|---|---|---|---|
| Twin | 1.00 | 1.00 | 0.97 | 0.99 |
| Funnel | 1.14 | 0.99 | 0.98 | 1.02 |
| Unconfined | 0.79 | 1.00 | 0.86 | 0.92 |
| Packing | **1.34** | **1.04** | **1.31** | **1.16** |
| Combined | 1.14 | 1.03 | 1.30 | 1.12 |

## 7 Conclusion and Future Work

In this work we presented several novel branching strategies for the maximum independent set problem. Our strategies either follow a decomposition-based or reduction-rule-based approach. The decomposition-based strategies make use of increasingly sophisticated methods of finding vertices that are likely to decompose the graph into two or more connected components.

Even though these strategies often come with a non negligible overhead, they work well for graphs that have a suitable structure, such as social networks. For instances that still favor the default branching strategy of branching on the vertex of highest degree, our reduction-rule-based strategies provide a smaller but more consistent speedup. These rules aim to facilitate the application of reduction rules which leads to smaller graphs that can be solved more quickly.

Overall, using one of our proposed strategies allows us to find the optimal solution the fastest for most instances tested. However, deciding which particular strategy to use for a given instance still remains an open problem. Finding suitable graph characteristics to do so provides an interesting opportunity for future work. Furthermore, our experimental evaluation on a combined approach that tries to use all reduction-rule-based strategies at the same time achieves a smaller number of branches than the default strategy for a large set of instances. However, the running time of this approach still suffers from frequent checks whether a particular vertex is a potential branching vertex. A more sophisticated and incremental way of tracking when a vertex becomes a branching vertex might provide significant performance benefits. In turn, this might lead to a branching strategy that consistently outperforms branching on the vertex of highest degree independent of the instance type.

## 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. `doi:10.1016/j.tcs.2015.09.023`.

2 Maram Alsahafy and Lijun Chang. Computing maximum independent sets over large sparse graphs. In *International Conference on Web Information Systems Engineering*, pages 711–727. Springer, 2020.

3 Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and Johan M. M. van Rooij. Fast algorithms for max independent set. *Algorithmica*, 62(1-2):382–415, 2012. `doi:10.1007/s00453-010-9460-7`.

4 Sergiy Butenko and Wilbert E Wilhelm. Clique-detection models in computational biochemistry and genomics. *European Journal of Operational Research*, 173(1):1–17, 2006.

5 Randy Carraghan and Panos M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990. `doi:10.1016/0167-6377(90)90057-C`.

6 Lijun Chang, Wei Li, and Wenjie Zhang. Computing A near-maximum independent set in linear time by reducing-peeling. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1181–1196. ACM, 2017. `doi:10.1145/3035918.3035939`.

7 Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40-42):3736–3756, 2010. `doi:10.1016/j.tcs.2010.06.026`.

8 Tammy M. K. Cheng, Yu-En Lu, Michele Vendruscolo, Pietro Liò, and Tom L. Blundell. Prediction by graph theoretic measures of structural effects in proteins arising from non-synonymous single nucleotide polymorphisms. *PLoS Computational Biology*, 4(7), 2008. `doi:10.1371/journal.pcbi.1000135`.

9 Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F Werneck. Accelerating local search for the maximum independent set problem. In *International symposium on experimental algorithms*, pages 118–133. Springer, 2016.

10 Camil Demetrescu, Andrew V Goldberg, and David S Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc., 2009.

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

12 M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration. In *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*, volume 148, pages 25:1–25:23, 2019. doi:10.4230/LIPIcs.IPEC.2019.25.

13 Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *J. ACM*, 56(5):25:1–25:32, 2009. doi:10.1145/1552285.1552286.

14 Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. An exact algorithm for maximum k-plexes in massive graphs. In *IJCAI*, pages 1449–1455, 2018.

15 M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proceedings of the 6th ACM Symposium on Theory of Computing*, STOC '74, pages 47–63. ACM, 1974.

16 Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

17 Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.

18 Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):196, 2019.

19 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*, pages 1–11. SIAM, 2020. doi:10.1137/1.9781611976229.1.

20 Demian Hespe, Christian Schulz, and Darren Strash. Scalable kernelization for maximum independent sets. *Journal of Experimental Algorithmics (JEA)*, 24(1):1–22, 2019.

21 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

22 John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

23 David S Johnson. Cliques, coloring, and satisfiability: second dimacs implementation challenge. *DIMACS series in discrete mathematics and theoretical computer science*, 26:11–13, 1993.

24 Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In *Experimental Algorithms, 9th International Symposium*, volume 6049, pages 83–93. Springer, 2010. doi:10.1007/978-3-642-13193-6_8.

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

26 Chu-Min Li, Zhiwen Fang, and Ke Xu. Combining maxsat reasoning and incremental upper bound for the maximum clique problem. In *25th IEEE International Conference on Tools with Artificial Intelligence*, pages 939–946. IEEE Computer Society, 2013. doi:10.1109/ICTAI.2013.143.

27 Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 2017. doi:10.1016/j.cor.2017.02.017.

28 Chu-Min Li, Hua Jiang, and Ruchu Xu. Incremental maxsat reasoning to reduce branches in a branch-and-bound algorithm for maxclique. In *Learning and Intelligent Optimization - 9th International Conference*, volume 8994, pages 268–274. Springer, 2015. doi:10.1007/978-3-319-19084-6_26.

29 Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1611.

30 Rick Plachetta and Alexander van der Grinten. Sat-and-reduce for vertex cover: Accelerating branch-and-reduce by sat solving. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 169–180. SIAM, 2021.

31 Deepak Puthal, Surya Nepal, Cécile Paris, Rajiv Ranjan, and Jinjun Chen. Efficient algorithms for social network coverage and reach. In *IEEE International Congress on Big Data*, pages 467–474. IEEE Computer Society, 2015. doi:10.1109/BigDataCongress.2015.75.
32 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.
33 Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 27(5):144, 2008. doi:10.1145/1409060.1409097.
34 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer, 2013.
35 Christian Schorr. Improved branching strategies for maximum independent sets. Master's thesis, Karlsruhe Institute of Technology, 2020.
36 Pablo San Segundo and Cristóbal Tapia. Relaxed approximate coloring in exact maximum clique search. *Computers & Operations Research*, 44:185–192, 2014. doi:10.1016/j.cor.2013.10.018.
37 Etsuji Tomita, Yoichi Sutani, Takanori Higashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. *IEICE Trans. Inf. Syst.*, 96-D(6):1286–1298, 2013. doi:10.1587/transinf.E96.D.1286.
38 Mingyu Xiao and Hiroshi Nagamochi. Confining sets and avoiding bottleneck cases: A simple maximum independent set algorithm in degree-3 graphs. *Theoretical Computer Science*, 469:92–104, 2013. doi:10.1016/j.tcs.2012.09.022.
39 Mingyu Xiao and Hiroshi Nagamochi. Exact algorithms for maximum independent set. *Inf. Comput.*, 255:126–146, 2017. doi:10.1016/j.ic.2017.06.001.

## A Detailed Experimental Results

We now present detailed results of our experimental evaluation. Detailed tables show running times $t$ (in seconds) and speedup $s$. Speedups are computed by dividing the running time of maximum degree branching by the running time of the respective technique. Timeouts are assigned a running time of ten hours. Note, that this is the same as our time limit. We also present the aggregated speedup $s_{\text{total}}$ computed by dividing the running time of both algorithms over all instances (omitting instances were both algorithms do not finish within our time limit). A value is highlighted in bold if it is the best one within a row.

**Table 4** Detailed results for our decomposition-based strategies on the PACE instances.

| Graph | max. deg. | articulation | edge cuts | nested dis. |
|---|---:|---:|---:|---:|
| PACE | t | t (s) | t (s) | t (s) |
| 05 | **1.97** | 2.00 (0.98) | 2.00 (0.99) | 2.44 (0.81) |
| 06 | **0.85** | 0.87 (0.98) | 0.87 (0.98) | 1.33 (0.64) |
| 10 | **2.24** | 2.27 (0.99) | 2.26 (0.99) | 2.66 (0.84) |
| 16 | 25,836.77 | 26,175.23 (0.99) | **25,763.30 (1.00)** | 25,865.40 (1.00) |
| 19 | **3.17** | 3.22 (0.98) | 3.18 (0.99) | 3.63 (0.87) |
| 31 | **74.37** | 76.03 (0.98) | 75.45 (0.99) | 74.82 (0.99) |
| 33 | **1.01** | 1.03 (0.98) | 1.02 (0.99) | 40.09 (0.03) |
| 35 | **7.64** | 7.84 (0.97) | 7.77 (0.98) | 8.13 (0.94) |
| 36 | **1.84** | 1.87 (0.98) | 1.85 (0.99) | 3.93 (0.47) |
| 37 | **10.27** | 10.48 (0.98) | 10.47 (0.98) | 10.75 (0.96) |
| 38 | 12.33 | 11.24 (1.10) | **3.25 (3.79)** | 15.35 (0.80) |
| 39 | **93.79** | 96.82 (0.97) | 95.96 (0.98) | 95.21 (0.99) |
| 40 | **4,690.64** | 4,794.37 (0.98) | 4,758.15 (0.99) | 4,712.57 (1.00) |
| 41 | **48.56** | 49.84 (0.97) | 49.39 (0.98) | 49.35 (0.98) |
| 42 | **37.32** | 38.11 (0.98) | 37.91 (0.98) | 37.87 (0.99) |
| 43 | **175.11** | 178.81 (0.98) | 177.26 (0.99) | 175.24 (1.00) |
| 44 | **92.90** | 95.13 (0.98) | 94.28 (0.99) | 93.40 (0.99) |
| 45 | **25.41** | 26.01 (0.98) | 25.73 (0.99) | 25.90 (0.98) |
| 46 | **109.55** | 111.95 (0.98) | 111.00 (0.99) | 110.22 (0.99) |
| 47 | **58.47** | 59.70 (0.98) | 59.38 (0.98) | 59.22 (0.99) |
| 48 | **25.28** | 25.80 (0.98) | 25.60 (0.99) | 25.80 (0.98) |
| 49 | **17.80** | 18.19 (0.98) | 18.10 (0.98) | 18.30 (0.97) |
| 50 | **48.87** | 50.01 (0.98) | 49.56 (0.99) | 49.40 (0.99) |
| 51 | **56.70** | 58.00 (0.98) | 57.63 (0.98) | 57.52 (0.99) |
| 52 | **22.16** | 22.68 (0.98) | 22.53 (0.98) | 22.69 (0.98) |
| 53 | **59.88** | 61.42 (0.97) | 60.77 (0.99) | 60.42 (0.99) |
| 54 | **32.08** | 32.89 (0.98) | 32.73 (0.98) | 32.67 (0.98) |
| 55 | **6.83** | 6.97 (0.98) | 6.92 (0.99) | 7.32 (0.93) |
| 56 | **97.00** | 99.09 (0.98) | 98.31 (0.99) | 97.80 (0.99) |
| 57 | **66.01** | 67.76 (0.97) | 67.18 (0.98) | 66.83 (0.99) |
| 58 | **48.12** | 48.83 (0.99) | 48.72 (0.99) | 48.63 (0.99) |
| 59 | **13.30** | 13.60 (0.98) | 13.54 (0.98) | 13.80 (0.96) |
| 60 | **79.56** | 81.58 (0.98) | 80.94 (0.98) | 80.23 (0.99) |
| 61 | **21.91** | 22.31 (0.98) | 22.26 (0.98) | 22.36 (0.98) |
| 62 | **66.22** | 68.48 (0.97) | 67.40 (0.98) | 66.80 (0.99) |
| 63 | **69.06** | 70.55 (0.98) | 69.91 (0.99) | 69.35 (1.00) |
| 64 | **29.58** | 30.07 (0.98) | 29.99 (0.99) | 30.09 (0.98) |
| 65 | **36.84** | 37.53 (0.98) | 37.28 (0.99) | 37.29 (0.99) |
| 66 | **8.06** | 8.28 (0.97) | 8.23 (0.98) | 8.63 (0.93) |
| 67 | **122.74** | 124.79 (0.98) | 124.25 (0.99) | 123.38 (0.99) |
| 68 | **8.79** | 8.92 (0.99) | 8.86 (0.99) | 9.24 (0.95) |
| 69 | **43.11** | 44.13 (0.98) | 43.85 (0.98) | 43.63 (0.99) |
| 70 | **11.79** | 12.00 (0.98) | 11.97 (0.99) | 12.25 (0.96) |
| 71 | **36.20** | 36.83 (0.98) | 36.66 (0.99) | 36.64 (0.99) |
| 72 | **46.44** | 47.47 (0.98) | 46.91 (0.99) | 46.86 (0.99) |
| 73 | **43.02** | 44.07 (0.98) | 43.77 (0.98) | 43.65 (0.99) |
| 74 | **7.06** | 7.24 (0.97) | 7.14 (0.99) | 7.49 (0.94) |
| 77 | **13.30** | 13.65 (0.97) | 13.51 (0.98) | 13.79 (0.96) |
| $s_{\text{total}}$ | **1.00** | 0.99 | 1.00 | 1.00 |

**Table 5** Detailed results for our decomposition-based strategies on the DIMACS instances.

| Graph | max. deg. | articulation | edge cuts | nested dis. |
|---|---:|---:|---:|---:|
| DIMACS | t | t (s) | t (s) | t (s) |
| C125.9 | **0.98** | 1.01 (0.97) | 1.00 (0.98) | 1.43 (0.69) |
| MANN_a27 | **0.48** | 0.49 (0.98) | 0.49 (0.98) | 0.98 (0.49) |
| MANN_a45 | **73.80** | 75.24 (0.98) | 74.93 (0.98) | 74.70 (0.99) |
| brock200_1 | **137.34** | 140.20 (0.98) | 137.56 (1.00) | 140.01 (0.98) |
| brock200_2 | **4.59** | 4.69 (0.98) | 4.70 (0.98) | 10.07 (0.46) |
| brock200_3 | 22.06 | 22.33 (0.99) | **21.92 (1.01)** | 26.39 (0.84) |
| brock200_4 | **28.34** | 28.72 (0.99) | 28.35 (1.00) | 32.48 (0.87) |
| gen200_p0.9_44 | **152.61** | 156.30 (0.98) | 154.50 (0.99) | 153.49 (0.99) |
| gen200_p0.9_55 | **131.24** | 134.64 (0.97) | 133.04 (0.99) | 132.58 (0.99) |
| hamming8-4 | **19.29** | 19.65 (0.98) | 19.49 (0.99) | 25.38 (0.76) |
| johnson16-2-4 | **39.87** | 41.17 (0.97) | 40.21 (0.99) | 40.33 (0.99) |
| keller4 | **2.62** | 2.68 (0.98) | 2.65 (0.99) | 4.37 (0.60) |
| p_hat1000-1 | **860.24** | 868.71 (0.99) | 870.04 (0.99) | 906.24 (0.95) |
| p_hat1000-2 | **33,035.45** | 33,656.50 (0.98) | 33,508.10 (0.99) | 33,247.45 (0.99) |
| p_hat1500-1 | **8,935.77** | 9,015.15 (0.99) | 9,015.74 (0.99) | 8,994.28 (0.99) |
| p_hat300-1 | **3.70** | 3.79 (0.98) | 3.82 (0.97) | 23.94 (0.15) |
| p_hat300-2 | **5.53** | 5.66 (0.98) | 5.63 (0.98) | 21.76 (0.25) |
| p_hat300-3 | 189.58 | 191.06 (0.99) | **188.96 (1.00)** | 196.89 (0.96) |
| p_hat500-1 | **38.63** | 39.26 (0.98) | 39.41 (0.98) | 59.29 (0.65) |
| p_hat500-2 | **96.36** | 97.82 (0.99) | 97.58 (0.99) | 107.29 (0.90) |
| p_hat500-3 | **14,860.70** | 14,895.15 (1.00) | 14,979.65 (0.99) | 14,909.35 (1.00) |
| p_hat700-1 | 163.30 | **162.84 (1.00)** | 163.17 (1.00) | 177.34 (0.92) |
| p_hat700-2 | **906.32** | 917.87 (0.99) | 914.96 (0.99) | 917.50 (0.99) |
| san1000 | **895.34** | 902.64 (0.99) | 903.38 (0.99) | 920.28 (0.97) |
| san200_0.7_1 | **10.85** | 11.01 (0.98) | 10.90 (1.00) | 14.45 (0.75) |
| san200_0.7_2 | 0.33 | 0.34 (0.95) | **0.32 (1.01)** | 2.34 (0.14) |
| san200_0.9_1 | **13.93** | 14.37 (0.97) | 14.08 (0.99) | 14.94 (0.93) |
| san200_0.9_2 | **34.15** | 34.77 (0.98) | 34.35 (0.99) | 34.90 (0.98) |
| san200_0.9_3 | **1,069.00** | 1,094.54 (0.98) | 1,078.09 (0.99) | 1,071.31 (1.00) |
| san400_0.5_1 | **9.21** | 9.35 (0.98) | 9.36 (0.98) | 16.76 (0.55) |
| san400_0.7_1 | **1,125.52** | 1,139.20 (0.99) | 1,131.38 (0.99) | 1,130.07 (1.00) |
| san400_0.7_2 | 3,062.38 | **3,053.97 (1.00)** | 3,083.59 (0.99) | 3,073.66 (1.00) |
| san400_0.7_3 | **4,411.82** | 4,464.53 (0.99) | 4,447.19 (0.99) | 4,423.16 (1.00) |
| sanr200_0.7 | **48.35** | 49.51 (0.98) | 48.71 (0.99) | 52.13 (0.93) |
| sanr200_0.9 | **679.25** | 696.41 (0.98) | 688.51 (0.99) | 680.29 (1.00) |
| sanr400_0.5 | **373.40** | 374.20 (1.00) | 374.26 (1.00) | 380.08 (0.98) |
| sanr400_0.7 | **29,766.80** | 30,390.80 (0.98) | 30,270.10 (0.98) | 30,001.55 (0.99) |
| $s_{\text{total}}$ | **1.00** | 0.99 | 0.99 | 0.99 |

**Table 6** Detailed results for our decomposition-based strategies on sparse networks.

| Graph | max. deg. | articulation | edge cuts | nested dis. |
|---|---:|---:|---:|---:|
| Sparse net. | t | t (s) | t (s) | t (s) |
| as-skitter | **2,058.32** | 2,100.57 (0.98) | 2,071.06 (0.99) | 2,068.46 (1.00) |
| baidu-relatedpages | **0.82** | 0.88 (0.94) | 0.86 (0.96) | 7.22 (0.11) |
| bay | 1.68 | 1.87 (0.90) | **1.31 (1.28)** | 23.43 (0.07) |
| col | 5,019.93 | 4,737.48 (1.06) | **3,872.65 (1.30)** | 5,101.46 (0.98) |
| fla | 25.33 | **23.47 (1.08)** | 24.58 (1.03) | 329.42 (0.08) |
| hudong-internallink | **0.99** | 1.55 (0.64) | 1.46 (0.68) | 1.99 (0.50) |
| in-2004 | **5.22** | 5.46 (0.96) | 5.37 (0.97) | 16.18 (0.32) |
| libimseti | **1,497.59** | 1,507.54 (0.99) | 1,503.49 (1.00) | 1,704.53 (0.88) |
| musae-twitch_DE | **20,906.93** | 21,470.00 (0.97) | 20,987.30 (1.00) | 20,949.83 (1.00) |
| musae-twitch_FR | **37.13** | 37.81 (0.98) | 37.32 (1.00) | 41.55 (0.89) |
| petster-fs-dog | **6.82** | 10.21 (0.67) | 8.67 (0.79) | 12.47 (0.55) |
| soc-LiveJournal1 | **9.87** | 11.50 (0.86) | 11.06 (0.89) | 23.91 (0.41) |
| web-BerkStan | **134.22** | 360.88 (0.37) | 138.84 (0.97) | 207.92 (0.65) |
| web-Google | **0.61** | 0.85 (0.71) | 0.68 (0.89) | 1.46 (0.41) |
| web-NotreDame | 12.10 | **9.07 (1.33)** | 12.11 (1.00) | 48.83 (0.25) |
| web-Stanford | >36,000 | **8.38 (>4,294.84)** | 27.41 (>1,313.18) | 42.80 (>841.16) |
| $s_{\text{total}}$ | 1.00 | 2.17 | **2.29** | 2.15 |

**Table 7** Detailed results for our reduction-based strategies on the PACE instances.

| Graph | max. deg. | Twin | Funnel | Unconfined | Packing | Combined |
|---|---:|---:|---:|---:|---:|---:|
| PACE | t | t (s) | t (s) | t (s) | t (s) | t (s) |
| 05 | 1.97 | 1.96 (1.01) | 1.99 (0.99) | 2.04 (0.97) | **1.66 (1.19)** | 2.11 (0.93) |
| 06 | 0.85 | 0.85 (1.00) | 0.74 (1.15) | 0.92 (0.92) | **0.67 (1.27)** | 0.81 (1.05) |
| 10 | 2.24 | 2.23 (1.01) | 2.23 (1.00) | 2.32 (0.97) | **1.88 (1.19)** | 2.06 (1.09) |
| 16 | 25,836.77 | 25,856.57 (1.00) | 22,446.13 (1.15) | 34,642.13 (0.75) | **18,511.88 (1.40)** | 22,590.78 (1.14) |
| 19 | 3.17 | 3.14 (1.01) | 2.90 (1.09) | 3.25 (0.98) | **2.60 (1.22)** | 3.04 (1.04) |
| 31 | 74.37 | 74.31 (1.00) | 58.14 (1.28) | 73.23 (1.02) | 55.99 (1.33) | **54.11 (1.37)** |
| 33 | 1.01 | **1.01 (1.00)** | 1.15 (0.88) | 1.14 (0.89) | 1.02 (0.99) | 1.29 (0.79) |
| 35 | 7.64 | 7.63 (1.00) | 7.37 (1.04) | 7.90 (0.97) | **6.54 (1.17)** | 7.75 (0.99) |
| 36 | **1.84** | 1.86 (0.99) | 11.44 (0.16) | 162.22 (0.01) | 1.90 (0.97) | 75.52 (0.02) |
| 37 | 10.27 | 10.31 (1.00) | 10.27 (1.00) | 10.63 (0.97) | **8.21 (1.25)** | 10.90 (0.94) |
| 38 | 12.33 | 12.36 (1.00) | 11.08 (1.11) | 11.40 (1.08) | 11.44 (1.08) | **10.05 (1.23)** |
| 39 | 93.79 | 93.99 (1.00) | **32.43 (2.89)** | 127.32 (0.74) | 93.99 (1.00) | 98.25 (0.95) |
| 40 | 4,690.64 | 4,689.28 (1.00) | 4,285.37 (1.09) | 4,530.07 (1.04) | 4,176.59 (1.12) | **4,131.79 (1.14)** |
| 41 | 48.56 | 48.42 (1.00) | 42.00 (1.16) | 48.66 (1.00) | **36.87 (1.32)** | 38.74 (1.25) |
| 42 | 37.32 | 37.19 (1.00) | 35.69 (1.05) | 37.60 (0.99) | **28.55 (1.31)** | 36.07 (1.03) |
| 43 | 175.11 | 174.63 (1.00) | 158.08 (1.11) | 172.91 (1.01) | **130.75 (1.34)** | 154.96 (1.13) |
| 44 | 92.90 | 92.97 (1.00) | 82.64 (1.12) | 94.37 (0.98) | **69.68 (1.33)** | 90.20 (1.03) |
| 45 | 25.41 | 25.37 (1.00) | 25.29 (1.01) | 26.20 (0.97) | **19.83 (1.28)** | 26.38 (0.96) |
| 46 | 109.55 | 109.47 (1.00) | 92.61 (1.18) | 108.01 (1.01) | **79.76 (1.37)** | 82.72 (1.32) |
| 47 | 58.47 | 58.18 (1.00) | 53.01 (1.10) | 59.16 (0.99) | **42.32 (1.38)** | 52.28 (1.12) |
| 48 | 25.28 | 25.21 (1.00) | 22.65 (1.12) | 25.72 (0.98) | **18.56 (1.36)** | 22.93 (1.10) |
| 49 | 17.80 | 17.76 (1.00) | 16.43 (1.08) | 19.02 (0.94) | **12.97 (1.37)** | 16.18 (1.10) |
| 50 | 48.87 | 48.90 (1.00) | 46.07 (1.06) | 49.75 (0.98) | **37.70 (1.30)** | 47.09 (1.04) |
| 51 | 56.70 | 56.58 (1.00) | 51.45 (1.10) | 57.63 (0.98) | **43.45 (1.31)** | 50.32 (1.13) |
| 52 | 22.16 | 22.12 (1.00) | 20.56 (1.08) | 22.99 (0.96) | **15.78 (1.40)** | 20.82 (1.06) |
| 53 | 59.88 | 59.88 (1.00) | 54.78 (1.09) | 60.43 (0.99) | **46.87 (1.28)** | 55.74 (1.07) |
| 54 | 32.08 | 32.02 (1.00) | 29.29 (1.10) | 32.89 (0.98) | **26.55 (1.21)** | 27.76 (1.16) |
| 55 | 6.83 | 6.80 (1.00) | 6.50 (1.05) | 6.99 (0.98) | **5.23 (1.31)** | 6.35 (1.08) |
| 56 | 97.00 | 96.45 (1.01) | 88.78 (1.09) | 98.09 (0.99) | **70.18 (1.38)** | 81.46 (1.19) |
| 57 | 66.01 | 65.97 (1.00) | 57.60 (1.15) | 65.90 (1.00) | **49.95 (1.32)** | 52.45 (1.26) |
| 58 | 48.12 | 47.74 (1.01) | 45.82 (1.05) | 48.56 (0.99) | **35.94 (1.34)** | 46.62 (1.03) |
| 59 | 13.30 | 13.30 (1.00) | 12.73 (1.04) | 13.72 (0.97) | **10.61 (1.25)** | 12.30 (1.08) |
| 60 | 79.56 | 79.36 (1.00) | 71.73 (1.11) | 80.70 (0.99) | **59.65 (1.33)** | 71.85 (1.11) |
| 61 | 21.91 | 21.91 (1.00) | 20.47 (1.07) | 22.28 (0.98) | **17.50 (1.25)** | 21.06 (1.04) |
| 62 | 66.22 | 66.18 (1.00) | 59.16 (1.12) | 67.83 (0.98) | **49.87 (1.33)** | 59.64 (1.11) |
| 63 | 69.06 | 68.81 (1.00) | 61.23 (1.13) | 70.81 (0.98) | **53.40 (1.29)** | 58.65 (1.18) |
| 64 | 29.58 | 29.38 (1.01) | 26.96 (1.10) | 29.46 (1.00) | **22.35 (1.32)** | 26.78 (1.10) |
| 65 | 36.84 | 36.72 (1.00) | 33.42 (1.10) | 37.93 (0.97) | **28.23 (1.30)** | 31.17 (1.18) |
| 66 | 8.06 | 8.06 (1.00) | 7.47 (1.08) | 8.21 (0.98) | **6.21 (1.30)** | 7.97 (1.01) |
| 67 | 122.74 | 122.34 (1.00) | 113.33 (1.08) | 123.58 (0.99) | **95.55 (1.28)** | 112.43 (1.09) |
| 68 | 8.79 | 8.75 (1.00) | 8.92 (0.99) | 8.94 (0.98) | **6.69 (1.31)** | 8.57 (1.03) |
| 69 | 43.11 | 43.11 (1.00) | 38.46 (1.12) | 44.18 (0.98) | **33.88 (1.27)** | 39.86 (1.08) |
| 70 | 11.79 | 11.73 (1.00) | 10.09 (1.17) | 12.22 (0.96) | **9.71 (1.21)** | 9.76 (1.21) |
| 71 | 36.20 | 35.91 (1.01) | 32.22 (1.12) | 35.37 (1.02) | **27.23 (1.33)** | 33.39 (1.08) |
| 72 | 46.44 | 46.18 (1.01) | 41.66 (1.11) | 46.68 (0.99) | **36.28 (1.28)** | 41.86 (1.11) |
| 73 | 43.02 | 43.00 (1.00) | 40.38 (1.07) | 43.77 (0.98) | **31.91 (1.35)** | 43.51 (0.99) |
| 74 | 7.06 | 7.06 (1.00) | 6.67 (1.06) | 7.86 (0.90) | **5.48 (1.29)** | 6.96 (1.01) |
| 77 | 13.30 | 13.25 (1.00) | 12.74 (1.04) | 13.80 (0.96) | **10.61 (1.25)** | 12.31 (1.08) |
| $s_{\text{total}}$ | 1.00 | 1.00 | 1.14 | 0.79 | **1.34** | 1.14 |

**Table 8** Detailed results for our reduction-based strategies on the DIMACS instances.

| Graph | max. deg. | Twin | Funnel | Unconfined | Packing | Combined |
|---|---|---|---|---|---|---|
| DIMACS | t | t (s) | t (s) | t (s) | t (s) | t (s) |
| C125.9 | 0.98 | 0.98 (1.00) | 0.92 (1.07) | 0.98 (1.00) | **0.85 (1.15)** | 0.91 (1.08) |
| MANN_a27 | 0.48 | 0.48 (1.00) | 0.57 (0.85) | 0.52 (0.92) | **0.48 (1.01)** | 0.59 (0.82) |
| MANN_a45 | 73.80 | 73.76 (1.00) | 83.81 (0.88) | 78.58 (0.94) | **71.86 (1.03)** | 85.47 (0.86) |
| brock200_1 | 137.34 | 136.98 (1.00) | 140.15 (0.98) | 137.32 (1.00) | **135.14 (1.02)** | 138.64 (0.99) |
| brock200_2 | 4.59 | 4.60 (1.00) | 4.71 (0.98) | 4.59 (1.00) | **4.58 (1.00)** | 4.70 (0.98) |
| brock200_3 | 22.06 | 21.78 (1.01) | 22.38 (0.99) | 21.85 (1.01) | **21.76 (1.01)** | 22.46 (0.98) |
| brock200_4 | 28.34 | **28.15 (1.01)** | 29.09 (0.97) | 28.16 (1.01) | 28.25 (1.00) | 29.24 (0.97) |
| gen200_p0.9_44 | 152.61 | 152.40 (1.00) | 136.94 (1.11) | 169.47 (0.90) | **132.81 (1.15)** | 149.63 (1.02) |
| gen200_p0.9_55 | 131.24 | 131.20 (1.00) | 125.61 (1.04) | 127.51 (1.03) | 102.10 (1.29) | **50.64 (2.59)** |
| hamming8-4 | 19.29 | 19.30 (1.00) | 19.78 (0.98) | **19.12 (1.01)** | 19.35 (1.00) | 19.67 (0.98) |
| johnson16-2-4 | 39.87 | 39.79 (1.00) | 41.63 (0.96) | 41.40 (0.96) | **38.70 (1.03)** | 43.09 (0.93) |
| keller4 | 2.62 | 2.62 (1.00) | 2.68 (0.98) | 2.63 (1.00) | **2.58 (1.02)** | 2.65 (0.99) |
| p_hat1000-1 | 860.24 | **859.74 (1.00)** | 870.92 (0.99) | 873.91 (0.98) | 862.77 (1.00) | 871.60 (0.99) |
| p_hat1000-2 | 33,035.45 | 33,314.15 (0.99) | 32,999.15 (1.00) | 32,812.80 (1.01) | **30,913.22 (1.07)** | 31,202.52 (1.06) |
| p_hat1500-1 | 8,935.77 | **8,935.50 (1.00)** | 9,009.69 (0.99) | 8,954.18 (1.00) | 8,958.19 (1.00) | 9,046.97 (0.99) |
| p_hat300-1 | 3.70 | 3.69 (1.00) | 3.78 (0.98) | 3.69 (1.00) | **3.68 (1.00)** | 3.78 (0.98) |
| p_hat300-2 | 5.53 | 5.53 (1.00) | 5.68 (0.97) | 5.54 (1.00) | **5.48 (1.01)** | 5.63 (0.98) |
| p_hat300-3 | 189.58 | 187.77 (1.01) | 189.16 (1.00) | 185.68 (1.02) | **175.01 (1.08)** | 179.53 (1.06) |
| p_hat500-1 | 38.63 | 38.70 (1.00) | 39.36 (0.98) | 39.03 (0.99) | **38.61 (1.00)** | 39.34 (0.98) |
| p_hat500-2 | 96.36 | 96.39 (1.00) | 97.87 (0.98) | 96.21 (1.00) | **95.08 (1.01)** | 96.96 (0.99) |
| p_hat500-3 | 14,860.70 | 14,887.15 (1.00) | 14,624.90 (1.02) | 14,765.90 (1.01) | **13,429.92 (1.11)** | 13,712.38 (1.08) |
| p_hat700-1 | 163.30 | **160.75 (1.02)** | 163.63 (1.00) | 160.81 (1.02) | 163.24 (1.00) | 163.31 (1.00) |
| p_hat700-2 | 906.32 | 908.46 (1.00) | 914.56 (0.99) | 906.78 (1.00) | **866.08 (1.05)** | 879.99 (1.03) |
| san1000 | **895.34** | 898.16 (1.00) | 906.21 (0.99) | 901.40 (0.99) | 913.29 (0.98) | 932.29 (0.96) |
| san200_0.7_1 | 10.85 | **10.78 (1.01)** | 11.01 (0.99) | 10.91 (0.99) | 10.93 (0.99) | 11.06 (0.98) |
| san200_0.7_2 | 0.33 | 0.32 (1.04) | 0.33 (0.98) | **0.31 (1.07)** | 0.32 (1.01) | 0.33 (0.99) |
| san200_0.9_1 | 13.93 | 13.90 (1.00) | 13.35 (1.04) | **4.94 (2.82)** | 12.03 (1.16) | 12.13 (1.15) |
| san200_0.9_2 | 34.15 | 33.87 (1.01) | 21.46 (1.59) | 12.32 (2.77) | 15.80 (2.16) | **10.01 (3.41)** |
| san200_0.9_3 | 1,069.00 | 1,068.17 (1.00) | 1,016.33 (1.05) | 639.01 (1.67) | 843.40 (1.27) | **600.71 (1.78)** |
| san400_0.5_1 | 9.21 | 9.21 (1.00) | 9.37 (0.98) | **9.13 (1.01)** | 9.24 (1.00) | 9.37 (0.98) |
| san400_0.7_1 | 1,125.52 | **1,121.99 (1.00)** | 1,146.32 (0.98) | 1,125.12 (1.00) | 1,132.10 (0.99) | 1,151.14 (0.98) |
| san400_0.7_2 | 3,062.38 | 3,063.23 (1.00) | 3,066.62 (1.00) | 3,463.29 (0.88) | **3,048.94 (1.00)** | 3,489.72 (0.88) |
| san400_0.7_3 | 4,411.82 | 4,405.26 (1.00) | 4,487.18 (0.98) | **4,398.18 (1.00)** | 4,497.81 (0.98) | 4,521.80 (0.98) |
| sanr200_0.7 | 48.35 | **48.34 (1.00)** | 50.09 (0.97) | 48.41 (1.00) | 48.49 (1.00) | 50.25 (0.96) |
| sanr200_0.9 | 679.25 | 679.65 (1.00) | 633.59 (1.07) | 664.95 (1.02) | **531.48 (1.28)** | 567.49 (1.20) |
| sanr400_0.5 | 373.40 | **370.59 (1.01)** | 376.93 (0.99) | 377.71 (0.99) | 370.72 (1.01) | 376.10 (0.99) |
| sanr400_0.7 | 29,766.80 | 29,838.40 (1.00) | 30,466.35 (0.98) | 29,844.65 (1.00) | **29,473.60 (1.01)** | 30,242.80 (0.98) |
| $s_{\text{total}}$ | 1.00 | 1.00 | 0.99 | 1.00 | **1.04** | 1.03 |

**Table 9** Detailed results for our reduction-based strategies on sparse networks.

| Graph | max. deg. | Twin | Funnel | Unconfined | Packing | Combined |
|---|---|---|---|---|---|---|
| Sparse net. | t | t (s) | t (s) | t (s) | t (s) | t (s) |
| as-skitter | 2,058.32 | 2,054.41 (1.00) | 1,849.79 (1.11) | 1,977.94 (1.04) | **1,681.87 (1.22)** | 1,704.73 (1.21) |
| baidu-relatedpages | 0.82 | **0.80 (1.02)** | 0.84 (0.97) | 0.85 (0.97) | 0.83 (0.98) | 0.93 (0.88) |
| bay | **1.68** | 1.68 (1.00) | 8.22 (0.20) | 4.71 (0.36) | 1.89 (0.89) | 8.38 (0.20) |
| col | **5,019.93** | 5,752.08 (0.87) | 5,416.72 (0.93) | 8,187.80 (0.61) | 9,370.05 (0.54) | 5,924.10 (0.85) |
| fla | **25.33** | 25.41 (1.00) | 45.62 (0.56) | 76.60 (0.33) | 34.78 (0.73) | 42.75 (0.59) |
| hudong-internallink | **0.99** | 1.31 (0.76) | 1.27 (0.78) | 1.21 (0.82) | 1.55 (0.64) | 1.12 (0.88) |
| in-2004 | 5.22 | **4.88 (1.07)** | 5.25 (0.99) | 10.85 (0.48) | 5.50 (0.95) | 10.73 (0.49) |
| libimseti | 1,497.59 | 1,452.17 (1.03) | 1,620.09 (0.92) | **1,440.71 (1.04)** | 1,476.25 (1.01) | 1,706.07 (0.88) |
| musae-twitch_DE | 20,906.93 | 20,996.87 (1.00) | 21,190.67 (0.99) | 22,650.53 (0.92) | **19,345.03 (1.08)** | 23,006.50 (0.91) |
| musae-twitch_FR | 37.13 | 37.04 (1.00) | 38.58 (0.96) | 41.15 (0.90) | **35.60 (1.04)** | 42.46 (0.87) |
| petster-fs-dog | 6.82 | **6.62 (1.03)** | 8.16 (0.84) | 8.66 (0.79) | 9.68 (0.70) | 9.20 (0.74) |
| soc-LiveJournal1 | 9.87 | **6.64 (1.49)** | 9.57 (1.03) | 9.49 (1.04) | 11.33 (0.87) | 10.69 (0.92) |
| web-BerkStan | 134.22 | 135.47 (0.99) | **122.30 (1.10)** | 146.94 (0.91) | 123.60 (1.09) | 174.07 (0.77) |
| web-Google | 0.61 | **0.53 (1.15)** | 0.69 (0.87) | 0.68 (0.89) | 0.78 (0.78) | 0.68 (0.89) |
| web-NotreDame | **12.10** | 12.63 (0.96) | 15.23 (0.79) | 12.38 (0.98) | 14.09 (0.86) | 17.52 (0.69) |
| web-Stanford | >36,000 | >36,000 | >36,000 | >36,000 | **17,886.35 (>2.01)** | 17,989.97 (>2.00) |
| $s_{\text{total}}$ | 1.00 | 0.97 | 0.98 | 0.86 | **1.31** | 1.30 |

# Nearest-Neighbor Queries in Customizable Contraction Hierarchies and Applications

**Valentin Buchhold**
Karlsruhe Institute of Technology, Germany

**Dorothea Wagner**
Karlsruhe Institute of Technology, Germany

## Abstract

Customizable contraction hierarchies are one of the most popular route planning frameworks in practice, due to their simplicity and versatility. In this work, we present a novel algorithm for finding $k$-nearest neighbors in customizable contraction hierarchies by systematically exploring the associated separator decomposition tree. Compared to previous bucket-based approaches, our algorithm requires much less target-dependent preprocessing effort. Moreover, we use our novel approach in two concrete applications. The first application are *online* $k$-closest point-of-interest queries, where the points of interest are only revealed at query time. We achieve query times of about 25 milliseconds on a continental road network, which is fast enough for interactive systems. The second application is travel demand generation. We show how to accelerate a recently introduced travel demand generator by a factor of more than 50 using our novel nearest-neighbor algorithm.

**2012 ACM Subject Classification** Theory of computation → Shortest paths; Theory of computation → Nearest neighbor algorithms; Applied computing → Transportation

**Keywords and phrases** Nearest neighbors, points of interest, travel demand generation, radiation model, customizable contraction hierarchies

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.18

**Supplementary Material** *Software*: `https://github.com/vbuchhold/routing-framework`
archived at `swh:1:dir:10f1a2a6b742bea7b45d72d563cf0de934e9648b`

## 1 Introduction

Motivated by route planning in road networks, the last two decades have seen intense research on speedup techniques [4] for Dijkstra's shortest-path algorithm [17], which rely on a slow preprocessing phase to enable fast queries. Particularly relevant to real-world production systems are customizable speedup techniques, which split preprocessing into a metric-independent part, taking only the network structure into account, and a metric-dependent part (the *customization*), incorporating edge weights (the *metric*). A fast and lightweight customization is a key requirement for important features such as real-time traffic updates and personalized metrics. The most prominent customizable techniques are *customizable route planning* (CRP) [12] and *customizable contraction hierarchies* (CCHs) [16]. Both achieve similar performance but with different trade-offs, and both are in use in industry.

Modern map-based services must support not only point-to-point queries but also many other types of queries. Over the years, both CRP and CCHs have been extended to numerous types of queries and problems. Efentakis and Pfoser [18] propose one-to-all and one-to-many algorithms within the CRP framework, and Efentakis et al. [19] extend CRP to nearest-neighbor queries. Delling and Werneck [14] present alternative CRP-based algorithms for the one-to-many and nearest-neighbor problem. Baum et al. [6] extend CRP so that it can find energy-optimal paths for electric vehicles, and Kobitzsch et al. [30] so that it can find multiple alternate routes from the source to the target.

Customizable contraction hierarchies and in particular *contraction hierarchies* (CHs) [23], the predecessors of CCHs, have also received considerable attention; see [4] for a recent overview. Since each CCH *is a* CH, all algorithms operating on CHs carry over to CCHs. Delling et al. [11] introduce PHAST, a one-to-all algorithm on CHs. RPHAST [13] is an extension to the one-to-many problem. Alternatively, one-to-many queries on CHs can be solved using the *bucket-based approach* by Knopp et al. [29]. Geisberger [22] extends the bucket-based approach to the nearest-neighbor problem.

In this work, we introduce a novel algorithm for finding $k$-nearest neighbors in CCHs. The *$k$-nearest neighbor problem* takes as input a graph $G = (V, E)$, a source $s \in V$, a nonempty set $T \subseteq V$ of targets, and an integer $k$ with $1 \leq k \leq |V|$. The goal is to find the $k$ targets $t_i \in T$ closest to $s$, i.e., those that minimize $dist(s, t_i)$, where $dist(v, w)$ is the shortest-path distance from $v$ to $w$. Modern nearest-neighbor algorithms tailored to road networks work in up to four phases [14, 13, 2]. *Preprocessing* takes as input only the network structure, *customization* incorporates the metric into the preprocessed data, *selection* (or *target indexing*) incorporates the set of targets into the data, and *queries* take a source and find the $k$ targets closest to the source. Our algorithm follows this standard four-phase setup.

Note that there is already a nearest-neighbor algorithm by Geisberger [22] which operates on CHs. However, its relatively heavy selection phase makes it only suitable for *offline* queries, where the set of targets is known in advance. This is the case for simple store locators of franchises. However, more common in interactive map-based services are *online* queries, where the set of targets is only revealed at query time. An example is the computation of the closest businesses whose name contains a user-defined keyword. We are not aware of any CH-based algorithm that can solve such queries.

There is indeed an algorithm [14] for online nearest-neighbor queries within the CRP framework. As already mentioned, however, CRP and CCHs are on a par with each other and both used in industry with good reasons. For a production system based on the CCH framework, it is usually not desirable to simultaneously maintain a CRP setup to support nearest-neighbor queries. All types of queries should be solvable within the CCH framework.

**Related Work.** We start by briefly reviewing the CH- and CRP-based nearest-neighbor algorithms mentioned above. Contraction hierarchies (CHs) [23] are a point-to-point route planning technique that is much faster than Dijkstra's algorithm (four orders of magnitude on continental networks). CHs replace systematic exploration of *all* vertices in the network with two much smaller search spaces (forward and reverse) in directed acyclic graphs, in which each edge leads to a "more important" vertex.

The basic idea behind the bucket-based nearest-neighbor algorithm [22] is to precompute and store the reverse CH search spaces of the targets during the selection phase. More precisely, if $v$ appears in the reverse search space from a target $t$ with distance $y$, then $(t, y)$ is stored in a *bucket* $B(v)$ associated with $v$. The bucket entries are sorted by nondecreasing distance. The query phase of the bucket-based nearest-neighbor algorithm computes the forward CH search space from the source $s$. For each vertex $v$ in the search space from $s$ with distance $x$, we scan the bucket $B(v)$. For each entry $(t, y) \in B(v)$, we obtain an $s$–$t$ path of length $x + y$. The algorithm maintains the $k$ closest targets seen so far and stops bucket scans when $x + y$ reaches the distance to the $k$-th closest target found so far.

Customizable route planning (CRP) [12] is a point-to-point route planning technique that splits preprocessing into a metric-independent part and a metric-dependent customization. Metric-independent preprocessing partitions the network into roughly balanced cells and creates *shortcuts* between each pair of boundary vertices in the same cell. Customization assigns costs to the shortcuts by computing shortest paths within each cell. Queries run a modification of bidirectional search that uses the shortcuts to skip over cells that contain neither the source nor the target. For better performance, we use multiple levels of overlays.

The CRP-based nearest-neighbor algorithm [14] marks all cells that contain one or more targets during the selection phase. Queries run a modification of Dijkstra's algorithm that skips over unmarked cells and descends into marked cells. Since the search discovers targets in increasing order of distance, we can stop when the $k$-th target is reached.

Of course, there are also nearest-neighbor algorithms tailored to road networks that are based on neither CRP nor CHs. Arguably the simplest one is *incremental network expansion* (INE) [33], which runs Dijkstra's algorithm until the $k$-th target is reached. Another straightforward approach is *incremental Euclidean restriction* (IER) [33]. The basic idea behind IER is to repeatedly retrieve the next closest target based on the straight-line distance (e.g., using an R-tree [26]) and compute the actual distance to it using any shortest-path algorithm as a black box. IER stops when the geometric distance to the next closest target exceeds the shortest-path distance to the $k$-th closest target so far encountered.

Since IER had only been evaluated using Dijkstra's algorithm, its performance was generally regarded as uncompetitive in practice. In particular, IER combined with Dijkstra cannot possibly be faster than INE. More recently, IER was combined with *pruned highway labeling* [3], yielding one of the fastest nearest-neighbor algorithms in many cases [2, 1].

More sophisticated nearest-neighbor algorithms are SILC [34, 35], ROAD [32, 31], and G-tree [40, 39]. Since previously published results had disagreed on the relative performance of these algorithms, Abeywickrama et al. [2] carefully reimplemented and reevaluated them once more. While G-tree was faster than SILC and ROAD in most cases, the differences were relatively small. Delling and Werneck [14] compare the CRP-based nearest-neighbor algorithm to G-tree, claiming that CRP outperforms G-tree. To sum up, all algorithms have comparable performance, with selection and query times of the same order of magnitude. However, a big advantage of CRP (and also of our algorithm) compared to the other approaches is a fast and lightweight customization phase, enabling important features such as real-time traffic updates and personalized metrics.

**Our Contribution.** We introduce a novel algorithm for finding $k$-nearest neighbors that operates on CCHs. Our algorithm systematically explores the associated separator decomposition tree in a way similar to nearest-neighbor queries [21] in kd-trees [7]. Its selection phase is orders of magnitude faster than the one of previous bucket-based approaches, which makes it a natural fit for online $k$-closest point-of-interest (POI) queries. On the road network of Western Europe, we achieve selection times of about 20 milliseconds and query times of a few milliseconds or less. This enables *interactive* online queries, which need to run both the selection and query phase for each client's request. We are not aware of any other nearest-neighbor algorithm operating on CCHs that enables interactive online queries.

In addition to closest-POI queries, we also look at a second concrete application. We show how a slightly modified version of our nearest-neighbor algorithm can be used for travel demand generation (or mobility flow prediction). Here, the problem we consider is computing the number $T_{vw}$ of trips between each pair $(v, w)$ of vertices $v, w$ in a road network. Depending on the expected length of the generated trips, we accelerate a recently introduced demand generator [8] by a factor of more than 50.

**Outline.** Section 2 reviews the CCH framework. Section 3 describes our novel nearest-neighbor algorithm in detail. Section 4 continues with two concrete applications in which our algorithm can be used. Section 5 presents an extensive experimental evaluation of various closest-POI algorithms and travel demand generators. Section 6 concludes with final remarks.

## 2 Preliminaries

We treat a road network as a bidirected graph $G = (V, E)$ where vertices represent intersections and edges represent road segments. Each edge $(v, w)$ has a nonnegative length $\ell(v, w)$ that represents the travel time from $v$ to $w$. A one-way road segment from $v$ to $w$ can be modeled by setting $\ell(w, v) = \infty$. The shortest-path distance (i.e., travel time) from $v$ to $w$ in $G$ is denoted by $dist(v, w)$. For simplicity, we assume that $G$ is strongly connected.

**Separator Decompositions.** A *separator decomposition* [5] of a strongly connected $n$-vertex bidirected graph $G = (V, E)$ is a rooted tree $\mathcal{T} = (\mathcal{X}, \mathcal{E})$ whose nodes $X \in \mathcal{X}$ are disjoint subsets of $V$ and that is recursively defined as follows. If $n = 1$, then $\mathcal{T}$ consists of a single node $X = V$. If $n > 1$, then $\mathcal{T}$ consists of a root $X \subseteq V$ that separates $G$ into multiple strongly connected subgraphs $G_0, \dots, G_{d-1}$. The children of $X$ are the roots of separator decompositions of $G_0, \dots, G_{d-1}$. For clarity, an element $v \in V$ is always called *vertex* and an element $X \in \mathcal{X}$ is always called *node*. We denote by $\mathcal{T}_X$ the subtree of $\mathcal{T}$ rooted at $X$ and we denote by $G_X$ the subgraph of $G$ induced by the vertices contained in $\mathcal{T}_X$. The vertex set of $G_X$ is represented by $V(G_X)$, and the edge set by $E(G_X)$.

In general, a separator $X \in \mathcal{X}$ should be small, and the resulting subgraphs $G_0, \dots, G_d$ should be balanced. Therefore, separator decompositions are typically obtained by recursive dissection (e.g., using Inertial Flow [36], FlowCutter [27], or InertialFlowCutter [25]).

**Nested Dissection Orders.** A separator decomposition $\mathcal{T}$ of $G$ induces a (not necessarily unique) *nested dissection order* $\pi$ on the vertices in $G$ [24]. To obtain one, we number the vertices in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, where the vertices in each node are visited in any order. Note that the resulting order $\pi = \langle \pi_0, \dots, \pi_{d-1}, \pi_d \rangle$ is split into $d+1$ contiguous subsequences $\pi_i$, where $d$ is the number of children $Y_j$ of the root $X$ of $\mathcal{T}$. The subsequences $\pi_0, \dots, \pi_{d-1}$ are nested dissection orders on $V(G_{Y_0}), \dots, V(G_{Y_{d-1}})$, and $\pi_d$ is an arbitrary order on $X$. (In the presence of turn costs, picking $\pi_d$ carefully improves performance [10].) We denote by $\pi^{-1}(v)$ the *rank* of $v$ in $\pi$.

**Customizable Contraction Hierarchies.** *Customizable contraction hierarchies* (CCH) [16] are a three-phase speedup technique to accelerate point-to-point shortest-path computations. The preprocessing phase computes a separator decomposition of $G$, determines an associated nested dissection order on the vertices in $G$, and *contracts* them in this order. To contract a vertex $v$, it is temporarily removed, and *shortcut* edges are added between its neighbors. The output of preprocessing is the input graph plus the shortcuts added during contraction. We call this graph $H$. We denote by $N_H^{\uparrow}(v)$ the set of neighbors of $v$ in $H$ ranked higher than $v$.

The customization phase computes the lengths of the edges in $H$ by processing them in bottom-up fashion. To process an edge $(u, w)$, it enumerates all triangles $\{v, u, w\}$ in $H$ where $v$ has lower rank than $u$ and $w$, and checks whether the path $\langle u, v, w \rangle$ improves the length of $(u, w)$. Alternatively, Buchhold et al. [9] enumerate all triangles $\{u, w, v'\}$ in $H$ where $v'$ has higher rank than $u$ and $w$, and check whether the path $\langle v', u, w \rangle$ improves the length of $(v', w)$, which accelerates customization by a factor of 2.

There are two query algorithms. First, one can run a bidirectional Dijkstra search on $H$ that only relaxes edges leading to vertices of higher ranks. Let a *forward CCH search* be a Dijkstra search that relaxes only outgoing upward edges, and a *reverse CCH search* one that relaxes only incoming downward edges. A *CCH query* runs a forward CCH search from the source and a reverse CCH search from the target until the search frontiers meet.

**Algorithm 1** Recursive formulation of our nearest-neighbor algorithm. At the first call, the parameter $X$ is the root of the separator decomposition tree.

```
Function searchSepDecomp($X$)
    if the recursion threshold is deceeded then
        examine all targets $t \in T \cap V(G_X)$ in the subgraph $G_X$
        return
    examine all targets $t \in T \cap X$ in the separator $X$
    $C \leftarrow \emptyset$
    foreach child $Y$ of $X$ do
        if $T \cap V(G_Y) \neq \emptyset$ then
            if $s \in V(G_Y)$ then
                $C \leftarrow C \cup \{(Y, 0)\}$
            else
                compute the distance $dist(s, Y)$ from $s$ to a closest vertex in $G_Y$
                $C \leftarrow C \cup \{(Y, dist(s, Y))\}$
    foreach $(Y, dist(s, Y)) \in C$ in ascending order of $dist(s, Y)$ do
        if $dist(s, Y)$ is less than the distance to the $k$-th closest target seen so far then
            searchSepDecomp($Y$)
```

In addition, there is a query algorithm based on the *elimination tree* of $H$. The parent of a vertex $v$ in the elimination tree is the lowest-ranked vertex in $N_H^{\uparrow}(v)$. Bauer et al. [5] prove that the ancestors of a vertex $v$ in the elimination tree are exactly the set of vertices scanned by a Dijkstra-based CCH search from $v$. An elimination tree search from $v$ therefore scans all vertices in the CCH search space of $v$ in order of increasing rank by traversing the path in the elimination tree from $v$ to the root. Since elimination tree queries use no priority queues, they are usually faster than Dijkstra-based CCH queries.

## 3 Our Nearest-Neighbor Algorithm

Our algorithm for finding nearest neighbors in CCHs is inspired by the algorithm of Friedman et al. [21] for finding nearest neighbors in kd-trees [7]. (However, our description requires no knowledge of that algorithm.) During the search, we maintain the $k$ closest targets seen so far in a max-heap $\hat{T}$ using their distances from the source as keys. Initially, $\hat{T} = \{\perp\}$ with $key(\perp) = \infty$. The basic idea is as follows: We systematically explore the separator decomposition tree, but visit only nodes $X$ whose corresponding subgraph $G_X$ contains vertices that are closer to the source than the $k$-th closest target found so far. For each visited node $X$, we compute the shortest-path distance from the source to each target in the separator $X$ (if any), and update $\hat{T}$ accordingly.

The precise algorithm is most easily formulated as a recursive procedure (see Algorithm 1). It takes a node $X$ in the separator decomposition tree as parameter. At the first call, $X$ is the root of the separator decomposition. The first step of the procedure is to *examine* all targets $t \in T \cap X$ in the separator $X$. To examine a target $t$, we compute the shortest-path distance $dist(s, t)$ from $s$ to $t$ with a standard elimination tree search. If $dist(s, t)$ is less than the maximum key in $\hat{T}$, we insert $t$ into the heap. If $\hat{T}$ now contains $k + 1$ elements, we delete the maximum element from the heap and discard it.

Next, we loop over all children $Y$ of $X$ in the separator decomposition tree. If the subgraph $G_Y$ induced by the vertices in $\mathcal{T}_Y$ contains any targets, we add a pair $(Y, \mathit{dist}(s, Y))$ to a set $C$. We denote by $\mathit{dist}(s, Y)$ the shortest-path distance from $s$ to a closest vertex in $G_Y$, i.e., $\mathit{dist}(s, Y) = \min_{v \in V(G_Y)} \mathit{dist}(s, v)$. If $G_Y$ contains the source, this distance is zero. Otherwise, we have to compute it, which we will discuss in the next sections.

Finally, we loop over all pairs $(Y, \mathit{dist}(s, Y)) \in C$ in ascending order of distance from the source. If $\mathit{dist}(s, Y)$ is less than the distance to the $k$-th closest target seen so far, we recurse on $Y$. Otherwise, $\mathcal{T}_Y$ cannot contain better solutions than those already known.

Note that when $G_X$ is large but contains only a few targets, it is less costly to loop over all these targets than to explore $\mathcal{T}_X$ until the leaves are reached. Therefore, when the number of targets in $G_X$ drops below a certain threshold, we stop the recursion and examine all targets $t \in T \cap V(G_X)$ in $G_X$ (we use a recursion threshold of 8 in our experiments, determined experimentally). The following sections work out the remaining details.

**Accessing Vertices and Targets in Subgraphs.** Given a node $X$ in the separator decomposition tree, our algorithm requires easy access to the set of vertices and the set of targets in the subgraph $G_X$ and in the separator $X$. Accessing the set of vertices in $G_X$ and in $X$ is particularly easy. To improve cache efficiency, the vertices in a CCH are reordered according to the order of contraction. That is, the vertices are numbered in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, where the vertices in each node are visited in any order. Hence, for each $X \in \mathcal{X}$, the vertices in $G_X$ are numbered contiguously, with the vertices in $G_X \setminus X$ appearing before the vertices in $X$. To support easy access to the vertices in $G_X$ and in $X$, we only need to store three indices with each $X$: the vertex in $G_X$ with the smallest index, the vertex in $G_X$ with the largest index, and the vertex in $X$ with the smallest index.

The set $T$ of targets is represented by a sorted array. To make the targets in subgraphs (or separators) easily accessible, we use an auxiliary array $A$ of size $|V| + 1$. The element $A[i]$, $0 \leq i \leq |V|$, stores the number of targets among the first $i$ vertices. Note that $A$ can be filled by a single sweep through $T$ and $A$. To access the targets in $G_X$ (or $X$), we first retrieve the index $l$ of the first vertex and the index $r$ of the last vertex in $G_X$ (or $X$), as discussed above. The number of targets in $G_X$ (or $X$) is then $A[r + 1] - A[l]$, and the actual targets are stored contiguously in $T[A[l]], \ldots, T[A[r + 1] - 1]$.

**Computing Shortest Paths to Subgraphs.** The most straightforward approach to compute the distance $\mathit{dist}(s, X)$ from $s$ to a closest vertex in $G_X$ is a standard Dijkstra-based CCH query, where the reverse search is initialized with all vertices in $G_X$. Let $d_{\mathsf{r}}$ and $Q_{\mathsf{r}}$ be the distance labels and the queue of the reverse search, respectively. To initialize the reverse search, we set $d_{\mathsf{r}}[v] = 0$ for each vertex $v \in V(G_X)$, $d_{\mathsf{r}}[w] = \infty$ for each vertex $w \in V \setminus V(G_X)$, and $Q_{\mathsf{r}} = V(G_X)$. This yields a correct but inefficient algorithm. However, we can do better.

We define the *boundary* $B(X)$ of $G_X$ as the set of vertices in $V \setminus V(G_X)$ that are adjacent to a vertex in $G_X$, i.e., $B(X) = \{w \in V \setminus V(G_X) : (v, w) \in E, v \in V(G_X)\}$. Note that the boundary of any $G_X$ is easily accessible without any additional preprocessing.

▶ **Lemma 1.** *Let $u$ be any vertex in $G_X$. Then, $\pi^{-1}(b) > \pi^{-1}(u)$ for each $b \in B(X)$.*

**Proof.** Consider any vertex $b \in B(X)$. Let $b$ be contained in the node $Y \notin V(\mathcal{T}_X)$. We claim that $Y$ lies on the path in $\mathcal{T}$ from $X$ to the root $R$. Assume otherwise, i.e., $Y$ does not lie on the $X$–$R$ path. Let $Z$ be the lowest common ancestor of $X$ and $Y$. Since $Z$ separates $G_X$ and $G_Y$ in $G_Z$, there is no edge in $G$ that connects $G_X$ and $G_Y$. This contradicts that $b$ is adjacent to a vertex in $G_X$. Thus, $Y$ lies on the $X$–$R$ path. Since the vertices are numbered in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, the vertices in $Y$ are assigned higher ranks than the ones in $X$. In particular, we have $\pi^{-1}(b) > \pi^{-1}(u)$. ◀

▶ **Theorem 2.** *Let $u$ be the highest-ranked vertex in $G_X$. Then, $B(X) = N_H^{\uparrow}(u)$.*

**Proof.** Let $b$ be a vertex in $B(X)$. We claim that $b \in N_H^{\uparrow}(u)$. Since $G_X$ is by definition connected, there is a path $\langle u, v_0, \dots, v_k, b\rangle$ in $G$ with $v_i \in G_X$. Since $\pi^{-1}(v_i) < \pi^{-1}(u)$ by definition and $\pi^{-1}(u) < \pi^{-1}(b)$ by Lemma 1, all $v_i$ are contracted before $u$ and $b$. Therefore, CCH preprocessing adds a shortcut $(u, b)$, and thus $b \in N_H^{\uparrow}(u)$.

Conversely, let $w$ be a vertex in $N_H^{\uparrow}(u)$, i.e., there is an edge $(u, w)$ in $H$. Since $u$ is the highest-ranked vertex in $G_X$ and $\pi^{-1}(u) < \pi^{-1}(w)$, we have $w \in V(G) \setminus V(G_X)$. We claim that $w \in B(X)$. Assume otherwise, i.e., $w \in V \setminus (V(G_X) \cup B(X))$. Since $B(X)$ separates $u$ and $w$ in $G$, the shortcut $(u, w)$ corresponds to a path $\langle u, \dots, b, \dots, w\rangle$ in $G$ with $b \in B(X)$. By construction, $b$ is contracted before $u$ and $w$. This contradicts Lemma 1. ◀

If $s \in V(G_X)$, then $dist(s, X) = 0$. So, assume $s \notin V(G_X)$. Since $B(X)$ separates $s$ and $G_X$, and all edge lengths are nonnegative, there is a closest vertex $v^*$ in $G_X$ such that there is a shortest $s$–$v^*$ path $\langle s, \dots, b, v^*\rangle$, $b \in B(X)$. Note that $(b, v^*)$ is a shortest edge among all edges $(b, v) \in E$, $v \in V(G_X)$; otherwise, $v^*$ would not be a closest vertex in $G_X$. Therefore, $dist(s, X) = \min_{b \in B(X)}(dist(s, b) + \min_{\{(b,v)\in E: v \in V(G_X)\}} \ell(b, v))$. That is, it suffices to initialize the reverse search of the query with all boundary vertices. More precisely, we set $d_{\mathsf{r}}[b] = \min_{\{(b,v)\in E: v\in V(G_X)\}} \ell(b, v)$ for each vertex $b \in B(X)$, $d_{\mathsf{r}}[w] = \infty$ for each vertex $w \in V \setminus B(X)$, and $Q_{\mathsf{r}} = B(X)$. This yields a reasonable algorithm, but we can do even better by exploiting elimination tree queries, which are usually faster than the Dijkstra-based CCH queries we have used so far.

Recall that the CCH search space $S(b)$ of a vertex $b$ corresponds to the path in the elimination tree from $b$ to the root $r$. An elimination tree search from $b$ therefore scans all vertices in $S(b)$ in order of increasing rank by traversing the $b$–$r$ path in the elimination tree. Given a set $B$ of vertices, it is not clear how to enumerate all vertices in the union of the search spaces, since the union generally corresponds to a subtree rather than a path in the elimination tree. However, we can exploit the fact that in our case $B$ is the boundary of $G_X$.

▶ **Theorem 3.** *Let $l$ be the lowest-ranked vertex in $B(X)$. Then, $S(l) = \bigcup_{b\in B(X)} S(b)$.*

**Proof.** Since $l \in B(X)$, we trivially have $S(l) \subseteq \bigcup_{b\in B(X)} S(b)$, so let $b \neq l$ be a vertex in $B(X)$. We claim that $S(b) \subseteq S(l)$. By Theorem 2, the highest-ranked vertex $u$ in $G_X$ is adjacent to both $l$ and $b$. Since $\pi^{-1}(u) < \pi^{-1}(l) < \pi^{-1}(b)$, CCH preprocessing adds a shortcut $(l, b)$ when $u$ is contracted. Therefore, we have $b \in S(l)$ and thus $S(b) \subseteq S(l)$. ◀

By Theorem 3, we can compute $dist(s, X)$ with a standard elimination tree query from $s$ to the lowest-ranked vertex in $B(X)$, where we initially set $d_{\mathsf{r}}[b] = \min_{\{(b,v)\in E: v\in V(G_X)\}} \ell(b, v)$ for each vertex $b \in B(X)$. Since a lower bound on $dist(s, X)$ suffices to preserve the correctness of our nearest-neighbor algorithm, we can also initialize the distance labels to zero. The resulting lower bound is only slightly worse than the exact distance, but initialization is somewhat faster. We observed the lowest running times when using lower bounds.

**Accelerating Shortest-Path Searches.** Note that the forward searches of all elimination tree queries done during the same nearest-neighbor query start at the same source. Unless we use special pruning criteria [9], the forward searches compute identical distance labels. To further accelerate our nearest-neighbor algorithm, we run the forward search *once* before the systematic exploration of the separator decomposition tree. Whenever we compute the distance to a target or subgraph, we run only the reverse search, which accesses the precomputed distance labels of the forward search.

After scanning a vertex $v$, a standard elimination tree search immediately initializes the distance label of $v$ to $\infty$, since it is not accessed anymore afterwards. We maintain this initialization approach for the reverse searches. The forward search, of course, must not immediately initialize the distance labels. Instead, after the exploration of the separator decomposition tree, we traverse the path in the elimination tree from the source to the root once again, and initialize the forward distance label of each visited vertex.

## 4 Applications

We continue with two substantially different applications in which our nearest-neighbor algorithm can be used. An obvious application are $k$-closest POI queries in map-based services. We can use our nearest-neighbor algorithm as is for this application, without further modifications. Afterwards, we look at a more abstract application (travel demand generation) where we make slight modifications to our algorithm.

### 4.1 Online Closest-POI Queries

Recall that modern closest-POI algorithms [14, 13, 2] work in up to four phases: preprocessing, customization, selection, and queries. We now divide the work our nearest-neighbor algorithm does into these standard phases. Note that our nearest-neighbor algorithm does nothing else but the standard CCH preprocessing and customization during the first two phases. To support easy access to the set of vertices in a subgraph or separator, we indeed need to associate three indices with each node $X \in \mathcal{X}$ but an efficient representation of the separator decomposition already stores this information. Therefore, we reuse the standard CCH preprocessing and customization, without further modifications.

The selection phase runs POI-dependent preprocessing. The only preprocessed data that depends on the set $P$ of POIs is the auxiliary array $A$, which makes the POIs in a subgraph or separator easily accessible. As already mentioned, $A$ can be filled by a single sweep through $P$ and $A$. Finally, the query phase runs the systematic exploration of the separator decomposition tree (including the forward search immediately before the exploration and the initialization of the forward distance labels immediately after the exploration).

Note that our selection phase is lightweight and (as our experiments will show) orders of magnitude faster than the one of previous bucket-based approaches. This makes our nearest-neighbor algorithm a natural fit for *online* $k$-closest POI queries, where the POIs are only revealed at query time. In this case, we need to run both the selection and query phase for each client's request. Except for simple store locators of franchises, online queries are more common than offline queries in interactive map-based services. For example, whenever the set of POIs is obtained from user-defined keywords, we face online queries.

### 4.2 Travel Demand Generation

A substantially different application in which our nearest-neighbor algorithm can be used is travel demand generation. Here, the problem we consider is computing the number $T_{vw}$ of trips between each pair $(v, w)$ of vertices $v, w \in V$. This problem arises when we want to generate large-scale benchmark data for evaluating transportation algorithms, or when we want to predict mobility flows. This section shows how our nearest-neighbor algorithm can be used to accelerate a recently introduced travel demand generator [8].

**Radiation Model.** The foundation for the aforementioned demand generator is the *radiation model* [37]. This model assumes that each vertex $v \in V$ has a nonnegative number $m_v$ of inhabitants and a nonnegative amount $n_v$ of opportunities. We denote by $M$ the total population in $G$ and by $N$ the total number of opportunities in $G$. The mobility flow out of each vertex is proportional to its population. Destination selection is based on the following main idea: Each traveler assigns to all opportunities a fitness or attractiveness value, drawn independently from a common distribution. Then, the traveler selects the closest opportunity with a fitness higher than the traveler's fitness threshold, drawn from the same distribution. The *radiation model with selection* [38] decreases the probability of selecting an opportunity by a factor of $1 - \lambda$. Intuitively, increasing $\lambda$ increases the expected trip length. In the simplest version, the number of opportunities is approximated by the population, i.e., there are $M$ opportunities in a graph with a population of $M$.

**Previous Implementations.** There are two practical implementations [8] of the radiation model. *DRAD* obtains high-quality solutions based on shortest-path distances and *TRAD* obtains high performance but uses geometric distances. Both implementations generate one trip after another. First, they draw the origin $O$ from a discrete distribution determined by the probability function $\Pr[O = v] = m_v/M$. Second, they choose the number $O_{\text{fit}}$ of opportunities with a fitness higher than the traveler's fitness threshold uniformly at random in $0..N$. Third, they draw the number $O_{\text{sel}}$ of selectable opportunities from a binomial distribution with $O_{\text{fit}}$ trials and success probability $1 - \lambda$. It remains to find the selectable opportunity closest to $O$, given the total number $O_{\text{sel}}$ of selectable opportunities in $G$. This is realized differently by the two implementations.

DRAD draws the number $O_{\text{int}}$ of opportunities that are closer to $O$ than any selectable opportunity from a negative hypergeometric distribution determined by $O_{\text{sel}}$ and $N$, and runs Dijkstra's algorithm from $O$, stopping as soon as $O_{\text{int}} + 1$ opportunities are visited. The last vertex scanned by the search is the destination of the current trip.

The basic idea of TRAD is to find the selectable opportunity closest to $O$ using a nearest-neighbor query [21] in a kd-tree [7]. Each node in a kd-tree corresponds to a region of the plane. The region of the root is the whole plane and the leaves correspond to small disjoint blocks partitioning the plane. The query algorithm traverses the kd-tree, starting at the root, and maintaining the number $O_{\text{sel}}(v)$ of selectable opportunities in the region corresponding to the current node $v$. Let $O_{\text{tot}}(v)$ be the total number of opportunities in the region of $v$.

When the traversal reaches an interior node $v$ in the kd-tree, the algorithm draws the number $O_{\text{sel}}(l)$ of selectable opportunities in the region of the left child $l$ from a hypergeometric distribution with $O_{\text{sel}}(v)$ draws without replacement from a population of size $O_{\text{tot}}(v)$ containing $O_{\text{tot}}(l)$ successes. The number $O_{\text{sel}}(r)$ of selectable opportunities in the region corresponding to the right child $r$ is set to $O_{\text{sel}}(v) - O_{\text{sel}}(l)$. The algorithm then recurses on the child whose region is closer to $O$, and when control returns, it recurses on the other child. The search is pruned at any vertex $v$ with $O_{\text{sel}}(v) = 0$, and at any vertex whose region is farther from $O$ than the closest selectable opportunity seen so far.

When the traversal reaches a leaf node $v$, the algorithm samples $O_{\text{sel}}(v)$ selectable opportunities in the region corresponding to $v$. For each of these opportunities, the algorithm checks whether it improves the closest selectable opportunity seen so far.

**Our Implementation.** We introduce a new implementation of the radiation model, which we call CRAD. Our implementation follows TRAD but uses nearest-neighbor queries in a customizable contraction hierarchy rather than in a kd-tree. In this way, we combine the efficient tree-based sampling approach from TRAD with shortest-path distances. As a result, our implementation obtains high-quality solutions like DRAD, but at much lower cost.

**Algorithm 2** Recursive procedure for finding the closest selectable opportunity in the subgraph $G_X$, given the number $O_{\text{sel}}(G_X)$ of selectable opportunities in $G_X$.

```
1  Function findClosestSelectableOpportunity($X, O_{sel}(G_X)$)
2      if the recursion threshold is deceeded then
3          sample $O_{\text{sel}}(G_X)$ selectable opportunities in the subgraph $G_X$
4          return
5      $\langle O_{\text{sel}}(G_{Y_0}), \dots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X)\rangle \leftarrow$
         multiHypergeomVariate($O_{\text{sel}}(G_X), \langle O_{\text{tot}}(G_{Y_0}), \dots, O_{\text{tot}}(G_{Y_{d-1}}), O_{\text{tot}}(X)\rangle$)
6      sample $O_{\text{sel}}(X)$ selectable opportunities in the separator $X$
7      $C \leftarrow \emptyset$
8      foreach child $Y$ of $X$ do
9          if $O_{sel}(G_Y) > 0$ then
10             if $O \in V(G_Y)$ then
11                 $C \leftarrow C \cup \{(Y, 0)\}$
12             else
13                 compute the distance $dist(O, Y)$ from $O$ to a closest vertex in $G_Y$
14                 $C \leftarrow C \cup \{(Y, dist(O, Y))\}$
15     foreach $(Y, dist(O, Y)) \in C$ in ascending order of $dist(O, Y)$ do
16         if $dist(O, Y)$ is less than distance to currently closest select. opportunity then
17             findClosestSelectableOpportunity($Y, O_{\text{sel}}(G_Y)$)
```

To use our nearest-neighbor algorithm in CRAD, we only need to make slight modifications to the procedure presented in Section 3 (see Algorithm 2 for the modified procedure). In addition to a node $X$ in the separator decomposition tree, it now takes the number $O_{\text{sel}}(G_X)$ of selectable opportunities in $G_X$ as second parameter. At the first call, $X$ is the root of the separator decomposition and $O_{\text{sel}}(G_X)$ is the number $O_{\text{sel}}$ of selectable opportunities in $G$, obtained as before in DRAD and TRAD. Let $Y_0, \dots, Y_{d-1}$ be the children of $X$. As the first step, the procedure now distributes the $O_{\text{sel}}$ selectable opportunities in $G_X$ over the subgraphs $G_{Y_0}, \dots, G_{Y_{d-1}}$ and the separator $X$. In contrast to TRAD where the opportunities are distributed among exactly two regions (left and right child), we now have $d+1$ regions ($d$ children and the separator). Therefore, $O_{\text{sel}}(G_{Y_0}), \dots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X)$ obey a *multivariate* hypergeometric distribution.

We can think of this distribution as drawing $O_{\text{sel}}(G_X)$ balls without replacement from an urn containing $O_{\text{tot}}(G_{Y_i})$ balls of type $i$ for $i = 0, \dots, d-1$ and $O_{\text{tot}}(X)$ balls of type $d$. We obtain $O_{\text{sel}}(G_{Y_i})$ balls of type $i$ for $i = 0, \dots, d-1$ and $O_{\text{sel}}(X)$ balls of type $d$.

After obtaining $O_{\text{sel}}(G_{Y_0}), \dots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X)$, we sample $O_{\text{sel}}(X)$ selectable opportunities in the separator $X$, and check whether any of them improves the closest selectable opportunity seen so far. Next, we loop over all children $Y$ of $X$ in the separator decomposition tree. If the subgraph $G_Y$ contains any selectable opportunities, we add a pair $(Y, dist(O, Y))$ to a set $C$ (recall that $O$ is the origin vertex of the current trip). The shortest-path distance $dist(O, Y)$ is computed as discussed in Section 3. Finally, we loop over all pairs $(Y, dist(O, Y)) \in C$ in ascending order of distance from the origin. If $dist(O, Y)$ is less than the distance to the closest selectable opportunity seen so far, we recurse on $Y$.

## 5 Experiments

This section presents a thorough experimental evaluation of both applications. First, we describe our experimental setup, including our benchmark machine, the inputs, and implementation details. Next, we evaluate various closest-POI algorithms, with a focus on their selection and query phases. Finally, we compare CRAD to DRAD and TRAD.

### 5.1 Experimental Setup

Our publicly available code[1] is written in C++17 and compiled with the GNU compiler 9.3 using optimization level 3. We use 4-heaps [28] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements). Our benchmark machine runs openSUSE Leap 15.2 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each with eight cores clocked at 3.50 GHz and $8 \times 64$ KiB of L1, $8 \times 1$ MiB of L2, and 24.75 MiB of shared L3 cache.

**Inputs.** Our benchmark instance is the road network of Western Europe. The network has a total of 18 017 748 vertices and 42 560 275 edges and was made available by PTV AG for the 9th DIMACS Implementation Challenge [15]. For the evaluation of the travel demand generators, we use the population grid[2] made available by Eurostat, the statistical office of the European Union. The grid has a resolution of one kilometer and covers all EU and EFTA member states, as well as the United Kingdom. We follow the approach in [8] to assign the grid to the graph. For each inhabitant, we pick a vertex lying in their cell uniformly at random and assign the inhabitant to it. If there is no such vertex, we discard the inhabitant.

**Implementation Details.** We use the network dissection algorithm Inertial Flow [36] to compute separator decompositions and associated nested dissection orders, with the balance parameter $b$ set to 3/10. CCH customization uses perfect witness searches [16].

For comparison, we carefully reimplemented the bucket-based nearest-neighbor algorithm by Geisberger [22], which we call BCH. CH preprocessing is taken from the open-source library RoutingKit[3]. Both the forward and reverse CH searches use stall-on-demand [23].

The bucket-based nearest-neighbor algorithm can be used as is on CCHs, without further modifications. For better performance, however, we use a tailored version where we replace the Dijkstra-based CH searches used during selection and queries by elimination tree searches. Note that in contrast to CH searches, CCH searches are faster without the stall-on-demand technique. On the other hand, stall-on-demand decreases the bucket sizes. Therefore, we use stall-on-demand only for the reverse searches. We call this version BCCH.

To keep implementation complexity of the demand generators low, we use existing implementations of random variate generation algorithms. The Standard Template Library (STL) offers the three distribution classes `uniform_int_distribution`, `binomial_distribution`, and `geometric_distribution`. The STL provides neither a hypergeometric nor a negative hypergeometric distribution. To generate hypergeometric variates, we use the stocc library[4]. Following [8], we approximate negative hypergeometric variates by geometric variates.

[1] `https://github.com/vbuchhold/routing-framework`
[2] `https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/population-distribution-demography/geostat`
[3] `https://github.com/RoutingKit/RoutingKit`
[4] `https://www.agner.org/random/`

**Table 1** Performance of different closest-POI algorithms for various POI distributions. For each distribution, we report the time to index a set of POIs (selection time), the space consumed by the index (selection space), and the time to find the $k = 1, 4, 8$ closest POIs (query time). For CRP, we take the figures for the online version from the original publication [14].

| | $\lvert P\rvert = 2^{12}, \lvert B\rvert = 2^{20}$ | | | | | $\lvert P\rvert = 2^{14}, \lvert B\rvert = \lvert V\rvert$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | selection | | query time [µs] | | | selection | | query time [µs] | | |
| | space | time | POIs to be reported | | | space | time | POIs to be reported | | |
| algo | [MiB] | [ms] | $k = 1$ | $k = 4$ | $k = 8$ | [MiB] | [ms] | $k = 1$ | $k = 4$ | $k = 8$ |
| Dij | – | – | 846 210 | 855 438 | 873 716 | – | – | 113.4 | 439.3 | 883.7 |
| BCH | 72.4 | 134 | 20 | 20 | 21 | 83.6 | 481 | 5.0 | 8.5 | 10.7 |
| BCCH | 85.5 | 453 | 51 | 52 | 53 | 134.9 | 1 753 | 6.0 | 8.8 | 11.1 |
| CCH | 68.7 | 21 | 2 353 | 3 501 | 4 629 | 68.7 | 23 | 306.7 | 494.8 | 702.0 |
| CRP | – | – | – | – | – | 0.0 | 8 | – | 640.0 | – |

## 5.2 Online Closest-POI Queries

We start by comparing our nearest-neighbor algorithm (simply called CCH in this section) to Dijkstra's algorithm, BCH, BCCH, and CRP. Note that the performance of closest-POI algorithms is affected not only by the number of POIs but also by their *distribution*. For example, the set of *all* restaurants may be distributed evenly over the whole network, whereas a certain franchise may operate in a local region. To model this, we follow the methodology used by Delling et al. [13] to evaluate one-to-many algorithms.

To obtain our problem instances, we first pick a center $c$ uniformly at random. We then use Dijkstra to grow a ball $B$ of size $|B|$ centered at $c$. Finally, we pick a POI set $P$ of size $|P|$ from $B$. By varying the parameters $|B|$ and $|P|$, we can model the aforementioned situations. For each combination, we generate 100 POI sets. Each POI set is evaluated with 100 sources picked at random. That is, each data point is an average over 10 000 queries.

**Main Results.** Table 1 shows the performance of different closest-POI algorithms for two POI distributions. We observe that Dijkstra's algorithm has reasonable performance when the POIs are evenly distributed over the whole graph ($|B| = |V|$). In this case, any potential source is relatively close to some POI, and thus the Dijkstra search can always stop early. However, Dijkstra's algorithm is not robust to the POI distribution. When $|B| = 2^{20}$, many potential sources are relatively far from any POI, and the average running times are around one second, too slow for interactive map-based services.

BCH achieves the best (offline) query times for both POI distributions. Note, however, that BCH is no competitor to BCCH, CCH, and CRP, since it operates on *standard* contraction hierarchies, which cannot handle frequent metric updates. We only include BCH in our experiments for comparison with BCCH, since the bucket-based nearest-neighbor algorithm has not been tested on *customizable* contraction hierarchies so far.

Although we tailored the bucket-based algorithm to CCHs, BCCH is still somewhat slower than BCH. This is expected, since CCHs contain more shortcuts and are thus denser than CHs. The slowdown is a factor of about 3.5 for selection. When $|B| = |V|$, BCCH has only slightly higher (offline) query times than BCH, since the queries relax only a few edges. However, BCCH queries are roughly 2.5 times slower than BCH queries when $|B| = 2^{20}$.

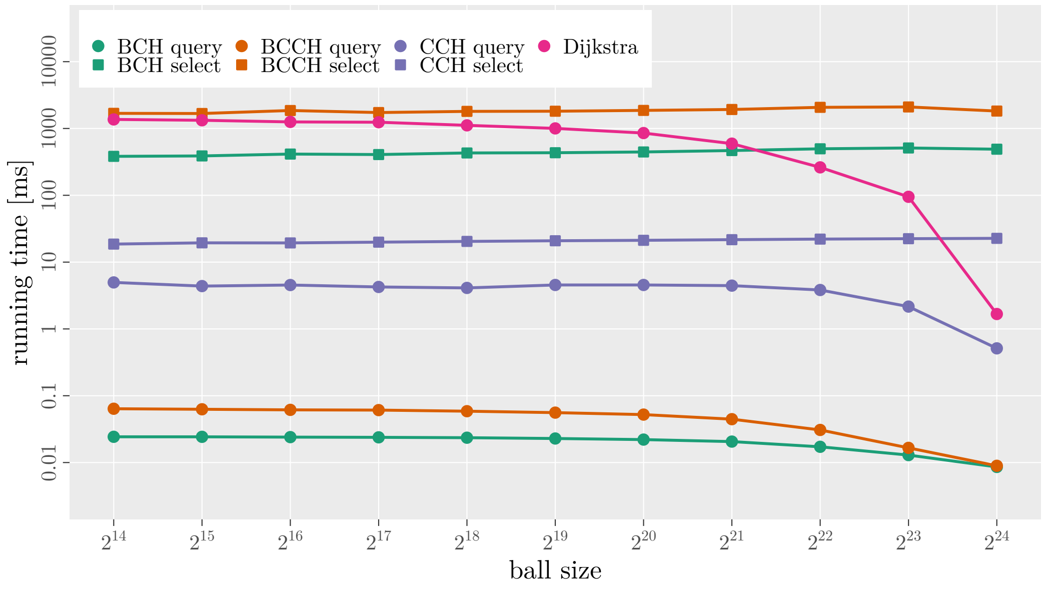


**Figure 1** Selection and query times of various closest-POI algorithms with $|P| = 2^{14}$ POIs picked at random from a ball of varying size $|B|$. Queries find the $k = 4$ closest POIs.

We observe that our nearest-neighbor algorithm (simply called CCH in this section) has considerably higher offline query times than BCCH. On the other hand, CCH achieves much faster selection times. For example, when $|P| = 2^{14}$, offline CCH queries are slower by a factor of 51–63 but CCH selection is faster by a factor of 77. Note that although CCH queries are significantly slower than BCCH queries, they are still slightly faster than CRP queries.

Online queries need to run both the selection and query phase for each client's request. Therefore, the time taken by an online query is the sum of the selection and query time. We observe that BCCH is not suitable for online queries. When $|P| = 2^{12}$, BCCH takes half a second to answer an online query, and it takes even 1.8 seconds when $|P| = 2^{14}$. In contrast, CCH takes only about 25 milliseconds for an online query.

Table 1 includes various alternative closest-POI algorithms. In addition, it seems natural to adapt existing one-to-many algorithms to the closest-POI problem. Promising candidates that are not based on buckets are CTD [20, 13] and RPHAST [13]. However, since CTD and RPHAST selection take more than 100 milliseconds when $|B| = |V|$, online closest-POI queries based on CTD or RPHAST would be at least four times slower than ours.

**Impact of the POI Distribution.** Our next experiment considers the impact of the ball size on the performance of the different closest-POI algorithms. Figure 1 plots selection and (offline) query times for various ball sizes. We omit online query times for clarity. Since the online query times are dominated by the selection times, online query times would closely follow the selection curves. Except for Dijkstra's algorithm, all selection and query times are very robust to the ball size. While all query algorithms benefit from an even distribution of the POIs (for the aforementioned reasons), this effect is most pronounced for Dijkstra.

**Impact of the Number of POIs.** Next, we evaluate the impact of the number of POIs on the performance of Dijkstra's algorithm, BCH, BCCH, and CCH. Figure 2 plots selection and (offline) query times for various numbers of POIs. As before, online query times would

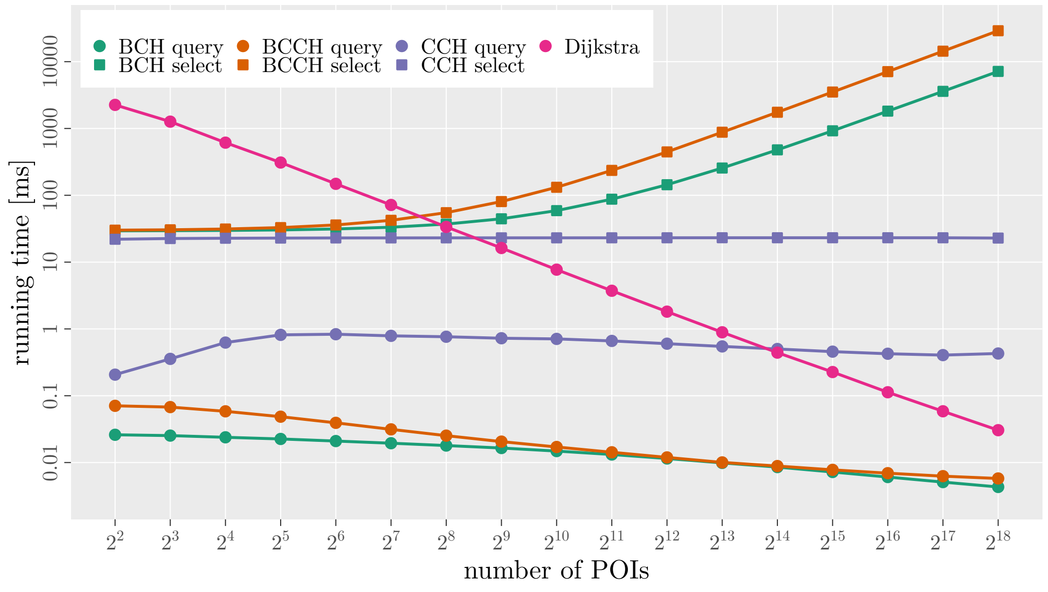


**Figure 2** Selection and query times of various closest-POI algorithms with a varying number $|P|$ of POIs picked at random from a ball of size $|B| = |V|$. Queries find the $k = 4$ closest POIs.

closely follow the selection curves. We observe that the CCH selection time is independent of the number of POIs, whereas the BCCH selection time grows linearly. For $|P| = 2^{14}$, CCH selection is 76 times faster than BCCH selection. The speedup increases to more than three orders of magnitude for $|P| = 2^{18}$, the largest number of POIs tested in our experiment.

Once again, queries tend to become faster as $|P|$ gets larger, since they can stop (in the case of Dijkstra-based searches) or prune (in the case of elimination tree searches) earlier. The exception are CCH queries, which become slower initially. The reason is that for very small values of $|P|$, we do not explore the separator decomposition tree but trigger the base case at the root (which simply finds $|P|$ point-to-point shortest paths by running standard elimination tree queries from the source to each POI).

## 5.3 Travel Demand Generation

Next, we evaluate CRAD, including a comparison to DRAD and TRAD. Since CRAD uses shortest-path distances rather than geometric distances, it obtains high-quality solutions like DRAD. We verified this experimentally by rerunning the experiments in the original publication [8] for CRAD, using the same instances and methodology. We refer to the original paper for a comparison of the solution quality with shortest-path and geometric distances.

In this work, we focus on the performance of the three implementations. Since DRAD is at its heart a Dijkstra search from the trip's origin to its destination, the performance depends heavily on the expected length of the generated trip (which is controlled by the parameter $\lambda$; see Section 4.2). In contrast, TRAD and CRAD are robust to the trip length.

Figure 3 plots the time to generate a single trip for various values of $\lambda$. Note that a value of $\lambda = 1 - 10^{-4}/1 = 0.9999$ leads on our instance to an average trip length of 9 minutes, and a value of $\lambda = 1 - 10^{-4}/100 = 0.999999$ to an average trip length of 72 minutes. Between two data points, the average trip length increases by about 7 minutes. All data points are averages over 100 000 trip generation executions.

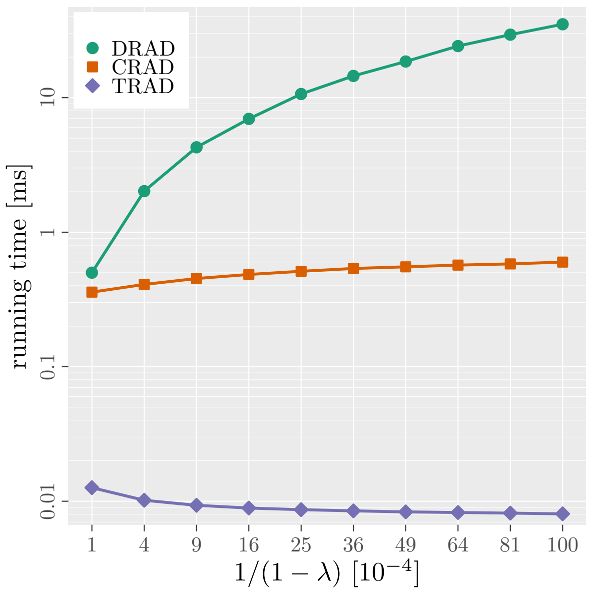


**Figure 3** Time to generate a single trip with different demand generators for various values of $\lambda$.

We observe that CRAD outperforms DRAD for each value of $\lambda$ tested. Since TRAD resorts to geometric distances, it still is faster than CRAD by a factor of 28–74. As it obtains worse solutions, however, TRAD is no competitor to CRAD. For an average trip length of about 23 minutes, CRAD gains an order of magnitude over DRAD, and for the largest value of $\lambda$ tested in our experiment, we see a speedup of 59. Note that this increase in speed is quite useful in practice. While travel demand generation does not need to run in real time, its performance should remain reasonable. However, DRAD takes about 7 hours to generate one million one-hour trips. In contrast, CRAD takes less than 10 minutes.

## 6 Conclusion

We presented a novel $k$-nearest neighbor algorithm that operates on CCHs. With selection times of about 20 milliseconds and query times of a few milliseconds or less, it is the first nearest-neighbor algorithm operating on CCHs that is fast enough for interactive online queries. Interestingly, our algorithm achieves similar performance as the online nearest-neighbor queries by Delling and Werneck [14] within the CRP framework. This confirms that CCHs and CRP are on an equal level and solve many types of problems equally well.

Moreover, we used our nearest-neighbor algorithm to significantly accelerate a recent travel demand generator. We proposed CRAD, a new implementation of the radiation model that combines the advantages of the two previous implementations DRAD and TRAD. CRAD obtains high-quality (shortest-path based) solutions like DRAD, but follows a more efficient tree-based sampling approach like TRAD.

Future work includes accelerating our nearest-neighbor algorithm even further. Note that we compute distances to subgraphs corresponding to the topmost nodes in the separator decomposition more often than distances to subgraphs corresponding to leaves. It would be interesting to see if it pays to precompute the reverse search spaces of the topmost subgraphs. Another possible approach would be to keep frequently used reverse search spaces in an LRU cache. Another interesting project is a parallel version of our algorithm that uses for example task-based parallelism to explore the separator decomposition tree. Finally, it would be interesting to port other point-of-interest algorithms to CCHs, for example best-via queries.

## References

1 Tenindra Abeywickrama and Muhammad Aamir Cheema. Efficient landmark-based candidate generation for kNN queries on road networks. In K. Selçuk Candan, Lei Chen, Torben Bach Pedersen, Lijun Chang, and Wen Hua, editors, *Proceedings of the 22nd International Conference on Database Systems for Advanced Applications (DASFAA'17)*, volume 10178 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2017. doi:10.1007/978-3-319-55699-4_26.

2 Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. K-nearest neighbors on road networks: A journey in experimentation and in-memory implementation. *Proceedings of the VLDB Endowment*, 9(6):492–503, 2016. doi:10.14778/2904121.2904125.

3 Takuya Akiba, Yoichi Iwata, Ken ichi Kawarabayashi, and Yuki Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*, pages 147–154. SIAM, 2014. doi:10.1137/1.9781611973198.14.

4 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.

5 Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. *Theoretical Computer Science*, 645:112–127, 2016. doi:10.1016/j.tcs.2016.07.003.

6 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-optimal routes for electric vehicles. In Craig A. Knoblock, Peer Kröger, John Krumm, Markus Schneider, and Peter Widmayer, editors, *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*, pages 54–63. ACM Press, 2013. doi:10.1145/2525314.2525361.

7 Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. doi:10.1145/361002.361007.

8 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Efficient calculation of microscopic travel demand data with low calibration effort. In Farnoush Banaei-Kashani, Goce Trajcevski, Ralf Hartmut Güting, Lars Kulik, and Shawn D. Newsam, editors, *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'19)*, pages 379–388. ACM Press, 2019. doi:10.1145/3347146.3359361.

9 Valentin Buchhold, Peter Sanders, and Dorothea Wagner. Real-time traffic assignment using engineered customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 24(2):2.4:1–2.4:28, 2019. doi:10.1145/3362693.

10 Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. Customizable contraction hierarchies with turn costs. In Dennis Huisman and Christos D. Zaroliagis, editors, *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*, volume 85 of *OpenAccess Series in Informatics (OASIcs)*, pages 9:1–9:15. Schloss Dagstuhl, 2020. doi:10.4230/OASIcs.ATMOS.2020.9.

11 Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. doi:10.1016/j.jpdc.2012.02.007.

12 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.

13 Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster batched shortest paths in road networks. In Alberto Caprara and Spyros C. Kontogiannis, editors, *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 52–63. Schloss Dagstuhl, 2011. doi:10.4230/OASIcs.ATMOS.2011.52.

**14** Daniel Delling and Renato F. Werneck. Customizable point-of-interest queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):686–698, 2015. `doi:10.1109/TKDE.2014.2345386`.

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

**16** Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, 2016. `doi:10.1145/2886843`.

**17** Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

**18** Alexandros Efentakis and Dieter Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In Andreas S. Schulz and Dorothea Wagner, editors, *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer, 2014. `doi:10.1007/978-3-662-44777-2_30`.

**19** Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. SALT. A unified framework for all shortest-path query variants on road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2015. `doi:10.1007/978-3-319-20086-6_23`.

**20** Jochen Eisner, Stefan Funke, Andre Herbst, Andreas Spillner, and Sabine Storandt. Algorithms for matching and predicting trajectories. In Matthias Müller-Hannemann and Renato F. Werneck, editors, *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX'11)*, pages 84–95. SIAM, 2011. `doi:10.1137/1.9781611972917.9`.

**21** Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977. `doi:10.1145/355744.355745`.

**22** Robert Geisberger. *Advanced Route Planning in Transportation Networks*. PhD thesis, Karlsruhe Institute of Technology, 2011. `doi:10.5445/IR/1000021997`.

**23** Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. `doi:10.1287/trsc.1110.0401`.

**24** Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973. `doi:10.1137/0710032`.

**25** Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):1–20, 2019. `doi:10.3390/a12090196`.

**26** Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984. `doi:10.1145/602259.602266`.

**27** Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM Journal of Experimental Algorithmics*, 23(1):1.2:1–1.2:34, 2018. `doi:10.1145/3173045`.

**28** Donald B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4(3):53–57, 1975. `doi:10.1016/0020-0190(75)90001-0`.

**29** Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007. `doi:10.1137/1.9781611972870.4`.

**30** Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. Evolution and evaluation of the penalty method for alternative graphs. In Daniele Frigioni and Sebastian Stiller, editors, *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 94–107. Schloss Dagstuhl, 2013. `doi:10.4230/OASIcs.ATMOS.2013.94`.

**31** Ken C. K. Lee, Wang-Chien Lee, and Baihua Zheng. Fast object search on road networks. In Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, editors, *Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09)*, pages 1018–1029. ACM Press, 2009. doi:10.1145/1516360.1516476.

32 Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. ROAD: A new spatial object search framework for road networks. *IEEE Transactions on Knowledge and Data Engineering*, 24(3):547–560, 2012. doi:10.1109/TKDE.2010.243.

33 Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In Johann-Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 802–813. Morgan Kaufmann, 2003.

34 Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable network distance browsing in spatial databases. In Jason Tsong-Li Wang, editor, *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, pages 43–54. ACM Press, 2008. doi:10.1145/1376616.1376623.

35 Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. Efficient query processing on spatial networks. In Cyrus Shahabi and Omar Boucelma, editors, *Proceedings of the 13th ACM International Workshop on Geographic Information Systems (GIS'05)*, pages 200–209. ACM Press, 2005. doi:10.1145/1097064.1097093.

36 Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015. doi:10.1007/978-3-319-20086-6_22.

37 Filippo Simini, Marta C. González, Amos Maritan, and Albert-László Barabási. A universal model for mobility and migration patterns. *Nature*, 484(7392):96–100, 2012. doi:10.1038/nature10856.

38 Filippo Simini, Amos Maritan, and Zoltán Néda. Human mobility in a continuum approach. *PLOS ONE*, 8(3):1–8, 2013. doi:10.1371/journal.pone.0060069.

39 Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. G-tree: An efficient index for KNN search on road networks. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM'13)*, pages 39–48. ACM Press, 2013. doi:10.1145/2505515.2505749.

40 Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2175–2189, 2015. doi:10.1109/TKDE.2015.2399306.

# A Graph-Based Similarity Approach to Classify Recurrent Complex Motifs from Their Context in RNA Structures

**Coline Gianfrotta**
Université de Versailles Saint-Quentin-en-Yvelines, Université Paris-Saclay, DAVID lab, France
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400, Orsay, France

**Vladimir Reinharz**
Department of Computer Science, Université du Québec à Montréal, Québec, Canada

**Dominique Barth**
Université de Versailles Saint-Quentin-en-Yvelines, Université Paris-Saclay, DAVID lab, France

**Alain Denise**
Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91400, Orsay, France
Université Paris-Saclay, CNRS, I2BC, 91400, Orsay, France

## Abstract

This article proposes to use an RNA graph similarity metric, based on the MCES resolution problem, to compare the occurrences of specific complex motifs in RNA graphs, according to their context represented as subgraph. We rely on a new modeling by graphs of these contexts, at two different levels of granularity, and obtain a classification of these graphs, which is consistent with the RNA 3D structure.

RNA many non-translational functions, as a ribozyme, riboswitch, or ribosome, require complex structures. Those are composed of a rigid skeleton, a set of canonical interactions called the secondary structure. Decades of experimental and theoretical work have produced precise thermodynamic parameters and efficient algorithms to predict, from sequence, the secondary structure of RNA molecules. On top of the skeleton, the nucleotides form an intricate network of interactions that are not captured by present thermodynamic models. This network has been shown to be composed of modular motifs, that are linked to function, and have been leveraged for better prediction and design. A peculiar subclass of complex structural motifs are those connecting RNA regions far away in the secondary structure. They are crucial to predict since they determine the global shape of the molecule, therefore important for the function.

In this paper, we show by using our graph approach that the context is important for the formation of conserved complex structural motifs. We furthermore show that a natural classification of structural variants of the motifs emerges from their context. We explore the cases of three known motif families and we exhibit their experimentally emerging classification.

**2012 ACM Subject Classification** Applied computing → Molecular structural biology

**Keywords and phrases** Graph similarity, clustering, RNA 3D folding, RNA motif

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.19

## 1 Introduction

RNA molecules are some of the major actors of the cell: many families of so-called non-coding RNAs intervene, along with proteins, in all major cellular processes. An RNA molecule is composed of a sequence of nucleotides (A, C, G, U) which folds in space into a three-dimensional structure. The function of an RNA molecule is strongly related to its three-dimensional structure. This is why many works since the 1970s have been dedicated to predict the structure of any RNA molecule from its sequence. The folding depends on interactions between the nucleotides. Strong interactions, called canonical interactions, first form what is called helices, stacks of canonical base pairs. They connect loops, and form the skeleton of the structure. Those loops are composed of weaker interactions, called non-canonical interactions and give the molecule its final structure [14].

It has been observed that specific loop geometries are conserved and found through various RNAs with different functions, with varying sequence [12, 17]. This conservation has been leveraged by graph and other geometric methods to predict structure from sequence [18, 25]. Yet all those methods only focus on interactions networks within one loop, which have been extensively studied [21, 5]. While specific complex joining loops together are well known, as the A-minor [13], only recent algorithmic progress, using a graph representation of the RNA, have allowed to extend this automatic classification to combinations of loops connected between themselves through additional non-canonical interactions [23]. A major challenge in the field is the prediction of the location of those interconnected pairs of loops, a crucial determinant for the structure, and therefore the function of the RNA.

To tackle this challenge, we propose that the structural context of a motif [10] such as a A-minor in a molecule can be used as a discriminant for peculiar complex geometries, as those joining pairs of loops. It is a matter of determining whether two structurally similar contexts induce identical geometry and function. Considering a modeling of molecules by graphs [8] or hypergraphs, several definitions and similarity approaches between molecules have already been studied [22], mainly due to the principle stating that structurally similar molecules are expected to display similar properties [26, 9, 16, 24]. To measure the similarity of structures of molecules, one main approach considers the resolution of the problem of finding a Maximum Common Edge Subgraph [22] (MCES) between two graphs. This NP-complete problem is initially seen as a generalization of graph isomorphism, with different metrics evaluating the size of this subgraph compared to those of the two graphs to be compared, in particular some specific to a molecular context [26, 6, 1, 2].

When consider solving the MCES problem to measure the structural similarity of molecular graphs, two limitations could occur. First, the required computation time is exponential with respect to the number of vertices of the two graphs, which is a major limitation when considering comparing one molecule with all molecules in a database. Second, considering molecular graphs could provide a similarity measure not sufficiently focused on structural similarity, especially if two molecular structures are similar, but the associated graphs differ slightly in number and nature of vertices and links. This is why we introduce here a new graph representation of the molecular structures of RNA at a level of granularity lower than that of the nucleotides, allowing in particular a reduction in the size of the graphs to be processed. We then solve MCES problem on these graphs, based on specific subgraph isomorphism definition, to study the targeted structural similarity.

We show that the similarity in our new graph representation correlates with the geometric distance between the 3D models, while reducing by 75% the computation time. We validate our approach by applying it to three specific known and complex RNA structural motifs. We observe that the clustering induced by the similarity measure segregates well the different structural contexts. This study shows that the structural context matters for those complex motifs and could be leveraged for the prediction of their location.

## 2 Representation of the Context of RNA Structural Motifs

### 2.1 Prior Definitions

Our study will focus on a peculiar kind of interactions between nucleotides that contribute to the 3D shape of an RNA molecule: the canonical and non canonical interactions. These interactions belong to 12 pairing families, according to their geometry, defined in the Leontis–Westhof nomenclature [11]. Three faces on each nucleotide can interact with another nucleotide. The pairing family depends on the face of the two interacting nucleotides (Watson-Crick (W), Hoogsteen (H), or Sugar (S)) and on the orientation of the interaction (*cis* (C) or *trans* (T)). In the Leontis–Westhof nomenclature, these families are represented by a three-letter code indicating the orientation of the interaction and the faces of the two nucleotides (for example, CSS for cis Sugar-Sugar), or by a symbol (see examples in Figure 1 and all the symbols in appendix A.1 Table 3). The canonical interactions belong to the CWW family.

We will now give some prior definitions, useful for the construction of our representation. After defining an RNA graph, we will describe particular RNA graphs that we will focus on in this work. We will then define what an occurrence of one of this particular RNA graphs is. To finish, we will define a particular subgraph of RNA graph we will use in Section 2.2 to represent structural contexts.

▶ **Definition 1.** RNA graph
An **RNA graph** is a connected mixed graph $G = (V, A, E)$, with $A$ a set of directed edges, also called arcs, and $E$ a set of undirected edges. This graph represents all or part of an RNA tertiary structure. Vertices of $V$ correspond to nucleotides, edges of $A$ to the bonds of the primary sequence, and edges of $E$ to canonical and non-canonical interactions between nucleotides.

- The set $A$ of directed edges constitutes one or several path(s) forming the primary sequence of molecules, oriented from the 5' end to the 3' end.
- For each edge $[x, y] \in E$, we define a type $\boldsymbol{t([x, y])}$. This type corresponds to the pairing family to which the undirected edge belongs, according to the Leontis-Westhof nomenclature [11]. In particular, undirected edges corresponding to canonical bonds are annotated as such (CAN). Not all non canonical bonds are symmetrical, which is why $t([x, y])$ can be different from $t([y, x])$.
- For each vertex $x \in V$, we also define a type $\boldsymbol{\tau(x)}$ according to its direct neighborhood. This type will be taken into account in the search for graph isomorphisms (see Section 3.1).
  - $\tau(x) = 0$ if $x$ has no incident edge (belonging to $E$)
  - $\tau(x) = C$ if $x$ has just one incident edge $[x, y] \in E$ and if $t([x, y]) = CAN$.
  - $\tau(x) = N$ if $x$ has at least one incident edge $[x, y'] \in E$ such as $t([x, y']) \neq CAN$ and no incident edge $[x, y] \in E$ such as $t[x, y] = CAN$.
  - $\tau(x) = M$ if $x$ has an incident edge $[x, y] \in E$ such as $t([x, y]) = CAN$ and at least another incident edge $[x, y'] \in E$ such as $t([x, y']) \neq CAN$.
- For each vertex $x \in V$ such as $\tau(x) = C$, we define the ***canonical neighbor*** of $x$ as the neighbor $y \in V$ of $x$ such as $[x, y] \in E$ and $t([x, y]) = CAN$. By definition of vertex types $\tau$, this neighbor exists and it is unique because a nucleotide cannot form more than one canonical bond.

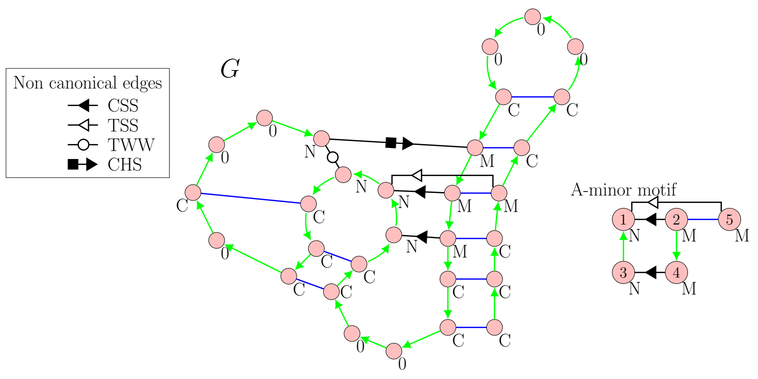


**Figure 1** Two examples of RNA graphs : a typical graph $G$ and the A-minor motif. The arcs are in green, the undirected edges of the canonical type are in blue and the undirected edges of the other types are in black, annotated by the Leontis–Westhof nomenclature [11]. Each vertex is annotated by its type.

Examples of RNA graphs are presented in Figure 1.

Note that, since we focus on the *structural* context only in this study, we do not consider the sequence (i.e. the types of nucleotides) in the RNA graph.

This work is focusing on particular RNA graphs, we called ***motifs***, that represent substructures frequently found in RNA tertiary structures as explained in the introduction (see Figure 1 for the example of A-minor motif).

▶ **Definition 2.** Motif occurrence
Given an RNA graph $G$, **a motif occurrence** is a partial subgraph of $G$, denoted as $O = (V^O, E^O, A^O)$, which is isomorphic to a motif $M = (V_M, A_M, E_M)$, with respect to the types of edges and vertices.

A motif occurrence is then a subgraph induced by the arcs and the edges of the motif $M$. The vertices of $V^O$ will be noted like the vertices of $V_M$ in $M$, for ease of writing. For example, for an A-minor motif, vertices of $V^O$ will be noted 1,2,3,4,5 (Figure 1). In the motif occurrence $O$, the types of the vertices of $V^O$ become specific to each vertex (for each $x \in V^O, \tau(x) = x$). An example of A-minor occurrence in an RNA graph $G$ is shown in Figure 2a.

▶ **Definition 3.** Specific subgraph of an RNA graph and a motif occurrence
Given an RNA graph $G$ and a motif occurrence $O$, the graph $\mathbf{G^{O,A-}} = (\mathbf{V}, \mathbf{A} \setminus \mathbf{A^O})$ is the spanning subgraph of $G$ having no edge of $E$ and having all the arcs of $A$ except those of the motif occurrence $O$.

The graph $G^{O,A-}$ is composed of chains, each of which containing at least one vertex of the motif occurrence $O$. There is an example of a graph $G^{O,A-}$ in Figure 2b. This subgraph will help us to define the structural context of a motif occurrence (see Section 2.2).

## 2.2 Definition of a k-extension

As seen in introduction, we are interested in comparing the structural contexts of motif occurrences in RNA graphs. This context consists in a special subgraph which contains and surrounds the motif. As the bounds in the primary sequence play an important role in

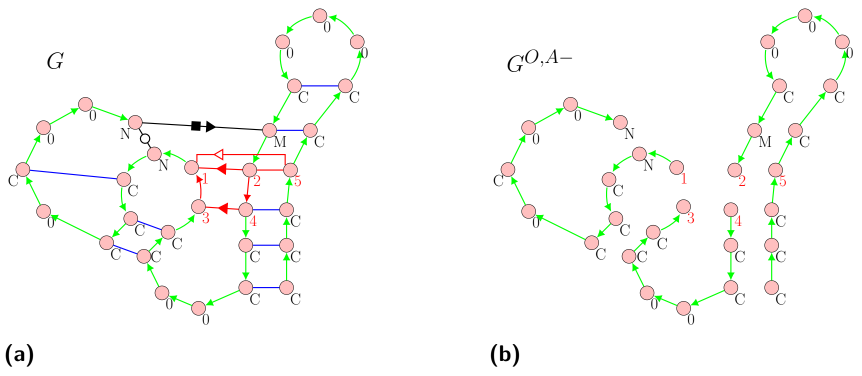


**Figure 2** In (a), the RNA graph $G$ with an A-minor occurrence in red, and in (b), the subgraph $G^{O,A-}$ of $G$. Every vertex is annotated by its type.

the tertiary structure, the graph $G^{O,A^-}$, which contains only this kind of bounds, plays a fundamental role in the definition of the context of a motif. We give below the definition of the structural context, that we call **k-extension of a motif occurrence**.

▶ **Definition 4.** *$k$-extension of a motif occurrence $O$*
Given a motif occurrence $O$, a subset $S$ of its vertices and an integer $k$, the **k-extension of a motif occurrence O according to S** is the subgraph $G_O = (V_O, A_O, E_O)$ of an RNA graph $G$, induced by three sets of vertices (which may be non-disjoint):

- the set $V^O$ of the vertices of the occurrence O (see definition 2)
- the set of vertices $V_k^O$ being at a distance strictly lower than $k$ in the graph $G^{O,A^-}$ (see definition 3), from one of the vertices of $S$.
- the set of vertices $V_k^{O+}$ neighbors by an edge of a vertex of $V_k^O$ in $G$.

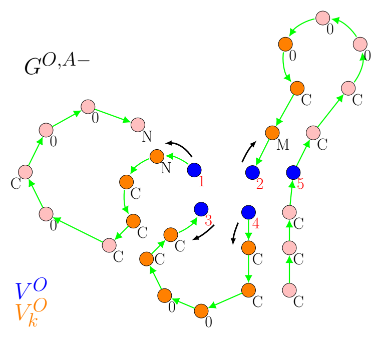
The set $S$ is the subset of vertices of the motif occurrence, from which we want to extend it. This choice will be explained in Section 5. For example, in the A-minor motif, we consider the first four vertices, i.e. the subset $\{1, 2, 3, 4\}$ (see the black arrows in Figure 3a).

In the example of RNA graph in Figure 3, the set of vertices $V^O$ is represented in blue, the set $V_k^O$ in orange and the set $V_k^{O+}$ in green (Figure 3b).

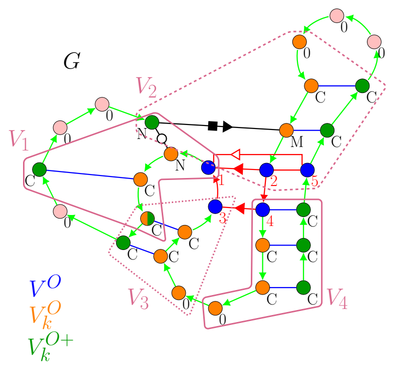
The vertices of $V_O$ are grouped into several subsets (not necessarily disjoint). In $G^{O,A-}$, we consider each path having for extremity one of the vertices of the motif occurrence $O$. The vertices of $V_k^O$ belonging to each of these paths constitute a subset of vertices. The vertices of $V_k^{O+}$ belong to the same subset(s) as their neighbor(s) in $V_k^O$. When we extend around an A-minor motif, we have in general four subsets of vertices, as shown in Figure 3b (framed in purple).

## 2.3 Definition of a Contracted k-extension

RNA structures are subject to modifications, due to evolution. Slight local changes in structures, like adding or deleting one nucleotide in a loop or an helix, may not change noticeably the 3D structure of the molecule, and thus may not change its function. This is why we present below a contracted representation of the context, allowing to represent similar but different contexts in an almost identical way. As will be seen in the Results section (Section 5), this new representation not only allows to better take the evolution into account, but also significantly decreases the computation time when comparing motif contexts.



**(a)** Graph $G^{O,A-}$ in which the vertices of $V^O$ are annotated in blue and the vertices of $V^O_k$ are annotated in orange for $k = 4$.



**(b)** RNA Graph G in which the vertices of $V^O$ are annotated in blue, the vertices of $V^O_k$ in orange and the vertices of $V^{O+}_k$ in green for $k = 4$.

**Figure 3** Construction of a k-extension for k=4. The vertices belonging to the k-extension are colored in blue, orange and green in both graphs $G^{O,A}$ (a) and $G$ (b). In (b) the four subsets of vertices ($V_1$, $V_2$, $V_3$ and $V_4$) are framed in purple.

We define a second graph $\tilde{G}_O$, derived from $G_O$, in which some edges and some vertices are contracted.

To do so, we have to define first the notion of *contractable path*, that will determine the vertices and the edges to contract. We define it in the graph $G^{O,A-}_O$ which is the spanning subgraph of the k-extension $G_O$ that contains no edge of $G_O$ and all the arcs of $G_O$ except those of the motif occurrence $O$ (see definition 3).

▶ **Definition 5.** Contractable paths in $G^{O,A-}_O$
A **contractable path** is a maximal path, in the graph $G^{O,A-}_O$ in which:

- the vertices are all of type C or all of type 0 and all belong to the same subset $V^O_k$ or all to the same subset $V^{O+}_k$
- and if the vertices are all of type C, the canonical neighbors of these vertices (see definition 1) also induce a contractable path in $G^{O,A-}_O$.

These paths connect vertices that are not involve in any edges (type 0) or vertices that are only involve in canonical edges (type C). It allows us to represent secondary structure elements, such as helices and loops, as blocs. Examples are presented in Figure 4.

▶ **Definition 6.** Contracted k-extension of a motif occurrence $O$
A **contracted k-extension**, denoted as $\tilde{G}_O$, is a graph derived from a k-extension $G_O$, in which the vertices of each contractable path in $G^{O,A-}_O$ are contracted in one single vertex. If these vertices are of type C, their canonical neighbors also induce a contractable path in $G^{O,A-}_O$ (according to the definition 5), and will thus be contracted. In this case, an edge of canonical type connects the two contracted vertices.

Each contracted vertex in $\tilde{G}_O$ is of the same type $\tau$ as the vertices from which it is derived in $G_O$. The number of vertices of $G_O$ grouped in $\tilde{G}_O$ in one single vertex $x \in V_O$ is noted $p(x)$. An example of graph $\tilde{G}_O$ is presented in Figure 4.

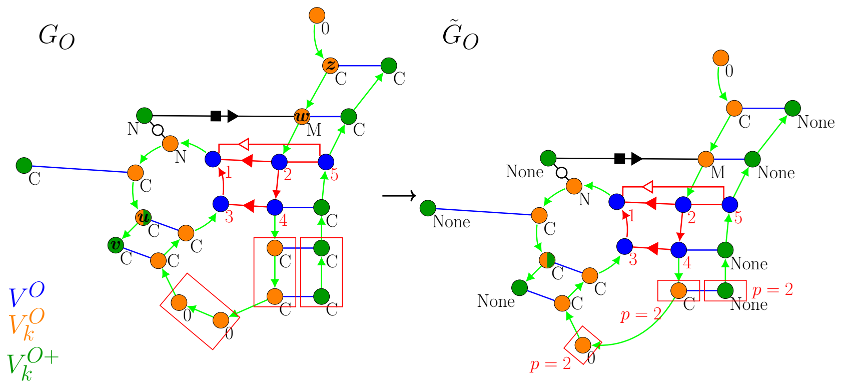


**Figure 4** On the left, the graph $G_O$ with 3 contractable paths circled in red. The path between the vertices $u$ and $v$ is not a contractable path because the vertex $u$ belongs to the subset $V_O^k$ and the vertex $v$ does not. In the same way, there is no contractable path between $w$ and $z$ because they are not of the same type (M for $w$ and C for $z$). Because of that, their canonical neighbors do not induce a contractable path either. On the right, the graph $\tilde{G}_O$ obtained by contraction, with the contracted vertices, framed in red and annotated by their weight $p$. The types of the vertices of $V_k^{O+}$ become *None*.

As defined in definition 5, the contractable paths are maximal paths, i.e. a set of contractable vertices cannot belong to a larger set of contractable vertices. Thus, the resulting graph $\tilde{G}_O$ is unique. Moreover, the same vertex cannot belong to two different contractable paths. Consequently, the graph $\tilde{G}_O$ does not depend on the order of treatment of the contractable paths.

In this model, the vertices of $V_k^{O+}$ in $\tilde{G}_O$ take the type *None* to differentiate them from the vertices of $V_k^O$.

The notations we defined in this section are summarized in the Table 1.

**Table 1** Summary of the graphs we define.

| | |
|---|---|
| $G$ | RNA graph |
| $O$ | motif occurrence in $G$ (subgraph of $G$, isomorphic to a motif) |
| $G^{O,A-}$ | spanning subgraph of $G$ containing no edge of $G$ and all the arcs of $G$ (except those belonging to the motif occurrence $O$) |
| $G_O$ | k-extension of a motif occurrence $O$ in $G$ (subgraph of $G$) |
| $G_O^{O,A-}$ | spanning subgraph of $G_O$ containing no edge of $G_O$ and all the arcs of $G_O$ (except those belonging to the motif occurrence $O$) |
| $\tilde{G}_O$ | contracted k-extension of a motif occurrence $O$ in $G$ (obtained from the contraction of vertices, arcs and edges in $G_O$) |

## 3 Similarity between Contracted k-extensions

We aim to compare the structural contexts of motif occurrences in RNA structures. For this purpose, we compare the contracted k-extensions of motif occurrences (noted $\tilde{G}_O$ in Section 2.3), in order to obtain, for each pair of contracted k-extensions, a common subgraph that maximizes a similarity metric we will define below in Section 3.2.

## 3.1 Maximum Common Subgraph : Variant of the MCES Problem

We will start by defining the maximum common subgraph on which we will calculate a similarity metric. To do so, we rely on the MCES problem.

The MCES problem aims to find a subgraph, common to any two graphs $G$ and $H$, maximizing the number of edges.

In our study, we search for a common subgraph as such, with supplementary constraints on the vertices and the edges, that we detail in the next paragraph.

Let $\tilde{G}_{O_1} = (\tilde{V}_{O_1}, \tilde{E}_{O_1}, \tilde{A}_{O_1})$ and $\tilde{G}_{O_2} = (\tilde{V}_{O_2}, \tilde{E}_{O_2}, \tilde{A}_{O_2})$ be two graphs of contracted k-extensions obtained from two motif occurrences $O_1$ and $O_2$ (Section 2.3).

We define $\tilde{G'}_{O_1} = (\tilde{V'}_{O_1}, \tilde{E'}_{O_1}, \tilde{A'}_{O_1})$ a subgraph of $\tilde{G}_{O_1}$ such that $\tilde{G'}_{O_1}$ contains the vertices of the motif occurrence $O_1$, and $\tilde{G'}_{O_2} = (\tilde{V'}_{O_2}, \tilde{E'}_{O_2}, \tilde{A'}_{O_2})$ a subgraph of $\tilde{G}_{O_2}$ such that $\tilde{G'}_{O_2}$ contains the vertices of the motif occurrence $O_2$.

We seek to find a subgraph $\tilde{G'}_{O_1}$, isomorphic to $\tilde{G'}_{O_2}$, and such that :

- each vertex $u \in \tilde{V'}_{O_1}$ is mapped to a vertex $v \in \tilde{V'}_{O_2}$ of the same type and belonging to a same subset of vertices (see 2.2),
- and such that each edge $[u_1, u_2] \in \tilde{E'}_{O_1}$ is mapped to an edge $[v_1, v_2] \in \tilde{E'}_{O_2}$ of the same type

Moreover, the subgraph $\tilde{G'}_{O_1}$ is not necessarily connected, but for all pairs of vertices $\{u, v\} \in \tilde{V'}^2_{O_1}$ in $\tilde{G'}_{O_1}$, if there is a path containing only arcs in $\tilde{G}_{O_1}$ between $u$ and $v$, there has to be a path containing only arcs in $\tilde{G}_{O_2}$ between the vertex mapped with $u$ in $\tilde{G'}_{O_1}$ and the vertex mapped with $v$ in $\tilde{G'}_{O_2}$. It means that the subgraph $\tilde{G'}_{O_1}$ must take into account the order of the vertices in these paths in the contracted k-extensions $\tilde{G}_{O_1}$ and $\tilde{G}_{O_2}$.

The subgraph $\tilde{G'}_{O_1}$ is thus a common subgraph to $\tilde{G}_{O_1}$ and $\tilde{G}_{O_2}$.

It has been shown that the decision problem associated with the calculation of a MCES between any two graphs is NP-complete [7]. Algorithms have been developed, able to solve the MCES problem for small instances, in particular for graphs representing molecules, such as the RASCAL algorithm [22]. This algorithm is an exact resolution of the problem. To find the MCES between two graphs G and H, it constructs the modular graph product P between the line graphs of G and H, and searches for a maximum clique in this graph P with a branch and bound method. We relied on this method to obtain the maximum common subgraph between our contracted k-extensions. We also developed a heuristic that builds the best common subgraph step by step, starting with the vertices with the highest degree.

## 3.2 Definition of the Similarity Metric to Maximize : the Contextual Graph Similarity

We will now explain how to evaluate the common subgraph we found, by defining a similarity measure, we call *contextual graph similarity.*

Although we have based ourselves on the RASCAL algorithm, the metric we want to maximize is slightly different. In the RASCAL algorithm, the similarity measure computes the number of edges and vertices in the common subgraph relative to the number of edges and vertices in the two initial graphs. Our contextual graph similarity takes into account only the number of edges, and not the number of vertices, in a common subgraph between two contracted k-extensions, because the interactions within an RNA molecule contribute the most to its tertiary structure. We do not consider arcs either, because we want to focus on the importance of canonical and non canonical interactions in RNA 3D structures.

▶ **Definition 7.** Contextual graph similarity
The **contextual graph similarity** between the two contracted k-extensions $\tilde{G}_{O_1}$ and $\tilde{G}_{O_2}$ is calculated as follows :

$$sim(\tilde{G'}_{O_1}, \tilde{G}_{O_1}, \tilde{G}_{O_2}) = \frac{\sum\limits_{[u,v] \in \tilde{E'}_{O_1} \setminus E^{O_1}} min(p(u), p(u'))}{max(\sum\limits_{[u,v] \in \tilde{E}_{O_1} \setminus E^{O_1}} p(u), \sum\limits_{[u,v] \in \tilde{E}_{O_2} \setminus E^{O_2}} p(u))}$$

with $u' \in \tilde{V'}_{O_2}$ the vertex in $\tilde{G'}_{O_2}$ (subgraph of $\tilde{G}_{O_2}$ isomorphic to $\tilde{G'}_{O_1}$, see Section 3.1), that is mapped with $u \in \tilde{V'}_{O_1}$ in $\tilde{G'}_{O_1}$

We count the proportion of edges in the common subgraph $\tilde{G'}_{O_1}$ compared to the maximum number of edges between $\tilde{G}_{O_1}$ and $\tilde{G}_{O_2}$. We do not take into account the edges of $E^{O_1}$, i.e. the edges of the occurrence $O_1$ of the motif (or $O_2$ as the occurrences are isomorphic). Indeed, by definition, these edges are present in all contracted k-extensions.

The vertices incident to the same edge have necessarily the same weight, noted $p$ (see Section 2.3). Each edge in $\tilde{G'}_{O_1}$ is weighted by the minimum weight of its incident vertices in $\tilde{G'}_{O_1}$ and their mapped vertices in $\tilde{G'}_{O_2}$. Each edge in $\tilde{G}_{O_1}$ or $\tilde{G}_{O_2}$ is weighted by the weight of its incident vertices. This number corresponds to the number of nucleotides that the vertex represents (see Section 2.3). In this definition of the metric,the weights of the vertices are thus taken into account, which means that small differences in the structure will be counted. However, thanks to the contracted graphs, it is possible to parameterize the metric to take into account the weights of the vertices in a less restrictive way.

To illustrate the behaviour of our metric, examples of common subgraphs with high and low contextual similarities are shown in Figure 5.

We can note that we are interested in the maximum contextual graph similarity value between two contracted k-extensions, that can be obtained from several different common subgraphs.

## 4 Classification of k-extensions and Search for a Maximum Common Graph to a Class

We seek to establish a classification of contracted k-extensions of motif occurrences (noted $\tilde{G}_O$ in Section 2.3).

For this purpose, we define a graph $G_s = (V_s, E_s, \omega)$, called **similarity graph**, in which each vertex represents a contracted k-extension of motif occurrence and there is an edge between all pairs of vertices, weighted by the contextual similarity value. This weighting is noted by the function $\omega : E_s \to [0, 1]$. In this similarity graph, we remove the edges weighted by a value inferior to a threshold $s$. This threshold $s$ is set so that contracted k-extensions with contextual similarity less than $s$ are considered as not similar.

Then we define a **classification** as a set of subsets of vertices $W = \{V_{s1}, V_{s2}, ..., V_{sn}\}$, such that $V_s = \bigcup_{i=1}^{n} V_{si}$. The subsets of vertices in $W$ are not necessarily two by two disjoint, which means that a vertex can belong to two different classes. Our classification is therefore a coverage of $G_s$ and not a partition. It allows us to take into account the case where one contracted k-extension is close to two other contracted k-extensions, which are, for their part, very different.

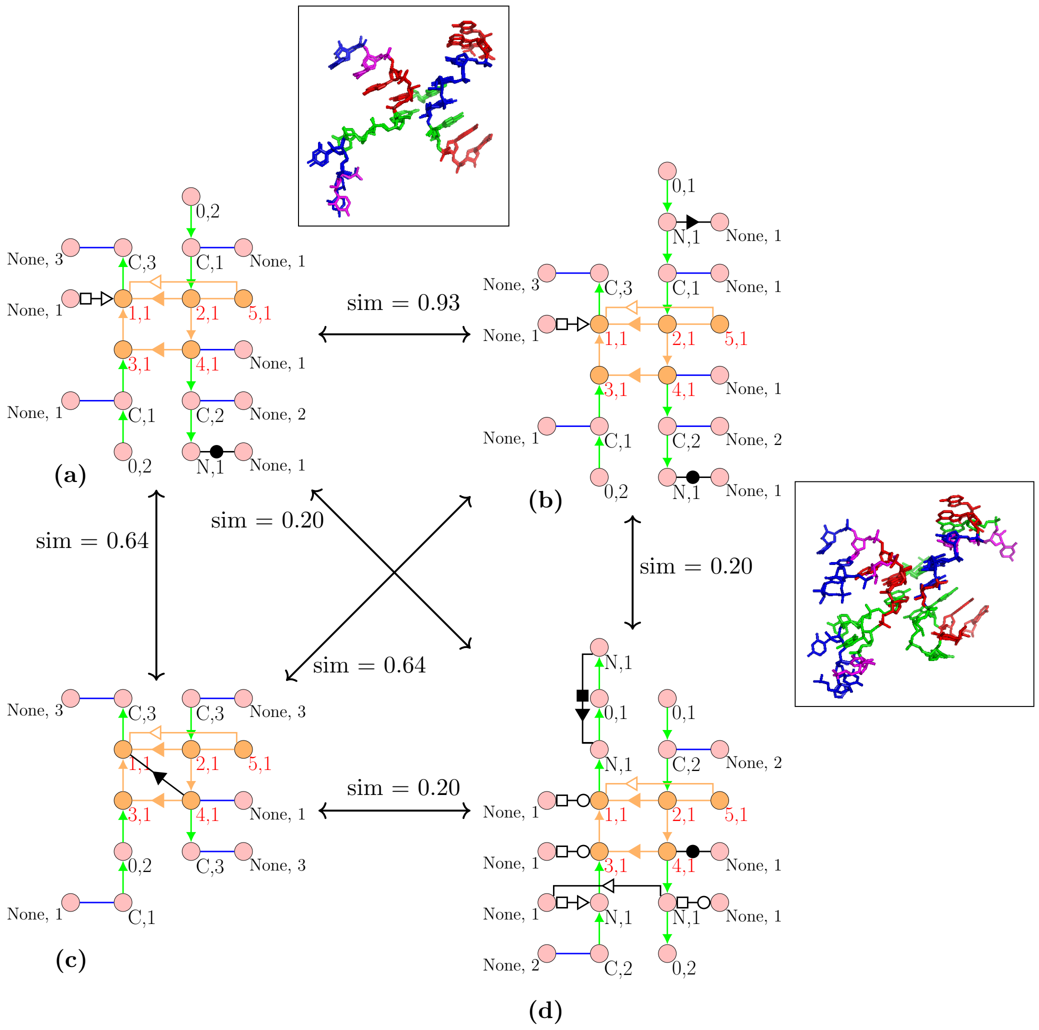
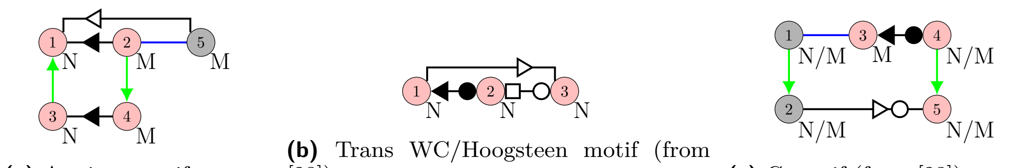


**Figure 5** Contextual graph similarities between four contracted 4-extensions. As seen before, arcs are represented in green, canonical edges in blue and non canonical edges in black, with the symbols of the Leontis–Westhof nomenclature. Each node is annotated by a doublet (type, weight). The 3D structure alignment of the 4-extensions (a) and (b) (resp. (b) and (c)) is presented at the top (resp. on the right). In the 3D structures, each type of nucleotides (A,C,G,U) is colored with the same colour. The two contracted 4-extensions above are the most similar (similarity of 0.93), and their corresponding 3D structures are very close too, as shown in the alignment. The 4-extension (c) has the smallest number of edges. However, it is still relatively similar to the 4-extensions (a) and (b) (similarity of 0.64). On the contrary, the 4-extension (d) is very different from the three others (similarity of 0.20), because it has many non canonical edges (represented in black) that do not appear in the other 4-extensions. The 3D alignment between (b) and (d) also highlights the differences.

We evaluate our classification according to cluster density and average similarity within clusters. Those two criteria allow us to obtain classes of similar contracted k-extensions, and so where the motif occurrences corresponding share close structural contexts. We thus apply a clustering method, developed in [20], that seeks to maximize those two criteria and also, that authorizes to obtain a coverage of the similarity graph and not a partition.



**(a)** A-minor motif. **(b)** Trans WC/Hoogsteen motif (from [23]). **(c)** G motif (from [23]).

**Figure 6** The three motifs we studied. The pink nodes constitute the subset of nodes from which we extend the motif to obtain the k-extensions (see Section 2.2).

We then characterize each of our classes by a representative. To do that, for each class of size $n$, we consider a maximum common subgraph to every contracted k-extensions of the class, defined in the same way as the maximum common subgraph for two graphs (Section 3.1), but for $n$ graphs. The quality of a class is notably linked to the size of this maximum common subgraph. The larger the size of the common subgraph, the more similar the contracted k-extensions of the class will be.

In Results section, we will analyze this classification in order to evaluate its relevance in a biological point of view.

## 5 Experimental Results

This section illustrates the relevance of our approach on three complex RNA motifs (Figure 6) : The A-minor motif, the Trans WC/Hoogsteen motif and a third motif which we call the G motif. These motifs are among those connecting RNA regions far away in the secondary structure. The A-minor motif frequently occurs in the RNA 3D structures, and has been proved to be involved in crucial cellular mechanisms [13]. The other two motifs come from the database of recurrent 3D motifs CaRNAval [23].

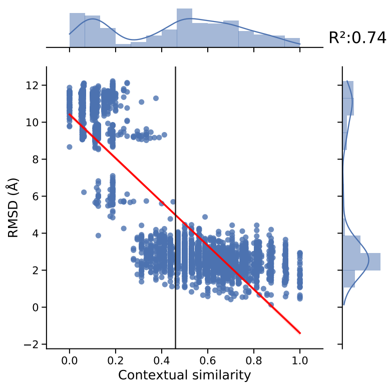
These three motifs are not predictable by current computational methods, to the best of our knowledge. That is why we choose to apply our method on those motifs. The fundamental question is: in terms of graphs, can the context of a motif help us to predict its presence in the molecule? The experiments shown in this section are intended to make progress towards answering this question. We first show that there is a clear correlation between our graph similarity and the geometrical similarity in 3D structures. Then we show that the automatic classification of RNA motif occurrences, based on our graph similarity, is consistent with RNA 3D structures. And finally, we show some advantages of the contracted graph representation, notably in terms of running time.

We applied our method on a dataset of non-redundant occurrences of those motifs from the PDB: 89 occurrences of the Trans WC/H motif, 391 of the A-minor motif, and 159 of the G motif. To choose the vertices from which we extend the motif (the subset $S$ in Definition 4), we considered vertices that are involved in non canonical edges and only one of the two incident vertices to a canonical edge (see the pink vertices in Figure 6).

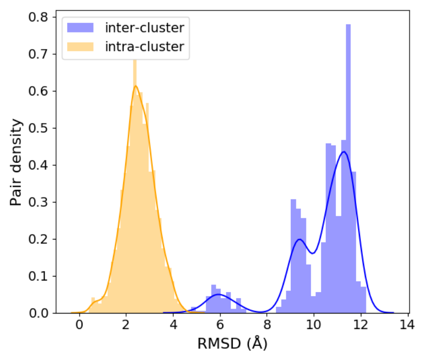
We chose an extension size $k$ of 4 because this size gives us the most discriminating results.

### 5.1 Correlation between Graph Similarity and 3D Similarity

Firstly, we compare our contextual similarity measure to a measure of similarity on the 3D structures. To do so, we consider, for each contracted k-extension in our dataset, the 3D structure of the RNA graph induced by the contracted k-extension. We then use the



**(a)** Distribution of the RMSD values according to the contextual graph similarity values. The linear regression line of the distribution is presented in red, and the correlation coefficient $R^2$ is indicated. The histograms in the margin of the diagram show the distribution of values of contextual similarity (above) and RMSD (on the right).



**(b)** Distribution of the inter-cluster and intra-cluster RMSD values, with a clustering obtained with a contextual similarity threshold of 0.46 (black line in (a)).

**Figure 7** Two representations of the distribution of RMSD values in relation to the contextual graph similarity values, for the Trans WC/H motif occurrences.

RMSD (Root Mean Square Deviation) [3] as a quantitative measure of similarity between 3D structures. We align each pair of 3D structures nucleotide by nucleotide and calculate the RMSD by considering each nucleotide by its carbon 3', as it is usually done ([15], [4]). Then we compare the RMSD values to our contextual graph similarity values. The lower the RMSD, the more similar the two considered structures are. On the contrary, a contextual graph similarity value near to 0 (resp. 1) indicates that the k-extensions are very different (resp. very similar).

We present the distribution of the RMSD values according to the contextual graph similarity values, for the occurrences of the Trans WC/H motif (Figure 7a). The correlation coefficient $R^2$ associated with this distribution is very high (0.74). This is confirmed by the diagram where two main sets of dots can be observed, corresponding to the occurrences with a very high RMSD (superior to 7.5Å) and a very low contextual similarity (inferior to 0.25), and to the occurrences with a low RMSD (inferior to 4Å) and with a contextual similarity superior to 0.3.

For the other two motifs (results shown in appendix A.2.1, Figure 9), the correlation coefficient is equal to 0.33 for the A-minor motif and to 0.56 for the G motif. Those two motifs have thus a correlation coefficient, not as high as the Trans WC/H motif. They seem to be less dependent on their local environment.

## 5.2 Motif Classification

We also classified the contracted k-extensions, according to the contextual graph similarity. We aim to determine whether grouping motif occurrences by similar environment leads to different classes, and whether these classes are consistent with the RMSD. To do so, we used the clustering method detailed in Section 4. This method requires to choose a similarity threshold, below which the contracted k-extensions cannot be placed in the same cluster.

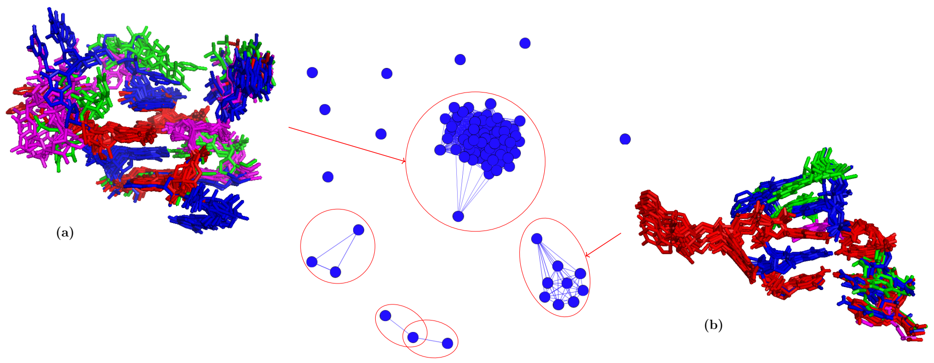


**Figure 8** Clustering and 3D alignments. In the middle, the similarity graph of the Trans WC/H motif occurrences. A node corresponds to an occurrence and there is an edge between two nodes if the contextual graph similarity between the contracted k-extensions is greater than 0.46. The clustering is indicated in red circles. On both sides, the 3D alignment of the contracted k-extensions of the two clusters. There is one color for each type of nucleotide. In (a), is presented the alignment of the 3D structures corresponding to a subset of contracted k-extensions of the largest cluster, and in (b), the alignment of the 3D structures corresponding to all of the contracted k-extensions of the smaller cluster.

We chose this threshold in an effort to have the better consistency between the contextual similarity values and the RMSD values. It means that the pairs of contracted k-extensions that have a contextual similarity value above (resp. below) the threshold must have similar (resp. not similar) 3D structures according to the RMSD. For the Trans WC/H motif, we choose a threshold of 0.46 because all the pairs of contracted k-extensions with a contextual similarity value above this threshold, correspond to 3D structures with a RMSD inferior to 5Å (Figure 7a). On the other hand, with this threshold, we lose some pairs with an RMSD value inferior to 3Å.

However, the Figure 7b shows a very clear consistency between the contextual similarity values and the RMSD values. Indeed, the RMSD of pairs of contracted k-extensions within a cluster does not generally exceed a value of 4.5Å and the RMSD of pairs of contracted k-extensions of two different clusters is generally superior to 4.5Å. This result thus also shows the relevance of our contextual graph similarity measure in relation to the RMSD.

Similar results hold for the two other motifs (see appendix A.2.1 Figure 10). The thresholds we have to choose to have a better consistency with RMSD values are higher (0.75 for the A-minor motif, and 0.65 for the G motif), and the maximum RMSD within clusters is higher for these two motifs, in particular for the A-minor motif where the RMSD values within clusters can reach 7Å for a few pairs of contracted k-extensions (appendix A.2.1 Figure 10).

We are now interested in the relevance of the classification itself. The classification we obtained for the Trans WC/H motif is composed of a large cluster of more than 60 occurrences, and four smaller clusters of respectively, 9, 3, 2 and 2 occurrences. The similarity graph (see Section 4) associated with this threshold is presented in Figure 8. It is a sufficiently dense graph for the classification to make sense, and to justify the use of a clustering method.

The cluster with 9 occurrences is composed of occurrences (and their contexts) sharing very close 3D structures (Figure 8, alignment (b)), and the maximum common subgraph (defined in Section 5.2) of the contracted k-extensions of this cluster is quite large. It is indeed composed of 6 edges (including 4 non canonical edges) which corresponds to one third of the number of edges in the smallest contracted k-extension of the cluster. These motif occurrences are found in RNA molecules of the same family, which explains their high 3D similarity. The largest cluster is composed of occurrences (and their contexts) sharing less close 3D structure. Indeed, the 3D alignment for a subset of occurrences of this cluster in Figure 8 (alignment (a)) is quite good for the right parts of the structure, but differences appear for the top left part. These motif occurrences are found in RNA molecules of different but close families. The classification obtained with the two other motifs, available in appendix A.2.2 Figure 11, also groups together motif occurrences sharing close 3D structures. These results show that the classification based on the contextual graph similarity measure, is able to group together motif occurrences in relevant clusters that share very similar environments in 3D.

## 5.3 Advantage of the Contracted Representation

The contracted graph representation presents two main advantages. The first one is the running time: the time needed to execute the search for a maximum common subgraph is largely reduced for the contracted k-extensions, even though it stays exponential with the exact method. The results obtained with the exact method, on the three motifs, are presented in Table 2. For the A-minor motif occurrences, for example, which corresponds to our larger dataset (391 occurrences), the contracted representation makes it possible to divide the execution time by 4.

Perhaps more importantly, contrarily to other approaches based also on graph isomorphism (e.g. [19, 23]), our metrics allows us to consider slight changes in the number of vertices and edges in the graphs as identical (see Section 2.3). This allows to group together contexts which are different at the graph level, but very similar in terms of 3D structure.

## 6 Conclusion and Perspectives

In this study, we wanted to determine if the structural context of complex motifs in RNA structures can give useful information about the 3D structure of this context and thus help to predict the presence and the position of the motif in RNA 3D structures.

To find out, we represented the structural contexts of motif occurrences by specific graphs, at two granularity levels, and developed a method, based on solving a MCES problem, to compare them using a dedicated similarity metric. The MCES problem is used in many approaches looking for similarities between molecules [26, 6, 1, 2], but here we have some additional constraints on the graphs and a different metric. The granularity of the graphs we defined allows us to consider two structural contexts as similar even if slight differences occur

**Table 2** Execution time of the search for maximum common subgraph for each pair of k-extensions (contracted or non contracted) for the three datasets, on a PC Intel Core i5-7440HQ 4x2.80GHzCPU.

| Motif | Execution time (in hours) for contracted k-extensions | Execution time (in hours) for non contracted k-extensions |
|---|---|---|
| A-minor motif (391 occurrences) | 4 | 16 |
| G motif (159 occurrences) | 0,8 | 2,2 |
| Trans W/H motif (89 occurrences) | 0,22 | 0,45 |

in terms of nucleotides and bonds, since they have little impact in the 3D configurations.

Our results show that there is a clear correlation between our contextual graph similarity measure and the RMSD, which is a measure of similarity on the 3D structures (Section 5.1). Moreover, the reduced size of our graphs compared to graphs representing each nucleotide and each interaction separately, allows a considerable saving in computing time, especially when searching in databases. We also established an automatic and exhaustive classification of the three motifs we studied (Section 5.2). This classification is consistent with the 3D structures, which means that it groups together motif occurrences that share both close structural contexts and close local 3D structures.

Regarding perspectives, we now have to study further the motif classifications. Notably, it will be worth considering the A-minor motif, which is ubiquitous in RNA structures and for which there is no available prediction method. We believe that a method which combines both our graph approach and sequence considerations could lead to useful results. Many other motifs have also to be studied, notably from the CaRNAval database [23].

From a more theoretical point of view, we plan to refine our similarity measure by devising weights for different classes of modifications in the RNA graphs. For example, nucleotide insertions and deletions could give different costs according to these parameters. Then, computing parameter values would need a thorough study of motifs in databases.

## References

1 Faisal N. Abu-Khzam, Nagiza F. Samatova, Mohamad A. Rizk, and Michael A. Langston. The maximum common subgraph problem: Faster solutions via vertex cover. In *IEEE/ACS International Conference on Computer Systems and Applications*, pages 367–373, 2007. `doi:10.1109/AICCSA.2007.370907`.

2 Tatsuya Akutsu and Hiroshi Nagamochi. Comparison and enumeration of chemical graphs. *Computational and Structural Biotechnology Journal*, 5, 2013. `doi:10.5936/csbj.201302004`.

3 Rafael Brüschweiler. Efficient RMSD measures for the comparison of two molecular ensembles. Root-mean-square deviation. *Proteins*, 50(1):26–34, 2003. `doi:10.1002/prot.10250`.

4 Emidio Capriotti and Marc A. Marti-Renom. RNA structure alignment by a unit-vector approach. *Bioinformatics*, 24(16):i112–i118, 2008. `doi:10.1093/bioinformatics/btn288`.

5 Grzegorz Chojnowski, Tomasz Waleń, and Janusz M. Bujnicki. RNA Bricks—a database of RNA 3D motifs and their interactions. *Nucleic acids research*, 42(D1):D123–D131, 2014.

6 Hanna Eckert and Jürgen Bajorath. Molecular similarity analysis in virtual screening: foundations, limitations and novel approaches. *Drug Discovery Today*, 12(5):225–233, 2007. `doi:10.1016/j.drudis.2007.01.011`.

7 Michael R. Garey and David S. Johnson. *Computers and intractability*, volume 29. WH Freeman New York, 2002.

8 Johann Gasteiger. *Handbook of Chemoinformatics: From Data to Knowledge*. Wiley, 1 edition, 2003. `doi:10.1002/9783527618279`.

9 Mark A. Johnson and Gerald M. Maggiora. *Concepts and applications of molecular similarity*. The American Chemical Society, 1988.

10 Neocles B. Leontis, Aurélie Lescoute, and Eric Westhof. The building blocks and motifs of RNA architecture. *Current Opinion in Structural Biology*, 16(3):279–287, 2006. `doi:10.1016/j.sbi.2006.05.009`.

11 Neocles B. Leontis and Eric Westhof. Geometric nomenclature and classification of RNA base pairs. *RNA*, 7(4):499–512, April 2001.

12 Neocles B. Leontis and Eric Westhof. Analysis of RNA motifs. *Current opinion in structural biology*, 13(3):300–308, 2003.

13 Aurélie Lescoute and Eric Westhof. The A-minor motifs in the decoding recognition process. *Biochimie*, 88(8):993–999, 2006. `doi:10.1016/j.biochi.2006.05.018`.

14 Aurélie Lescoute and Eric Westhof. The interaction networks of structured RNAs. *Nucleic acids research*, 34(22):6587–6604, 2006.

15 Marcin Magnus, Kalli Kappel, Rhiju Das, and Janusz M. Bujnicki. RNA 3D structure prediction guided by independent folding of homologous sequences. *BMC Bioinformatics*, 20(1):512, 2019. doi:10.1186/s12859-0193120-y.

16 Stefi Nouleho Ilemo, Dominique Barth, Oliver David, Franck Quessette, Marc-Antoine Weisser, and Dimitri Watel. Improving graphs of cycles approach to structural similarity of molecules. *PLOS ONE*, 14(12):1–25, 2019. doi:10.1371/journal.pone.0226680.

17 Carlos Oliver, Vincent Mallet, Roman Sarrazin-Gendron, Vladimir Reinharz, William L. Hamilton, Nicolas Moitessier, and Jérôme Waldispühl. Augmented base pairing networks encode RNA-small molecule binding preferences. *Nucleic acids research*, 48(14):7690–7699, 2020.

18 Marc Parisien and François Major. The MC-Fold and MC-Sym pipeline infers RNA structure from sequence data. *Nature*, 452:51–5, 2008. doi:10.1038/nature06684.

19 Samuela Pasquali, Hin H. Gan, and Tamar Schlick. Modular RNA architecture revealed by computational analysis of existing pseudoknots and ribosomal RNAs. *Nucleic Acids Research*, 33(4):1384–1398, 2005. doi:10.1093/nar/gki267.

20 Airel Pérez-Suàrez, José F. Martínez-Trinidad, Jésus A. Carrasco-Ochoa, and José E. Medina-Pagola. An algorithm based on density and compactness for dynamic overlapping clustering. *Pattern Recognition*, 46(11):3040–3055, 2013. doi:10.1016/j.patcog.2013.03.022.

21 Anton I. Petrov, Craig L. Zirbel, and Neocles B. Leontis. Automated classification of RNA 3D motifs and the RNA 3D Motif Atlas. *Rna*, 19(10):1327–1340, 2013.

22 John Raymond, Eleanor Gardiner, and Peter Willett. RASCAL: Calculation of Graph Similarity using Maximum Common Edge Subgraphs. *Computer Journal*, 45:631–644, April 2002.

23 Vladimir Reinharz, Antoine Soulé, Eric Westhof, Jérôme Waldispühl, and Alain Denise. Mining for recurrent long-range interactions in RNA structures reveals embedded hierarchies in network families. *Nucleic Acids Research*, 46(8):3841–3851, 2018. doi:10.1093/nar/gky197.

24 Roger Sayle, John May, Noel O'Boyle, J. Andrew Grant, Stefan Senger, and Darren V.S. Green. Chemical similarity based on graph edit distance:efficient implementation and the challenges of evaluation. In *7th Joint Sheffield Conference on Chemoinformatics*, 2015.

25 Jason Yao, Vladimir Reinharz, François Major, and Jérôme Waldispühl. RNA-MoIP: prediction of RNA secondary structure and local 3D motifs from sequence data. *Nucleic acids research*, 45(W1):W440–W444, 2017.

26 Laura A. Zager and George C. Verghese. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(45):86–94, 2008. doi:10.1016/j.aml.2007.01.006.

## A Appendix

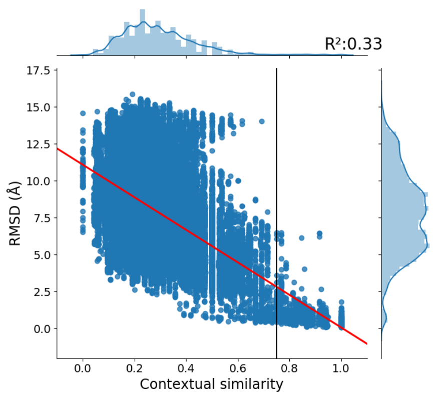
### A.1 Representation of the Context

**Table 3** Symbols of the Leontis–Westhof nomenclature for the non canonical interactions.

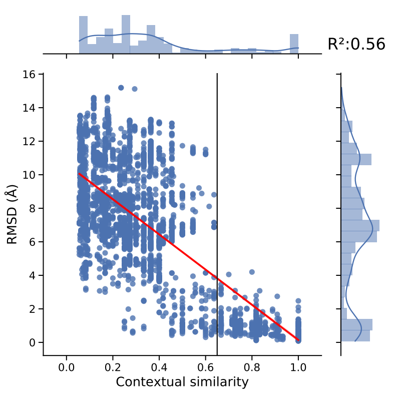
| Orientation | Interacting Edges | Symbol |
|---|---|---|
| *Cis* | Watson–Crick / Watson–Crick (cWW) | -●- |
| *Trans* | Watson–Crick / Watson–Crick (tWW) | -○- |
| *Cis* | Watson–Crick / Hoogsteen (cWH) | ●-■ |
| *Trans* | Watson–Crick / Hoogsteen (tWH) | ○-□ |
| *Cis* | Watson–Crick / Sugar Edge (cWS) | ●- ▶ |
| *Trans* | Watson–Crick / Sugar Edge (tWS) | ○-▷ |
| *Cis* | Hoogsteen / Hoogsteen (cHH) | -■- |
| *Trans* | Hoogsteen / Hoogsteen (tHH) | -□- |
| *Cis* | Hoogsteen / Sugar Edge (cHS) | ■- ▶ |
| *Trans* | Hoogsteen / Sugar Edge (tHS) | □- ▷ |
| *Cis* | Sugar Edge / Sugar Edge (cSS) | -▶- |
| *Trans* | Sugar Edge / Sugar Edge (tSS) | -▷- |

### A.2 Experimental Results

#### A.2.1 Correlation between Graph Similarity and 3D Similarity

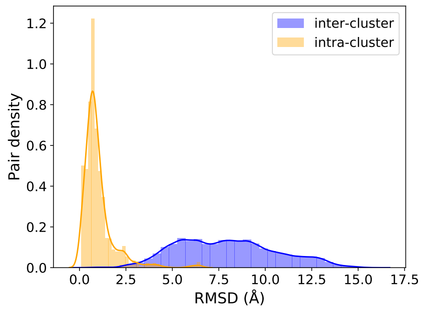


**(a)** A-minor motif.

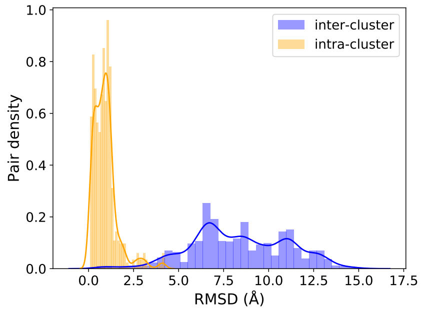


**(b)** G motif.

**Figure 9** Distribution of the RMSD values according to the contextual graph similarity values for the A-minor and the G motif. The linear regression line of the distribution is in red, and the correlation coefficient $R^2$ is indicated. The univariate distributions of RMSD and contextual graph similarity are presented in the margin of the diagram (above for the contextual graph similarity and on the right for the RMSD). The chosen contextual similarity threshold for the clustering for each motif is in black.



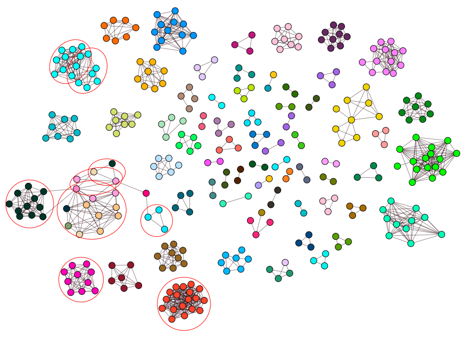
**(a)** A-minor motif
(contextual similarity threshold = 0.75).



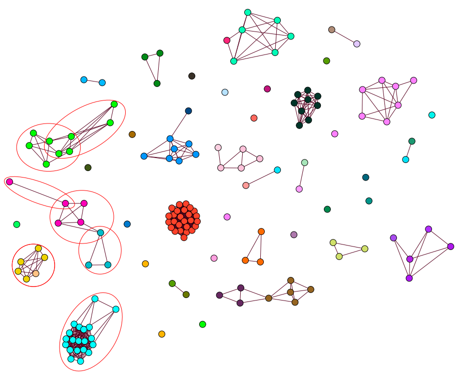
**(b)** G motif
(contextual similarity threshold = 0.65).

**Figure 10** Distribution of the RMSD values intercluster and intracluster, for the two other motifs, with a clustering obtained with the contextual graph similarity thresholds indicated for each case.

## A.2.2 Motif Classification



**(a)** A-minor motif
(contextual similarity threshold = 0.75).



**(b)** G motif
(contextual similarity threshold = 0.65).

**Figure 11** Similarity graphs for the two other motifs, with the similarity threshold indicated in each case. A node corresponds to a motif occurrence, and there is an edge between two nodes if the contextual graph similarity is greater than the indicated threshold. Examples of clusters are circled in red in both graphs. Nodes of the same color correspond to motif occurrences found in the same RNA family.

# Computing Vertex-Edge Cut-Pairs and 2-Edge Cuts in Practice

**Loukas Georgiadis**
Department of Computer Science & Engineering, University of Ioannina, Greece

**Konstantinos Giannis**
Gran Sasso Science Institute, L'Aquila, Italy

**Giuseppe F. Italiano**
LUISS University, Rome, Italy

**Evangelos Kosinas**
Department of Computer Science & Engineering, University of Ioannina, Greece

**Abstract**

We consider two problems regarding the computation of connectivity cuts in undirected graphs, namely identifying vertex-edge cut-pairs and identifying 2-edge cuts, and present an experimental study of efficient algorithms for their computation. In the first problem, we are given a biconnected graph $G$ and our goal is to find all vertices $v$ such that $G \setminus v$ is not 2-edge-connected, while in the second problem, we are given a 2-edge-connected graph $G$ and our goal is to find all edges $e$ such that $G \setminus e$ is not 2-edge-connected. These problems are motivated by the notion of twinless strong connectivity in directed graphs but are also of independent interest. Moreover, the computation of 2-edge cuts is a main step in algorithms that compute the 3-edge-connected components of a graph. In this paper, we present streamlined versions of two recent linear-time algorithms of Georgiadis and Kosinas that compute all vertex-edge cut-pairs and all 2-edge cuts, respectively. We compare the empirical performance of our vertex-edge cut-pairs algorithm with an alternative linear-time method that exploits the structure of the triconnected components of $G$. Also, we compare the empirical performance of our 2-edge cuts algorithm with the algorithm of Tsin, which was reported to be the fastest one among the previously existing for this problem. To that end, we conduct a thorough experimental study to highlight the merits and weaknesses of each technique.

**2012 ACM Subject Classification** Mathematics of computing → Graph algorithms

**Keywords and phrases** 2-Connectivity, Graph Algorithms, Split Components

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.20

**Supplementary Material** The source code, together with some sample input instances is available at *Software (Source Code)*: `https://github.com/KoGi89/VECuts`
archived at `swh:1:snp:1f050f2c505784ef144a790438e918ddb790fb41`

**Funding** Research at the University of Ioannina supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant", Project FANTA (eFficient Algorithms for NeTwork Analysis), number HFRI-FM17-431. G. F. Italiano is partially supported by MIUR, the Italian Ministry for Education, University and Research, under PRIN Project AHeAD (Efficient Algorithms for HArnessing Networked Data)

## 1 Introduction

Let $G = (V, E)$ be a connected undirected graph with $m$ edges and $n$ vertices. An edge $e \in E$ is a *bridge* of $G$ if $G \setminus e$ is not connected. Similarly, a vertex $v \in V$ is an *articulation point* of $G$ if $G \setminus v$ is not connected. Graph $G$ is *biconnected* (resp., 2-*edge-connected*) if it has no articulation points (resp., no bridges). Note that if a graph is biconnected then it is necessarily 2-edge-connected. A 2-*edge cut* of $G$ is a pair of edges $e$ and $f$ such that $G \setminus \{e, f\}$ is not connected. A 3-*edge-connected component* of $G$ is a maximal set $C \subset V$ such that there is no 2-edge cut in $G$ that disconnects any two vertices $u, v \in C$ (i.e., $u$ and $v$ are in the same connected component of $G \setminus \{e, f\}$ for any 2-edge cut $\{e, f\}$). A *separation pair* of $G$ is a 2-vertex cut of $G$, i.e., a pair of vertices $u$ and $v$ such that $G \setminus \{u, v\}$ is not connected. The *triconnected components* of a biconnected graph $G = (V, E)$ is a collection of smaller graphs that describe all the separation pairs in $G$, as well as the partition of the vertex set $V$ induced by each separation pair [8].

Here we consider two problems regarding the computation of connectivity cuts in undirected graphs, namely identifying vertex-edge cut-pairs and identifying 2-edge cuts, and present an experimental study of efficient algorithms for their computation. In the first problem, we are given a biconnected graph $G$ and our goal is to find all vertices $v$ such $G \setminus v$ is not 2-edge-connected, while in the second problem, we are given a 2-edge-connected graph $G$ and our goal is to find all edges $e$ such that $G \setminus e$ is not 2-edge-connected. These problems are motivated by the notion of twinless strong connectivity in directed graphs [6, 9, 14] but are also of independent interest. Moreover, the computation of 2-edge cuts is a main step in algorithms that compute the 3-edge-connected components of a graph [5, 12, 13, 17]. In this paper, we present streamlined versions of two recent linear-time algorithms of Georgiadis and Kosinas [6] that compute all vertex-edge cut-pairs and all 2-edge cuts, respectively. We note that both algorithms of [6] are based on a common framework applied on a depth-first search (DFS) tree structure $T$ of $G$ that yields algorithms that are conceptually simple, asymptotically optimal, and fast in practice. Furthermore, we believe that it may prove useful in solving efficiently other connectivity problems. We compare the empirical performance of our vertex-edge cut-pairs algorithm with an alternative linear-time method that exploits the structure of the triconnected components of $G$, that can be represented efficiently by an SPQR tree [1, 2]. Since SPQR trees can be constructed in linear time [7], this approach implies an alternative linear-time algorithm for computing the vertex-edge cut-pairs of $G$. In order to construct an SPQR tree, however, we need to know the triconnected components of the graph [7], and efficient algorithms that compute triconnected components are considered conceptually complicated, and thus difficult to implement (see, e.g., [4, 7, 8]). Also, we compare the empirical performance of our 2-edge cuts algorithm with the algorithm of Tsin [17], which was previously reported to be the fastest one among the previously existing for this problem. To that end, we conduct a thorough experimental study to highlight the merits and weaknesses of each technique.

## 2 Preliminaries

Recall that a graph is 2-edge-connected if it contains no bridges. For such a graph $G$, we say that an edge $e$ is a *cut-edge* if it forms a 2-edge cut together with some other edge. The framework of Georgiadis and Kosinas [6] relies on a classification of the elements we want to compute (e.g. cut-edges or vertices which belong to a vertex-edge cut), applied on a depth-first search (DFS) tree structure $T$ of $G$. We let $T(v)$ denote the subtree of $T$ rooted at vertex $v$. This classification is based on the distribution of the back-edges of $T$, represented by the sets $B(v)$ of the back-edges that start from $T(v)$ and end in an ancestor of $v$. To see why these sets are useful in determining connectivity relations of $G$, observe that if we remove a vertex $v$ (which is not a leaf or the root) or the tree-edge $(v, p(v))$, that connects $v$ to its parent $p(v)$ in $T$, from $G$, then $T(v)$ remains connected with the rest of the graph only through the back-edges in $B(v)$. Now, we can capture the connectivity information we want from the sets $B(v)$ by considering the higher ends and the lower ends of all back-edges in $B(v)$. Thus we define the nearest common ancestor of the higher ends of all back-edges in $B(v)$, denoted by $M(v)$, and the maximum and minimum lower ends of all back-edges in $B(v)$, denoted by $high(v)$ and $low(v)$, respectively. Using those (and similar) concepts, we can classify the elements we want to compute in such a way that we can provide necessary and sufficient conditions that characterize them and allow us to compute them efficiently.

## Concepts defined on a DFS-tree

We consider a DFS traversal of $G$, starting from an arbitrarily selected vertex $r$, and let $T$ be the resulting DFS tree [16]. A vertex $u$ is an ancestor of a vertex $v$ ($v$ is a descendant of $u$) if the tree path from $r$ to $v$ contains $u$. Thus we consider a vertex to be an ancestor (and, also, a descendant) of itself. We let $p(v)$ denote the parent of a vertex $v$ in $T$. If $u$ is a descendant of $v$ in $T$, we denote the set of vertices of the simple tree path from $u$ to $v$ as $T[u, v]$. The expressions $T[u, v)$ and $T(u, v]$ have the obvious meaning (i.e., the vertex on the side of the parenthesis is excluded from the tree path). Recall that $T(v)$ denotes the subtree of $T$ rooted at vertex $v$. We identify vertices in $G$ by their DFS number, i.e., the order in which they were discovered by the search. Hence, $u \leq v$ means that vertex $u$ was discovered before $v$. The edges of $T$ are called tree-edges, and the edges of $G$ that are not tree-edges are called back-edges, as their endpoints are related as ancestor and descendant on $T$. We denote the collection of all back-edges as $B$. When we write $(u, v)$ to denote a back-edge, we always mean that $v \leq u$, i.e., $u$ is an descendant of $v$ in $T$. The framework of Georgiadis and Kosinas [6], referred to as GK hereafter, uses the following key concepts that are defined on $T$:

- $B(v) := \{(x, y) \in B \mid x \in T(v) \text{ and } y < v\}$, the set of all back-edges that start from $T(v)$ and end in a proper ancestor of $v$.
- $B_p(v) := \{(x, y) \in B \mid x \in T(v) \text{ and } y < p(v)\}$, the set of all back-edges that start from $T(v)$ and end in a proper ancestor of $p(v)$.
- $l(v) := min\{\{v\} \cup \{y \mid (v, y) \in B(v)\}\}$, the lowest vertex that is connected with a back-edge with $v$ (or $v$ if there is no back-edge $(v, y)$).
- $low(v) := min\{y \mid (x, y) \in B(v)\}$, the lowest lower end of all back-edges in $B(v)$.
- $high(v) := max\{y \mid (x, y) \in B(v)\}$, the highest lower end of all back-edges in $B(v)$.
- $high_p(v) := max\{y \mid (x, y) \in B_p(v)\}$, the highest lower end of all back-edges in $B_p(v)$.
- $M(v) := nca\{x \mid (x, y) \in B(v)\}$, the nearest common ancestor of the higher ends of all back-edges in $B(v)$.
- $M_p(v) := nca\{x \mid (x, y) \in B_p(v)\}$, the nearest common ancestor of the higher ends of all back-edges in $B_p(v)$.
- $b\_count(v) := \#B(v)$, the number of elements of $B(v)$.
- $b_p\_count(v) := \#B_p(v)$, the number of elements of $B_p(v)$.
- $up(v) := \#\{(x, p(v)) \mid x \in T(v)\}$, the number of back-edges that start from $T(v)$ and end in the parent of $v$.

$B(v)$, $l(v)$, $low(v)$, $high(v)$, $M(v)$, $b\_count(v)$, and $up(v)$ are defined for all vertices $v \neq r$; similarly, $B_p(v)$, $high_p(v)$, $M_p(v)$, and $b_p\_count(v)$ are defined for all vertices $v \notin \{r, r_c\}$, where $r_c$ is the unique child of $r$, if $G$ is biconnected. Except for $B(v)$ and $B_p(v)$, all these parameters can be computed in total linear-time, for all the vertices on which they are defined (see [6], and also Algorithms 1 and 3 in Appendix A).

## 3 Computing vertex-edge cut-pairs in linear time

Here we present an overview of linear-time algorithms for computing the vertex-edge cut-pairs of a biconnected graph $G = (V, E)$. All algorithms compute, for each vertex $v \in V$, the number of edges $e \in E$ such that $G \setminus \{v, e\}$ is not connected. The corresponding values are stored in variables $count(v)$. We first describe the algorithm of Georgiadis and Kosinas [6], which operates on a DFS tree of $T$. Next, we provide a streamlined version that enhances its performance in practice. Finally, we describe how to compute the $count(v)$ values using a SPQR tree of $G$, and describe a simplification of this approach that only computes the relevant nodes of the SPQR tree.

### 3.1 Computing vertex-edge cut-pairs via the GK framework

We give an overview of the algorithm in [6] for computing all vertices that belong to a vertex-edge cut in a biconnected graph $G$. This algorithm computes, for every vertex $v$, the number of edges $e$ such that $G \setminus \{v, e\}$ is not connected. It works by classifying the vertex-edge cuts on the DFS tree in such a way that we can provide an efficient method to count the number of vertex-edge cuts of each type.

Let $T$ be a DFS-tree of $G$ rooted at $r$, and let $\{v, e\}$ be a vertex-edge cut-pair. We distinguish three cases, depending on the location of $e$ relative to $v$ on $T$: $e$ can either be a back-edge, or a tree-edge of the form $(u, p(u))$, with $u$ a proper ancestor of $v$, or a tree-edge of the form $(u, p(u))$, with $u$ a proper descendant of a child of $v$.

If $e$ is a back-edge, then there exists a child $c$ of $v$, such that $e$ connects $T(c)$ with $T(v, r]$ and is the only back-edge with this property. Thus, for every vertex $v$, the number of cut-pairs of the form $\{v, e\}$, where $e$ is a back-edge, cannot be greater than the number of children of $v$, and we can find all these cut-pairs explicitly: We only have to count, for every vertex $v$, the number of its children $c$ that have $b_p\_count(c) = 1$. All $b_p\_count(c)$, for every vertex $c$, can be computed during the depth-first search (see Algorithm 1 in Appendix A). If for a vertex $c$ we have $b_p\_count(c) = 1$, then $\{p(c), (M_p(c), low(c))\}$ is a cut-pair.

Now, if $e$ is a tree-edge of the form $(u, p(u))$, with $u$ a proper ancestor of $v$, then every back-edge that starts from $T(u)$ and ends in a proper ancestor of $u$ must start from a descendant of $v$. This means that $M(u)$ is a descendant of $v$, and we further distinguish two cases, depending on whether $M(u) = v$ or $M(u)$ is a proper descendant of $v$. In the first case, $u$ has the property that, for every child $c$ of $v$, either $u \leq low(c)$ or $u > high_p(c)$; in other words, $u$ does not belong to any set of the form $T[high_p(c), low(c))$, for any child $c$ of $v$. (And conversely: if $u$ has this property, and $M(u) = v$, then $\{v, (u, p(u))\}$ is a cut-pair.) Thus, we can find all vertex-edge cut-pairs of this type explicitly: we only have to find, for every vertex $v$, all elements in $M^{-1}(v)$ that do not belong to any set of the form $T[high_p(c), low(c))$, for any child $c$ of $v$. Now, if $M(u)$ is a proper descendant of $v$, it is a descendant of a child $c$ of $v$. In this case, we have $M_p(c) = M(u)$, and every back-edge that starts from $T(c)$ and ends in a proper ancestor of $v = p(c)$ must end in a proper ancestor of $u$, and therefore $high_p(c) < u$. (The converse is also true: if $u$ is a proper ancestor of $p(c)$ such that $M_p(c) = M(u)$ and $high_p(c) < u$, then $\{p(c), (u, p(u))\}$ is a cut-pair.) Thus, to count all vertex-edge cut-pairs of this type, it is sufficient to focus our attention on the lists $M_p^{-1}(m)$ and $M^{-1}(m)$, for every vertex $m$, and count all pairs $(c, u) \in M_p^{-1}(m) \times M^{-1}(m)$, such that $u$ is a proper ancestor of $p(c)$ with $high_p(c) < u$. To do this efficiently, we exploit the following fact: If $c$ is in $M_p^{-1}(m)$, and $U(c)$ is the set of all vertices $u$ in $M^{-1}(m)$ such that $u$ is a proper ancestor of $p(c)$ with $high_p(c) < u$, then, if $c'$ is also in $M_p^{-1}(m)$ and $c' \in T[c, high_p(c))$, we have $U(c') = U(c) \cap T(c', high_p(c))$. (For details, see Algorithm "$M(u) > v$" in [6].)

Finally, if $e$ is a tree-edge of the form $(u, p(u))$, with $u$ a proper descendant of a child of $v$, then every back-edge that starts from $T(u)$ and ends in a proper ancestor of $u$ must end in an ancestor of $v$. This means that $high(u)$ is an ancestor of $v$, and we further distinguish two cases, depending on whether $high(u) = v$ or $high(u)$ is a proper ancestor of $v$. In the first case, we can find all cut-pairs explicitly: we only have to find, for every vertex $v$, all $u$ in $high^{-1}(v)$ that are not children of $v$ and are such that either $low(u) = v$ or $low(u) < v$ and $M_p(c)$ is in $T(u)$, where $c$ is the child of $v$ which is an ancestor of $u$ (see Algorithm "$high(u) = v$" in [6]). If $high(u)$ is a proper ancestor of $v$, then $M(u) = M_p(c)$, where $c$ is the child of $v$ which is an ancestor of $u$. (Conversely: if $u$ and $c$ are such that $u$ is a proper descendant of $c$ with $M(u) = M_p(c)$ and $high(u) < p(c)$, then $\{p(c), (u, p(u))\}$ is a cut-pair.) Thus, to count all vertex-edge cut-pairs of this type, it is sufficient to focus our attention on the lists $M_p^{-1}(m)$ and $M^{-1}(m)$, for every vertex $m$, and count all pairs $(c, u) \in M_p^{-1}(m) \times M^{-1}(m)$, such that $u$ is a proper descendant of $c$ with $high(u) < p(c)$. To do this efficiently, we exploit the following fact: If $c$ is in $M_p^{-1}(m)$ and $U(c)$ is the set of all vertices $u$ in $M^{-1}(m)$ such that $u$ is a proper descendant of $c$ with $high(u) < p(c)$, then all $u$ in $U(c)$ have the same *high* point, call it $h$, and, if $c'$ is also in $M_p^{-1}(m)$ and such that $c \geq c'$ and $h < p(c')$, then $U(c') = U(c) \cup (T[c, c') \cap M^{-1}(u))$.

We refer to the algorithm of [6] as GK-VE.

## Streamlined version

Now we describe our improvements that both simplify the algorithm of [6] and also make it faster in practice. First, in order to compute all $b_p\_count(v)$, [6] suggests a recursive algorithm, which depends upon a specific sorting of the list of back-edges. Here, we observe that we can compute these values directly during the DFS of $G$, together with all $up(v)$, as shown in Algorithm 1 (see Appendix A). This works as follows. By definition, $b_p\_count(v)$ is the number of back-edges of the form $(x, y)$, where $x$ is a descendant of $v$ and $y$ is a proper ancestor of $p(v)$. Therefore, $b_p\_count(v) = b_p\_count(c_1) + \ldots + b_p\_count(c_k) + \#\{(v, y) \in B(v)\} - \#\{(x, p(v)) \mid x \in T(v)\}$, where $c_1, \ldots, c_k$ are the children of $v$ (if it has any). Thus, when we process a vertex $v$ and $u$ is in the adjacency list of $v$, we set $b_p\_count(v) \leftarrow b_p\_count(v) + b_p\_count(u)$ if $u$ is a child of $v$, or $b_p\_count(v) \leftarrow b_p\_count(v) + 1$ if $u$ is an ancestor of $v$ but not its parent. Now, to compute all $up(v) := \#\{(x, p(v)) \mid x \in T(v)\}$, we keep a variable $tempChild(v)$, for every vertex $v$, which is set to be the child of $v$ in which we have currently descended during the DFS. Then, when we process a vertex $x$ and $v$ is in the adjacency list of $x$, and also $v$ is an ancestor of $x$ but not its parent, we set $up(tempChild(v)) \leftarrow up(tempChild(v)) + 1$.

A second important improvement comes from the fact that [6] uses both the $high(v)$ and the $high_p(v)$ values, while we can observe that it suffices to use only the latter. Indeed, if $\{v, (u, p(u))\}$ is a vertex-edge cut-pair such that $u$ is a descendant of $v$, then $u$ is a proper descendant of a child $c$ of $v$ and $high(u) \leq v = p(c)$; thus we have $high(u) = high_p(u)$, since $high(u) < c \leq p(u)$. Then, we only have to make sure that every time we discover a vertex-edge cut $\{v, (u, p(u))\}$ of this type, we have $high(u) = high_p(u)$. This is the case if and only if there is no back-edge $(x, p(u))$ with $x \in T(u)$, which can be checked simply by testing whether $up(u) = 0$.

We refer to the above version as GK-VE-S.

## 3.2 Computing vertex-edge cut-pairs via SPQR trees

Here we describe how to compute the number of vertex-edge cut-pairs in linear time via SPQR trees [1, 2]. An SPQR tree $\mathcal{T}$ for a biconnected graph $G$ represents the triconnected components of $G$. Each node $\alpha \in \mathcal{T}$ is associated with an undirected graph or multigraph $G_\alpha$. Each vertex of $G_\alpha$ corresponds to a vertex of the original graph $G$. An edge of $G_\alpha$ is either a *virtual edge* that corresponds to a separation pair of $G$, or a *real edge* that corresponds to an edge of the original graph $G$. The node $\alpha$, and the graph $G_\alpha$ associated with it, has one of the following types:

- If $\alpha$ is an $S$-node, then $G_\alpha$ is a cycle graph with three or more vertices and edges.
- If $\alpha$ is a $P$-node, then $G_\alpha$ is a multigraph with two vertices and at least 3 parallel edges.
- If $\alpha$ is a $Q$-node, then $G_\alpha$ is a single real edge.
- If $\alpha$ is an $R$-node, then $G_\alpha$ is a simple triconnected graph.

Each edge $\{\alpha, \beta\}$ between two nodes of the SPQR tree is associated with two virtual edges, where one is an edge in $G_\alpha$ and the other is an edge in $G_\beta$. If $\{u, v\}$ is a separation pair in $G$, then one of the following cases applies (see, e.g., [18]):

**(a)** $u$ and $v$ are the endpoints of a virtual edge in the graph $G_\alpha$ associated with an $R$-node $\alpha$ of $\mathcal{T}$.

**(b)** $u$ and $v$ are vertices in the graph $G_\alpha$ associated with a $P$-node $\alpha$ of $\mathcal{T}$.

**(c)** $u$ and $v$ are vertices in the graph $G_\alpha$ associated with an $S$-node $\alpha$ of $\mathcal{T}$, such that either $u$ and $v$ are not adjacent, or the edge $\{u, v\}$ is virtual.

In case (c), if $\{u, v\}$ is a virtual edge, then $u$ and $v$ also belong to a $P$-node or an $R$-node. If $u$ and $v$ are not adjacent then $G \setminus \{u, v\}$ consists of two components that are represented by two paths of the cycle graph $G_\alpha$ associated with the $S$-node $\alpha$ and with the SPQR tree nodes attached to those two paths. Let $e = \{x, y\}$ be an edge of $G$ such that $\{v, e\}$ is a vertex-edge cut-pair of $G$. Then, $\mathcal{T}$ must contain an $S$-node $\alpha$ such that $v$, $x$ and $y$ are vertices of $G_\alpha$ and $\{x, y\}$ is not a virtual edge.

The above observation implies that we can use $\mathcal{T}$ to identify (and count) all vertex-edge cut-pairs of $G$. We do that as follows. We initialize $count(v) \leftarrow 0$ for all $v \in V$, and process the $S$-nodes of $\mathcal{T}$ individually. For each $S$-node $\alpha$ we count the number $m_\alpha$ of the real edges of $G_\alpha$. Then, for each vertex $v \in V(G_\alpha)$, we set $count(v)$ equal to $m_\alpha - |\{e \in E : e \text{ is adjacent to } v\}|$.

Gutwenger and P. Mutzel [7] showed that an SPQR tree can be constructed in linear time, by extending the triconnected components algorithm of Hopcroft and Tarjan. Moreover, given $\mathcal{T}$ it is straightforward to compute $count(v)$ in $O(n)$ time, for all vertices $v$. We refer to this algorithm as SPQR-VE.

We also considered a variant that avoids the computation of a complete SPQR tree. Since we only care about $S$-nodes, it suffices to compute only these nodes from the partition of the graph into *split components*. These components are formed during the execution of the Hopcroft-Tarjan algorithm as follows. When the algorithm finds a pair of separating vertices $u$ and $v$, it splits the graph at these two vertices into two smaller graphs, and adds the virtual edge $\{u, v\}$ in both graphs. The split components of $G$ are the graphs that are formed when we repeat this process until no more separating pairs exist. (Note that this partition is not uniquely defined.) A split component can be of three types: a $P$-component that consist of two vertices joined with multiple edges, an $S$-component that forms a triangle, or an $R$-component that is any other split component. Then, the $S$-nodes of the SPQR tree are formed by merging $S$-components that share a virtual edge. After we have computed just the $S$-nodes of the SPQR tree, we can identify the vertex-edge cut-pairs of $G$ as above. We refer to this algorithm as Split-VE.

## 4 Finding all cut-edges and computing the number of 2-cuts

Here we present an overview of linear-time algorithms for computing the 2-edge cuts of a 2-edge-connected graph $G = (V, E)$. These algorithms compute the cut-edges of $G$ (i.e., the edges that form a 2-edge cut together with some other edge). First, we describe the algorithm of [6], and our streamlined version of it. Then, we give an overview of the algorithm of Tsin [17], which was previously reported to be the fastest one among the previously existing for this problem.

### 4.1 Computing 2-edge cuts via the GK framework

The algorithm in [6] works on a DFS tree $T$ of $G$ rooted at $r$. It distinguishes two types of cut-pairs: those consisting of a back-edge and a tree-edge, and those consisting of two tree-edges. The first case is very easy to handle, since we only have to find the vertices $v \neq r$ that have $b\_count(v) = 1$, and mark the edges $(v, p(v))$ and $(M(v), low(v))$ as cut-edges. In the case where $(u, p(u))$, $(v, p(v))$ are two tree-edges, [6] proved the following condition: (1) $\{(u, p(u)), (v, p(v))\}$ is a cut-pair if and only if $M(u) = M(v)$ and $high(u) = high(v)$. Now, let $m$ be a vertex and $u_1, \ldots, u_k$ all the vertices in $M^{-1}(m)$ ordered decreasingly. Then we have $high(u_1) \geq \ldots \geq high(u_k)$. Thus, by (1), it is sufficient it is sufficient to traverse the decreasingly sorted list $M^{-1}(m)$ from the greatest to the lowest element, and mark the edges $(u, p(u))$ and $(v, p(v))$ that satisfy the following condition: (2) $u$ and $v$ are consecutive vertices in $M^{-1}(m)$ such that $high(u) = high(v)$.

We refer to this algorithm of [6] as GK-2E. We also note that a simple extension of this algorithm computes the 3-edge-connected components of $G$ as in [17].

#### Streamlined version

Algorithm GK-2E depends on the computation of the *high* points of all vertices $v \neq r$, so that it can check condition (2). Here, however, we observe that we can skip this computation thanks to the following Lemma.

▶ **Lemma 1.** *Let $u, v$ be two vertices ($\neq r$) with $M(u) = M(v)$. Then $high(u) = high(v)$ if and only if $b\_count(u) = b\_count(v)$.*

**Proof.** ($\Rightarrow$) Let $(x, y)$ be a back-edge such that $x$ is a descendant of $u$ and $y$ is a proper ancestor of $u$. Since $M(u) = M(v)$, we have that $x$ is a descendant of $v$. Furthermore, since $y \leq high(u) = high(v) < v$, we have that $y$ is a proper ancestor of $v$. This shows that $B(u) \subseteq B(v)$, and so we have $b\_count(u) \leq b\_count(v)$. Now, with a reversal of the roles of $u$ and $v$, we also get $b\_count(v) \leq b\_count(u)$. We conclude that $b\_count(u) = b\_count(v)$.

($\Leftarrow$) Since $u$ and $v$ have a common descendant, we can assume, without loss of generality, that $u$ is a descendant of $v$. Let $(x, y)$ be a back-edge such that $x$ is a descendant of $v$ and $y$ is a proper ancestor of $v$. Since $M(u) = M(v)$, we have that $x$ is a descendant of $u$. Furthermore, since $v$ is an ancestor of $u$, we have that $y$ is a proper ancestor of $u$. This shows that $B(v) \subseteq B(u)$. Since $b\_count(u) = b\_count(v)$, it must be the case that $B(v) = B(u)$. Thus we have $high(u) = high(v)$. ◀

Thus we can test condition (2) by checking whether $b\_count(u) = b\_count(v)$, instead of $high(u) = high(v)$. In this way, we can avoid the computation of all *high* points (during which we have to process all the back-edges), and so we get an algorithm which is about twice as fast as the original.

Now, in order to compute the number of 2-edge cuts, we first observe that those consisting of a back-edge and a tree-edge can be found explicitly, since their number can be at most $n-1$ (as they correspond to the vertices $v \neq r$ that have $b\_count(v) = 1$). Now, let $(v, p(v))$ be a tree-edge and $(u_1, p(u_1)), \ldots, (u_k, p(u_k))$ all the tree-edges which form a cut-pair with $(v, p(v))$. Then (1) implies that every $(u_i, p(u_i))$, for $i \in \{1, \ldots, k\}$, forms a cut-pair with $(u_j, p(u_j))$, for every $j \in \{1, \ldots, k\} \setminus \{i\}$. Furthermore, these are all the tree-edges (plus $(v, p(v))$) with which $(u_i, p(u_i))$ forms a cut-pair. Thus, to count the number of those cut-pairs efficiently, we can work as follows. We traverse the decreasingly sorted list $M^{-1}(m)$, for every vertex $m$, until we reach a vertex $v$ such that $(v, p(v))$ and $(u_1, p(u_1))$ is a cut-pair, where $u_1$ is the successor of $v$ in $M^{-1}(m)$. Then we keep traversing the list $M^{-1}(m)$, until we reach the lowest vertex $u_k$ such that $(u_k, p(u_k))$ forms a cut-pair with $(v, p(v))$ (i.e. that satisfies $b\_count(u_k) = b\_count(v)$). Then we add the quantity $k(k+1)/2$ to the number of 2-edge cuts, and we repeat the same process until we reach the end of $M^{-1}(m)$.

The entire algorithm which computes the cut-edges and the number of 2-edge cuts is shown in Algorithm 2 (see Appendix A). Variable $nextM(v)$ denotes the successor of $v$ in the decreasingly sorted list $M^{-1}(M(v))$ (or the end-of-list symbol $\emptyset$). It can be computed during the calculation of all $M(v)$, as shown in Algorithm 3 (see Appendix A).

We refer to the above version as GK-2E-S.

## 4.2 Tsin's algorithm

Tsin's algorithm [17] finds the 3-edge-connected components of a 2-edge-connected graph $G$. The algorithm consists of two parts, and the first one determines the set $E_{cut}$ that contains all the edges belonging to a cut-pair. In the second part, it processes the edge set $E_{cut}$ in order to form the 3-edge-connected components of $G$.

Here we describe the first part of the algorithm that is relevant to our problem. The algorithm performs a depth-first traversal in $G$, and it creates a DFS tree $T$ while separating the edges of $G$ into two sets, the tree-edges belonging in $T$ and the set of back-edges which contains all other edges. Tsin provides the following key definition. A *generator* is a cut-edge $e = (x, y)$, where $e$ is either a back-edge or a tree-edge, and there is no other tree edge in $T(y)$ or back-edge having an end-vertex in $T(y)$ that forms a cut-pair with $e$. It is shown that every edge in $E_{cut}$ belongs to a cut-pair containing a generator and therefore it suffices to determine the subset of cut-pairs that contain a generator. Moreover, [17] shows that the cut-pairs have a nesting structure, which allows the algorithm to use stacks in order to determine the cut-pairs. Specifically, each vertex $v$ is associated with a stack *stack(v)* that stores entries of the form $[(x, y), T[q, p]]$, where the edge $(x, y)$ is a generator or a potential generator and $T[q, p]$ is a path with edges that may form cut-pairs with $(x, y)$.

The algorithm distinguishes two cases depending on whether the current vertex $v$ that we traverse is a leaf of $T$ or not. In the former case (where $v$ is a leaf), we determine if $v$ is an ending point of an edge that is a generator, and we push a corresponding entry to *stack(v)* if needed. Otherwise, if $v$ is not a leaf, when the traversal returns from a child $w$ of $v$, we check the top of *stack(w)* to determine if we have found a new cut-pair and update *stack(w)*. When we finish this check for all the children of $v$, we update *stack(v)* and backtrack. Finally, when the traversal returns to the root of $T$, all edges belonging in a cut-pair form the set $E_{cut}$.

We refer to this algorithm as Tsin-2E.

## 5 Empirical Analysis

We wrote our implementations in `C++`, using `Visual Studio Compiler x64` with maximum optimization to favor speed (flag `/O2`) to compile the code. For computing the SPQR tree in algorithm SPQR-VE, we use the linear-time implementation of Gutwenger and Mutzel [7], which is available within the Open Graph Drawing Framework (OGDF) [3]. Similarly, for computing the split components in algorithm Split-VE, we use the implementation of the Hopcroft-Tarjan algorithm [8] provided in OGDF. In order to make a fair comparison among algorithms GK-VE-S, SPQR-VE and Split-VE, we also provide an OGDF-based implementation of GK-VE-S that we refer to as GK-VE-SF. Specifically, GK-VE-SF uses the OGDF representation of graphs, as well basic data structures such as arrays, lists and stacks.

Our main experiments were performed using the following setting: (I) A Dell Precision Tower 7820 CTO Base machine running Red Hat Enterprise 6, equipped with an Intel Xeon E5-2430 2.5GHz processor with 15MB L3 cache and 96GB DDR4 RAM at 2666 MHz. For the OGDF-based implementations we used a less powerful setting: (II) A Dell G5 15 Laptop running Windows 10 (Home 64 bit), equipped with an 8th Generation Intel Core i5 8300H 4GHz processor with 8 MB L3 cache and 8GB DDR4 RAM at 2,666 MHz. We used setting (II) because we did not have physical access to the server of (I) in order to install OGDF. OGDF also requires `CMake` (version 3.1+) a C++11 compliant compiler and GNU Make. We observed that the OGDF library was not able to compute the SPQR trees of graphs with more than $\sim$ 45000 vertices. (This problem was also reported in [10].) This is due to the use of recursion in three routines (DFS, pathFind, and pathSearch) in the implementation of the Hopcroft-Tarjan triconnected components algorithm (which is contained in file "Triconnectivity.cpp"). To fix this, we replaced the recursion with stacks. After this modification, we were able to handle graphs with millions of vertices and edges.

We did not use any parallelization in either setting, and each algorithm ran on a single core. We report CPU times measured with the `std::chrono::steady_clock::now` function.

### Real-world graphs

Table 1 shows some statistics of the graphs used in our experimental evaluation. We include both undirected and directed graphs, since one of our motivating applications (twinless strong connectivity) deals with directed graphs. We convert these directed graphs to undirected by ignoring edge directions. Furthermore, we augment these graphs so that they become biconnected, by applying the following procedure. Let $G$ be the input graph. Firstly, we compute the connected components $C_1, \ldots, C_k$ of $G$, and we join them in a path, by adding an edge connecting a vertex in $C_i$ to a vertex in $C_{i+1}$, for every $i \in \{1, \ldots, k-1\}$. Let $G'$ be the resulting graph. Then, we compute the leaves $B_1, \ldots, B_l$ of the tree representation of the biconnected components of $G'$, and we connect them in a path. To that end, for every $i \in \{1, \ldots, l-1\}$, we add an edge connecting a vertex $x_i \in B_i$ to a vertex $x_{i+1} \in B_{i+1}$, such that neither $x_i$ nor $x_{i+1}$ is an articulation point.

The corresponding experimental results for setting (I) are given in Table 2 and plotted in Figure 1, while Table 3 shows the results using setting (II). (The memory consumption of the algorithms tested in setting (I) is reported in Table 7 in Appendix B.) First, we make some remarks about the performance of our improved versions of the algorithms of [6]. In Table 2 we observe that our streamlined version GK-VE-S is consistently faster than GK-VE, and improves its running time by 16% up to 44%. The improvement obtained by our streamlined version is even more prominent in the case of 2-edge cuts. Specifically, GK-2E-S is consistently faster than GK-2E by 33% up to 47%.

**Table 1** Graph instances used in the experiments. The original graphs are taken from [11] and [15], and were converted to biconnected graphs by adding $m_e$ extra edges. The graph categories are: web graph (WG), network with ground truth communities (NGT), dynamic network (DN), collaboration network (CLN), heterogeneous network (HN), recommendation network (RN), dimacs10 (D10), interaction network (IN) and brain network (BN) Undirected graphs are indicated by U and directed graphs are indicated by D. The number of vertices $n$ and edges $m$ refer to the produced instances; $n_c$ is the number of vertices that form a vertex-edge cut pair with at least one edge.

| | | **Graph Details** | | | | |
|---|---|---|---|---|---|---|
| **Graph** | **Type** | $n$ | $m$ | $m_e$ | $n_c$ | **Reference** |
| Amazon0302 | WG (D) | 262111 | 906735 | 6946 | 12438 | SNAP [11] |
| com-amazon | NGT (U) | 548551 | 1168192 | 242323 | 267685 | SNAP [11] |
| com-dblp | NGT (U) | 425957 | 1219125 | 169262 | 166497 | SNAP [11] |
| web-NotreDame | WG (D) | 325729 | 1252708 | 162601 | 16475 | SNAP [11] |
| web-Stanford | WG (D) | 281903 | 2008593 | 15960 | 25658 | SNAP [11] |
| Amazon0601 | WG (D) | 403394 | 2455710 | 12305 | 27772 | SNAP [11] |
| ia-yahoo-messages | DN (U) | 3157315 | 3745264 | 3157299 | 3073682 | NR [15] |
| web-Google | WG (D) | 916428 | 4523768 | 201720 | 127081 | SNAP [11] |
| ca-cit-HepTh | CLN (D) | 2673133 | 5117848 | 2673049 | 2665102 | NR [15] |
| cit-HepTh | CLN (U) | 2673133 | 5117859 | 2673060 | 2665080 | NR [15] |
| visualise-us | HN (U) | 3247673 | 6495338 | 3247664 | 2669435 | NR [15] |
| web-BerkStan | WG (D) | 685230 | 6693612 | 44143 | 50673 | SNAP [11] |
| ca-IMDB | CLN (U) | 3782456 | 7564896 | 3782448 | 2902605 | NR [15] |
| ca-cit-HepPh | CLN (D) | 4596803 | 7745139 | 4596691 | 4584870 | NR [15] |
| cit-HepPh | CLN (U) | 4596803 | 7745148 | 4596700 | 4584710 | NR [15] |
| amazon-ratings | RN (U) | 5838027 | 11581127 | 5837994 | 3708476 | NR [15] |
| hugetrace-00000 | D10 (U) | 6879133 | 13758157 | 6879023 | 2307134 | NR [15] |
| rgg-n-2-20-s0 | D10 (U) | 6891620 | 13783038 | 6891417 | 5859427 | NR [15] |
| wiki-user-edits-page | IN (U) | 8998641 | 14571201 | 8998616 | 6930358 | NR [15] |
| hugetric-00010 | D10 (U) | 9885854 | 19771559 | 9885704 | 3309476 | NR [15] |
| delaunay-n22 | D10 (U) | 12582869 | 25165521 | 12582651 | 8404913 | NR [15] |
| co-papers-dblp | D10 (U) | 15245729 | 30491160 | 15245430 | 14721451 | NR [15] |
| co-papers-citeseer | D10 (U) | 16036720 | 32073082 | 16036361 | 15618987 | NR [15] |
| packing-b050 | D10 (U) | 17488243 | 34975965 | 17487763 | 26 | NR [15] |
| human-Jung2015 | BN (U) | 1827166 | 52455284 | 1827112 | 1463704 | NR [15] |
| rgg-n-2-23-s0 | D10 (U) | 8388608 | 71889991 | 8388597 | 2 | NR [15] |

Next, we compare GK-2E-S to Tsin-2E. Here, we observe that the two algorithm have similar performance, with GK-2E-S being 4% up to 24% faster than Tsin-2E on all but two instances. Moreover, in all instances GK-2E-S uses less memory than Tsin-2E. (See Table 7 in Appendix B.)

Now we turn to the OGDF-based implementations evaluated in setting (II). In Table 3 we report the running times and memory consumption of the corresponding algorithms. Notice that since setting (II) had limited RAM memory (8GB), we only included instances such that the memory consumption of all algorithms did not exceed the available capacity. First, we compare the performance of SPQR-VE and Split-VE. Here we notice that the computation of the full SPQR tree incurs a small overhead over the computation of just the split components (followed by merging $S$-components that share a virtual edge to form the $S$-nodes). Indeed, Split-VE is about 17% faster than SPQR-VE and consumes about 14% less memory on average. On the other hand, it is evident that both SPQR-VE and Split-VE are not competitive against GK-VE-SF. Indeed, GK-VE-SF is faster than Split-VE (resp., SPQR-VE) by a factor larger than 4 (resp., 4.5) and requires 1.9 (resp., 2.26) times less memory on average.

Finally, by comparing Table 3 to Table 7 (given in Appendix B), it is worth noticing that GK-VE-SF requires 6 times more memory space than GK-VE-S on average. This is because GK-VE-SF is implemented using the dynamic data structures of OGDF, in order to make a fair comparison with SPQR-VE and Split-VE. On the other hand, our implementation of GK-VE-S (as well as of all other algorithms tested in setting (I)), uses a much more compact representation of the input graph with just 2 static arrays. We remark that it is not possible to employ this compact representation in SPQR-VE and Split-VE, since for the computation of the split components we need to manipulate the adjacency lists and insert virtual edges.

**Table 2** Running times in seconds for the graphs of Table 1 in experimental setting (I). The best results in each row are marked in bold.

| | 2-edge cuts | | | vertex-edge cuts | |
|---|---|---|---|---|---|
| **Graph** | GK-2E | GK-2E-S | Tsin-2E | GK-VE | GK-VE-S |
| Amazon0302 | 0.16 | **0.10** | 0.13 | 0.33 | **0.25** |
| com-amazon | 0.33 | **0.20** | 0.23 | 0.66 | **0.55** |
| com-dblp | 0.26 | **0.16** | 0.18 | 0.54 | **0.42** |
| web-NotreDame | 0.14 | **0.08** | 0.09 | 0.29 | **0.19** |
| web-Stanford | 0.25 | **0.13** | 0.16 | 0.51 | **0.34** |
| Amazon0601 | 0.37 | **0.21** | 0.25 | 0.77 | **0.54** |
| ia-yahoo-messages | 1.22 | **0.82** | 0.88 | 2.57 | **2.15** |
| web-Google | 0.95 | **0.52** | 0.60 | 1.92 | **1.37** |
| ca-cit-HepTh | 1.08 | **0.70** | 0.81 | 2.35 | **1.80** |
| cit-HepTh | 1.08 | **0.70** | 0.82 | 2.34 | **1.78** |
| visualise-us | 1.60 | **0.99** | 1.14 | 3.29 | **2.54** |
| web-BerkStan | 0.52 | **0.28** | 0.30 | 1.12 | **0.64** |
| ca-IMDB | 2.15 | **1.33** | 1.44 | 4.41 | **3.43** |
| ca-cit-HepPh | 1.78 | **1.16** | 1.36 | 3.87 | **3.01** |
| cit-HepPh | 1.78 | **1.16** | 1.33 | 3.84 | **2.98** |
| amazon-ratings | 4.03 | **2.46** | 2.65 | 8.32 | **6.43** |
| hugetrace-00000 | 5.74 | **3.52** | 3.67 | 12.66 | **9.35** |
| rgg-n-2-20-s0 | 3.57 | **2.22** | 2.55 | 7.34 | **5.64** |
| wiki-user-edits-page | 4.42 | **2.79** | 3.24 | 9.37 | **7.36** |
| hugetric-00010 | 8.75 | 5.37 | **5.33** | 21.19 | **14.39** |
| delaunay-n22 | 8.55 | **5.34** | 6.41 | 19.49 | **14.12** |
| co-papers-dblp | 6.32 | **4.08** | 4.53 | 13.58 | **10.32** |
| co-papers-citeseer | 6.51 | **4.18** | 4.73 | 14.15 | **10.75** |
| packing-b050 | 8.87 | **5.66** | 6.42 | 7.86 | **5.21** |
| human-Jung2015 | 4.98 | **2.65** | 3.53 | 11.08 | **6.16** |
| rgg-n-2-23-s0 | 19.03 | 11.34 | **10.56** | 42.12 | **29.98** |

## Artificial graphs

In order to test the robustness of our algorithms and to obtain a better view of their relative performance, we also conducted experiments with artificial graphs that we produced in the following manner. We construct a biconnected graph by connecting cyclic and complete graphs in a tree-like structure. This allows us to control the number of vertex-edge and 2-edge cuts, as well as the density of the graph. Our generator receives as inputs the number of cyclic and complete graphs, denoted by $C$ and $K$ respectively, and the number of vertices

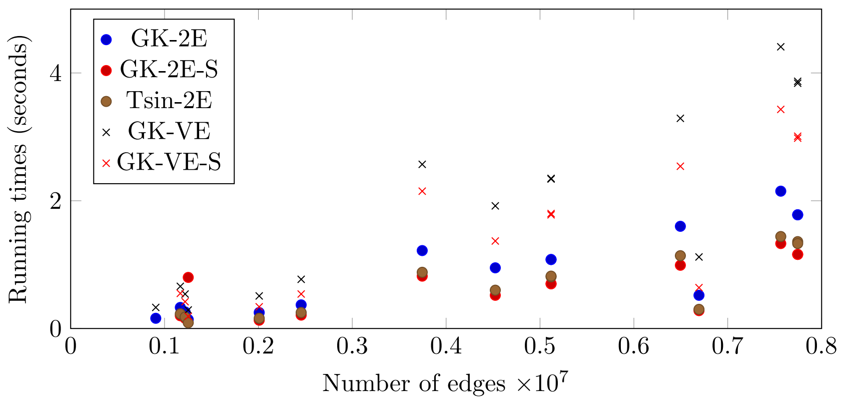
**Table 3** Running times in seconds and memory usage of the OGDF-based implementations for the graphs of Table 1 in experimental setting (II). The best results in each row are marked in bold.

| **Graphs** | **Running Times** | | | **RAM memory consuption** | | |
|---|---|---|---|---|---|---|
| | GK-VE-SF | SPQR-VE | Split-VE | GK-VE-SF | SPQR-VE | Split-VE |
| Amazon0302 | **1.04** | 3.18 | 2.63 | **294** MB | 756 MB | 589 MB |
| com-amazon | **1.76** | 4.94 | 4.05 | **586** MB | 1.1 GB | 946 MB |
| com-dblp | **1.53** | 4.92 | 3.93 | **494** MB | 1.2 GB | 887 MB |
| web-NotreDame | **0.82** | 4.08 | 3.32 | **438** MB | 1.1 GB | 855 MB |
| web-Stanford | **1.81** | 7.76 | 6.45 | **581** MB | 1.6 GB | 1.2 GB |
| Amazon0601 | **2.66** | 9.21 | 7.71 | **693** MB | 2 GB | 1.5 GB |
| ia-yahoo-messages | **6.68** | 19.62 | 17.90 | **2.3** GB | 3.8 GB | 3.7 GB |
| web-Google | **5.74** | 19.69 | 16.48 | **1.3** GB | 3.9 GB | 3 GB |
| ca-cit-HepTh | **1.09** | 7.81 | 6.53 | **2.5** GB | 4.7 GB | 4.4 GB |
| cit-hepTh | **1.11** | 8.00 | 6.50 | **2.5** GB | 4.7 GB | 4.4 GB |
| visualise-us | **9.94** | 34.12 | 28.88 | **2.9** GB | 5.8 GB | 5.4 GB |
| web-BerkStan | **4.37** | 27.86 | 20.87 | **1.7** GB | 5.2 GB | 4 GB |
| ca-IMDB | **7.74** | 22.31 | 18.32 | **3.2** GB | 6.8 GB | 6.1 GB |
| ca-cit-HepPh | **1.45** | 10.54 | 8.69 | **4.4** GB | 7.3 GB | 7.1 GB |
| cit-hepPh | **1.42** | 10.50 | 8.68 | **4.4** GB | 7.3 GB | 7.1 GB |

in every cyclic and complete graph, $n_C$ and $n_K$ respectively. Then it processes these graphs in random order, and it connects every one of them to a graph that was already processed, by inserting two edges that stem from two different vertices and end in two different vertices (thus ensuring biconnectivity). By carefully selecting the number and the sizes of the cyclic and complete graphs, we can determine the density of the resulting graph (since the number of edges of such a graph is given by $Cn_c + Kn_K(n_K - 1)/2 + 2(C + K - 1)$). In particular, we note that the graphs produced by our generator may contain a lot of cut-pairs (depending on $C$, $n_C$ and $K$), even if they are very dense.

**Table 4** Running times in seconds for artificial graphs in experimental setting (I). Parameters $C$ and $K$ correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 200$ vertices. The number of vertices $n$ and edges $m$ refer to the produced instances. The best results in each row are marked in bold.

| Graph details | | | | 2-edge cuts | | | vertex-edge cuts | |
|---|---|---|---|---|---|---|---|---|
| $C$ | $K$ | $n$ | $m$ | GK-2E | GK-2E-S | Tsin-2E | GK-VE | GK-VE-S |
| 5000 | 0 | 1000000 | 1009998 | 0.68 | **0.44** | 0.45 | 1.38 | **1.12** |
| 4500 | 500 | 1000000 | 10859998 | 1.30 | **0.74** | 0.82 | 2.66 | **1.76** |
| 4000 | 1000 | 1000000 | 20709998 | 1.92 | **1.02** | 1.16 | 3.99 | **2.39** |
| 3500 | 1500 | 1000000 | 30559998 | 2.52 | **1.30** | 1.51 | 5.23 | **2.97** |
| 3000 | 2000 | 1000000 | 40409998 | 3.11 | **1.60** | 1.85 | 6.50 | **3.57** |
| 2500 | 2500 | 1000000 | 50259998 | 3.71 | **1.86** | 2.20 | 7.76 | **4.18** |
| 2000 | 3000 | 1000000 | 60109998 | 4.32 | **2.13** | 2.54 | 8.99 | **4.76** |
| 1500 | 3500 | 1000000 | 69959998 | 4.88 | **2.41** | 2.88 | 10.27 | **5.36** |
| 1000 | 4000 | 1000000 | 79809998 | 5.48 | **2.68** | 3.21 | 11.45 | **6.05** |
| 500 | 4500 | 1000000 | 89659998 | 6.05 | **2.94** | 3.53 | 12.74 | **6.51** |
| 0 | 5000 | 1000000 | 99509998 | 6.63 | **3.19** | 3.88 | 13.99 | **7.09** |



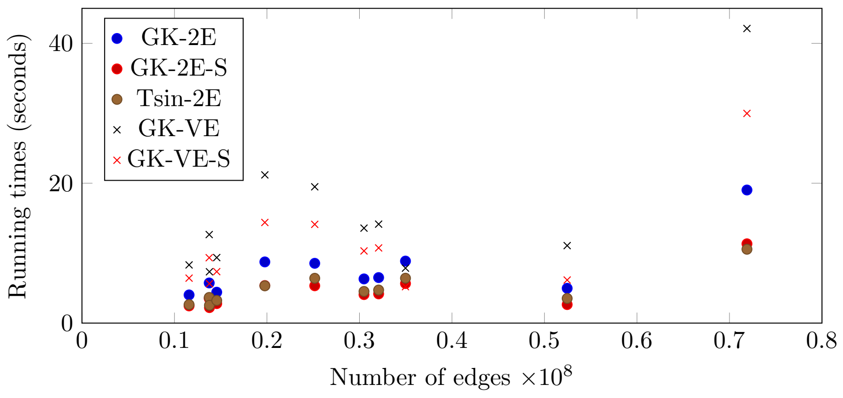


**Figure 1** Running times for the graphs of Table 1 in experimental setting (I). The top plot shows the running times for graphs with less than 10M edges, while the bottom plot shows the running times for graphs with more than 10M edges.

**Table 5** Running times in seconds for artificial graphs in experimental setting (I). Parameters $C$ and $K$ correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 1000$ vertices. The number of vertices $n$ and edges $m$ refer to the produced instances. The best results in each row are marked in bold.

| Graph details | | | | 2-edge cuts | | | vertex-edge cuts | |
|---|---|---|---|---|---|---|---|---|
| $C$ | $K$ | $n$ | $m$ | GK-2E | GK-2E-S | Tsin-2E | GK-VE | GK-VE-S |
| 1000 | 0 | 1000000 | 1001998 | 0.68 | **0.44** | 0.47 | 1.31 | **1.10** |
| 900 | 100 | 1000000 | 50851998 | 4.12 | **2.04** | 2.32 | 8.41 | **4.80** |
| 800 | 200 | 1000000 | 100701998 | 7.52 | **3.63** | 4.33 | 15.39 | **8.38** |
| 700 | 300 | 1000000 | 150551998 | 10.92 | **5.18** | 6.25 | 22.84 | **12.04** |
| 600 | 400 | 1000000 | 200401998 | 14.35 | **6.74** | 8.09 | 30.84 | **15.53** |
| 500 | 500 | 1000000 | 250251998 | 17.61 | **8.29** | 10.04 | 38.71 | **19.04** |
| 400 | 600 | 1000000 | 300101998 | 21.10 | **9.84** | 11.88 | 44.27 | **22.70** |
| 300 | 700 | 1000000 | 349951998 | 24.49 | **11.40** | 13.96 | 54.70 | **26.02** |
| 200 | 800 | 1000000 | 399801998 | 27.77 | **12.94** | 15.67 | 63.00 | **29.89** |
| 100 | 900 | 1000000 | 449651998 | 31.14 | **14.53** | 17.67 | 72.01 | **33.72** |
| 0 | 1000 | 1000000 | 499501998 | 34.47 | **16.15** | 19.37 | 80.53 | **37.00** |

**Table 6** Running times in seconds for artificial graphs in experimental setting (I). Parameters $C$ and $K$ correspond to the number of cyclic and complete graphs, respectively, where each such graph has $n_C = n_K = 2000$ vertices. The number of vertices $n$ and edges $m$ refer to the produced instances. The best results in each row are marked in bold.

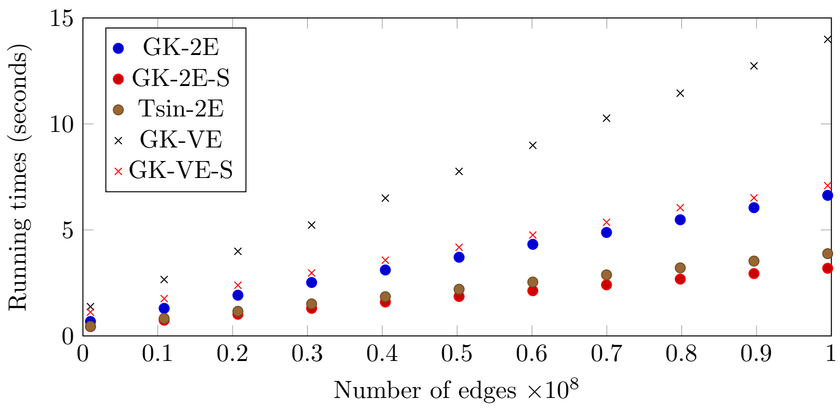
| Graph details | | | | 2-edge cuts | | | vertex-edge cuts | |
|---|---|---|---|---|---|---|---|---|
| $C$ | $K$ | $n$ | $m$ | GK-2E | GK-2E-S | Tsin-2E | GK-VE | GK-VE-S |
| 500 | 0 | 1000000 | 1000998 | 0.68 | **0.45** | 0.48 | 2.03 | **1.77** |
| 450 | 50 | 1000000 | 100850998 | 8.34 | **4.03** | 4.89 | 19.81 | **10.49** |
| 400 | 100 | 1000000 | 200700998 | 16.44 | **8.08** | 9.08 | 37.39 | **18.07** |
| 350 | 150 | 1000000 | 300550998 | 24.04 | **11.62** | 14.01 | 51.45 | **25.74** |
| 300 | 200 | 1000000 | 400400998 | 31.60 | **15.25** | 18.18 | 70.57 | **33.94** |
| 250 | 250 | 1000000 | 500250998 | 40.41 | **18.65** | 21.90 | 78.38 | **42.14** |
| 200 | 300 | 1000000 | 600100998 | 46.78 | **22.43** | 26.39 | 93.28 | **50.10** |
| 150 | 350 | 1000000 | 699950998 | 59.55 | **25.83** | 30.79 | 120.87 | **58.52** |
| 100 | 400 | 1000000 | 799800998 | 61.69 | **28.93** | 33.78 | 124.60 | **67.29** |
| 50 | 450 | 1000000 | 899650998 | 69.94 | **32.49** | 38.68 | 139.34 | **75.50** |
| 0 | 500 | 1000000 | 999500998 | 79.28 | **38.21** | 42.14 | 155.44 | **83.13** |

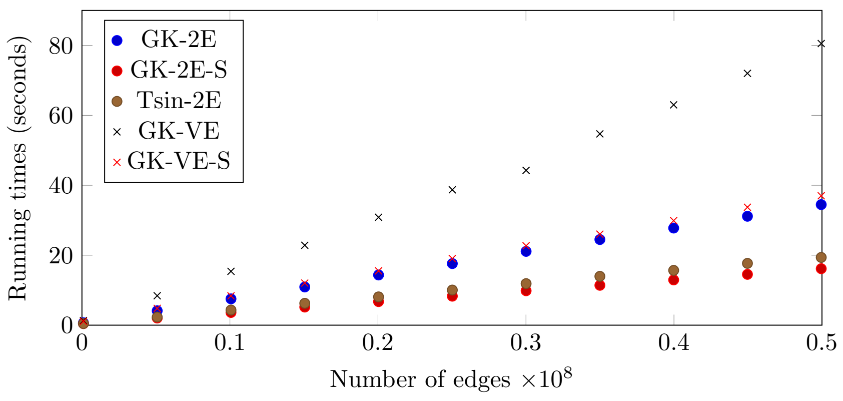
The corresponding experimental results are given in Tables 4, 5 and 6, and plotted in Figure 2. We choose the values of parameters $C$, $K$, $n_C$ and $n_K$ so that all produced instances have $n = 1000000$ vertices, and vary their density and structure. Also, for simplicity, we choose $n_C = n_K$ in all instances. Then, it is easy to observe that as we decrease the value of $C$ (and correspondingly increase $K$ so that we maintain the total number of vertices fixed), the number of vertex-edge cut-pairs as well as the number of 2-edge-cuts decrease, while the graph gets more dense. From the experimental results, however, we observe that the number of cuts does not affect the performance of the algorithms. Indeed, similarly to our previous experiments, GK-2E-S is consistently faster than GK-2E by more than 50% on average. Also, with respect to Tsin-2E, GK-2E-S is faster on all instances by 15% on average.

Regarding the performance of the algorithms for computing vertex-edge cut-pairs, we note that in this experiment, our streamlined version GK-VE-S achieves higher speed-ups with respect to GK-VE. Specifically, GK-VE-S is faster than GK-VE by 16% up to 54% (more than 45% on average).

## 6 Concluding remarks

We presented streamlined versions of two linear-time algorithms of [6] that compute the vertex-edge cut-pairs of a biconnected graph $G$ and the 2-edge cuts of a 2-edge-connected graph, respectively. Although we can compute these cuts in linear time using previously known techniques, the new algorithms have a major advantage: Both algorithms are based on a common framework, which results in conceptually simple and easy to implement algorithms, especially for computing vertex-edge cut-pairs, where the alternative linear-time algorithms exploit the structure of the triconnected components of $G$. Furthermore, our experimental results showed that our new algorithms perform significantly better in practice both in terms of running time and of space requirements.





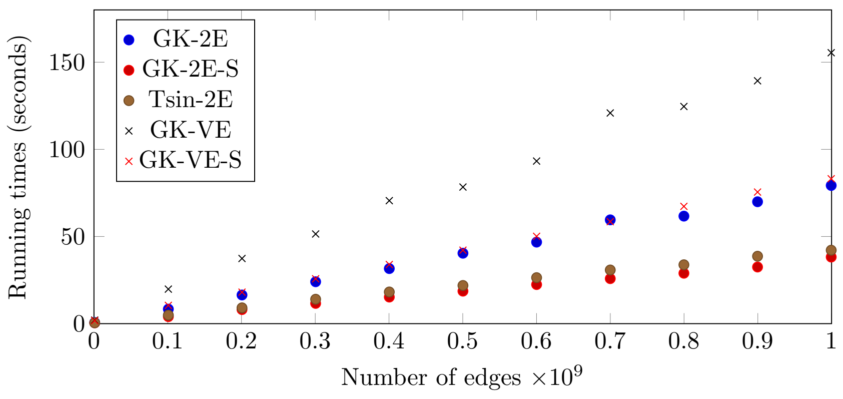


**Figure 2** Running times for artificial graphs in experimental setting (I). The plots, from top to bottom, show the running times for the graphs of Tables 4, 5, and 6, respectively.

## References

1 G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15(4):302–318, April 1996.

2 G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25(5):956–997, 1996.

3 M. Chimani, C. Gutwenger, M. Junger, G. W. Klau, K. Klein, and P. Mutzel. The open graph drawing framework. In *Handbook of Graph Drawing and Visualization*, pages 543–570. CRC Press, 2013.

4 D. Fussell, V. Ramachandran, and R. Thurimella. Finding triconnected components by local replacement. *SIAM J. Comput.*, 22(3):587–616, June 1993. doi:10.1137/0222040.

5 Z. Galil and G. F. Italiano. Reducing edge connectivity to vertex connectivity. *SIGACT News*, 22(1):57–61, 1991. doi:10.1145/122413.122416.

6 L. Georgiadis and E. Kosinas. Linear-Time Algorithms for Computing Twinless Strong Articulation Points and Related Problems. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ISAAC.2020.38.

7 C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In Joe Marks, editor, *Graph Drawing*, pages 77–90, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

8 J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.

9 R. Jaberi. Twinless articulation points and some related problems, 2019. arXiv:1912.11799.

10 Z. Jiang. An empirical study of 3-vertex connectivity algorithms. Master's thesis, University of Windsor, 2013. Electronic Theses and Dissertations, paper 4980.

11 J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

12 K. Mehlhorn, A. Neumann, and J. M. Schmidt. Certifying 3-edge-connectivity. *Algorithmica*, 77(2):309–335, February 2017. doi:10.1007/s00453-015-0075-x.

13 H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math*, 9(163), 1992. doi:10.1007/BF03167564.

14 S. Raghavan. Twinless strongly connected components. In F. B. Alt, M. C. Fu, and B. L. Golden, editors, *Perspectives in Operations Research: Papers in Honor of Saul Gass' 80th Birthday*, pages 285–304. Springer US, Boston, MA, 2006. doi:10.1007/978-0-387-39934-8_17.

15 R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL: http://networkrepository.com.

16 R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

17 Y. H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7(1):130–146, 2009. Selected papers from the 1st International Workshop on Similarity Search and Applications (SISAP). doi:10.1016/j.jda.2008.04.003.

18 Wikipedia contributors. SPQR tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SPQR_tree&oldid=951256273, 2020.

## A Omitted algorithms

**Algorithm 1** compute all $b_p\_count(v)$ and $up(v)$ while performing a DFS.

```
initialize all $dfs$ labels to $\emptyset$
$dfs \leftarrow 1$
initialize an array $p(v)$ with size $n$
initialize an array $tempChild(v)$ with size $n$
initialize all $b_p\_count(v)$ to 0
initialize all $up(u)$ to 0        /* this is also used in the modified algorithms
  "$high(u) = v$" and "$high(u) < v$", in order to test if $high(u) = high_p(u)$ */
DFS($r$)
Function $DFS$(vertex $v$)
begin
    $dfs(v) \leftarrow dfs$
    $dfs \leftarrow dfs + 1$
    foreach $vertex\ u\ adjacent\ to\ v$ do
        if $dfs(u) = \emptyset$ then
            $p(u) \leftarrow v$
            $tempChild(v) \leftarrow u$
            DFS($u$)
        end
        if $dfs(u) < dfs(v)$ and $u \neq p(v)$ then
            $b_p\_count(v) \leftarrow b_p\_count(v) + 1$
            $up(tempChild(u)) \leftarrow up(tempChild(u)) + 1$
        end
        else if $v = p(u)$ then
            $b_p\_count(v) \leftarrow b_p\_count(v) + b_p\_count(u)$
        end
    end
    $b_p\_count(v) \leftarrow b_p\_count(v) - up(v)$
end
```

**Algorithm 2** Compute the cut-edges and the number of 2-cuts.

```
perform a DFS, and compute all $low(v)$, $b\_count(v)$, $M(v)$ and $nextM(v)$, for all vertices
  $v \neq r$
$n2cuts \leftarrow 0$
// case back-edge - tree-edge
foreach $v \neq r$ do
    if $b\_count(v) = 1$ then
        mark the edges $(v, p(v))$ and $(M(v), low(v))$
        $n2cuts \leftarrow n2cuts + 1$
    end
end
// case tree-edge - tree-edge
foreach $v \neq r$ that has $M(v) = v$ do
    while $v \neq \emptyset$ do
        $u \leftarrow nextM(v)$
        $nCutEdges \leftarrow 0$
        while $u \neq \emptyset$ and $b\_count(u) = b\_count(v)$ do
            mark the edges $(v, p(v))$ and $(u, p(u))$
            $nCutEdges \leftarrow nCutEdges + 1$
            $u \leftarrow nextM(u)$
        end
        $n2cuts \leftarrow n2cuts + nCutEdges(nCutEdges + 1)/2$
        $v \leftarrow u$
    end
end
```

**Algorithm 3** Compute all $M(v)$ and $nextM(v)$.

```
foreach $v \neq r$ do
    $L(v) \leftarrow$ first child of $v$
    $R(v) \leftarrow$ last child of $v$
    $nextM(v) \leftarrow \emptyset$
end
foreach $v \neq r$, in a bottom-up fashion do
    $c \leftarrow v$
    $m \leftarrow v$
    while true do
        if $l(m) < v$ then $M(v) \leftarrow m$ break
        while $low(L(m)) \geq v$ do $L(m) \leftarrow$ next child of $m$
        while $low(R(m)) \geq v$ do $R(m) \leftarrow$ previous child of $m$
        if $L(m) \neq R(m)$ then $M(v) \leftarrow m$ break
        $c \leftarrow L(m)$
        $m \leftarrow M(c)$
    end
    if $c \neq v$ then
        $nextM(c) = v$
    end
end
```

## B Omitted experimental results

**Table 7** Memory consumption for the graphs of Table 1 in experimental setting (I). The best results in each row are marked in bold.

| | 2-edge cuts | | | vertex-edge cuts | |
|---|---|---|---|---|---|
| **Graph** | GK-2E | GK-2E-S | Tsin-2E | GK-VE | GK-VE-S |
| Amazon0302 | 46 MB | **42** MB | 58 MB | 61 MB | **51** MB |
| com-amazon | 81 MB | **72** MB | 90 MB | 110 MB | **90** MB |
| com-dblp | 69 MB | **62** MB | 83 MB | 94 MB | **77** MB |
| web-NotreDame | 61 MB | **55** MB | 77 MB | 65 MB | **53** MB |
| web-Stanford | 74 MB | **69** MB | 106 MB | 111 MB | **79** MB |
| Amazon0601 | 95 MB | **89** MB | 133 MB | 140 MB | **103** MB |
| ia-yahoo-messages | 394 MB | **346** MB | 390 MB | 518 MB | **449** MB |
| web-Google | 192 MB | **178** MB | 258 MB | 282 MB | **209** MB |
| ca-cit-HepTh | 378 MB | **338** MB | 412 MB | 513 MB | **424** MB |
| cit-HepTh | 373 MB | **338** MB | 412 MB | 513 MB | **424** MB |
| visualise-us | 460 MB | **410** MB | 513 MB | 634 MB | **522** MB |
| web-BerkStan | **209** MB | **209** MB | 332 MB | 314 MB | **233** MB |
| ca-IMDB | 542 MB | **485** MB | 597 MB | 738 MB | **608** MB |
| ca-cit-HepPh | 626 MB | **556** MB | 665 MB | 841 MB | **705** MB |
| cit-HepPh | 618 MB | **556** MB | 665 MB | 841 MB | **705** MB |
| amazon-ratings | 835 MB | **746** MB | 917 MB | 1.1 GB | **936** MB |
| hugetrace-00000 | 986 MB | **868** MB | 1.1 GB | 1.3 GB | **1.1** GB |
| rgg-n-2-20-s0 | 988 MB | **883** MB | 1.1 GB | 1.3 GB | **1.1** GB |
| wiki-user-edits-page | 1.2 GB | **1** GB | 1.2 GB | 1.6 GB | **1.3** GB |
| hugetric-00010 | 1.4 GB | **1.2** GB | 1.5 GB | 1.9 GB | **1.6** GB |
| delaunay-n22 | 1.8 GB | **1.6** GB | 1.9 GB | 2.4 GB | **2** GB |
| co-papers-dblp | 2.1 GB | **1.9** GB | 2.3 GB | 2.9 GB | **2.4** GB |
| co-papers-citeseer | 2.2 GB | **2** GB | 2.5 GB | 3.1 GB | **2.5** GB |
| packing-b050 | 2.4 GB | **2.2** GB | 2.7 GB | 3.3 GB | **2.7** GB |
| human-Jung2015 | **1.3** GB | **1.3** GB | 2.3 GB | 2.2 GB | **1.4** GB |
| rgg-n-2-23-s0 | **2.3** GB | **2.4** GB | 3.6 GB | 3.7 GB | **2.5** GB |

# How to Find the Exit from a 3-Dimensional Maze[*]

**Miki Hermann**
LIX, CNRS, École Polytechnique, Institut Polytechnique de Paris, 91120 Palaiseau, France

**Abstract**

We present several experimental algorithms for fast computation of variadic polynomials over non-negative integers.

**2012 ACM Subject Classification** Theory of computation → Theory and algorithms for application domains

**Keywords and phrases** Young tableaux, randomized algorithm, probabilistic algorithm

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.21

**Supplementary Material** *Software (Source Code & Data)*: `https://github.com/miki-hermann/gyt` archived at `swh:1:dir:aa547a4af49a9563138a13168af0160d9e709954`

## 1 Introduction and Motivation

Imagine a three-dimensional cubic maze structure called the *Cube*. Each side of the *Cube* spans 26 rooms and there are $26 \times 26 \times 26 = 17576$ rooms in total. Except for the rooms on the edges or faces of the *Cube*, each room has 6 neighbors: up, down, left, right, front, and back. Each room is identified by its coordinates $x$, $y$, and $z$, ranging from 0 to 25. Moreover, each room has a label written on its floor, determined by an unknown ternary function $f\colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ over natural numbers. The parameters of the function $f$ are the coordinates of the room. The only information you have about the function $f$ is that it is increasing in each coordinate, i.e., that the following relations hold

$$f(x,y,z) < f(x+1,y,z), \quad f(x,y,z) < f(x,y+1,z), \quad f(x,y,z) < f(x,y,z+1)$$

for each coordinate $(x,y,z)$. You do not know the labels of the rooms upfront, but discover them by visiting the rooms on your path to the exit. Labels are not unique: two different rooms can have the same label. You can pass from one room to another if there exists a door between them. Each pair of neighboring rooms share a door, which means that there is a door between any two rooms sharing a face. Except for the rooms on the edges and outer faces of the *Cube*, from a given room you can pass to a neighbor room up, down, left, right, front, or back. Formally speaking, from a room with coordinates $(x,y,z)$ you can pass to one of the rooms with coordinates

$$(x+1,y,z), \quad (x,y+1,z), \quad (x,y,z+1), \quad (x-1,y,z), \quad (x,y-1,z), \quad (x,y,z-1),$$

when $0 < x,y,z < 25$. Contrary to the movie, there are no deadly traps in the rooms. Nevertheless, you are not allowed to pass between rooms freely. You cannot return back to a previously visited room unless you have flagged it. If you are not sure which choice to make, you can flag the current room, so that you can return to it later. If you decide in a certain moment that you arrived at a dead-end, you can ask to be teleported back to the last flagged room. You can return to each flagged room only once, i.e., you have two choices to move from a flagged room to another room, allowing you a limited backtrack, contrary to unflagged rooms where you have only one choice. You have 29 flags available, i.e., you have

[*] Inspired by the Horror Movie *Cube*.

the possibility to return to 29 branchings. Once a room is flagged, you cannot remove it any more. The coordinates of the flagged rooms are maintained in a stack. You can return only to the room whose flag is on top of the stack. Once you return to a flagged room, its coordinates are popped from the stack. The exit room is labeled by 131350013988347832235. Your starting position is the room with coordinates $(0, 0, 25)$ labeled by 162981450557708740234375. Are you able to find the exit? What is the minimal number of rooms you must pass through from your starting position to the exit?[1]

## 2 Analysis

Before passing to the three- and more-dimensional case, let us analyze the problem in lower dimensions.

### 2.1 Linear Board

In one dimension, the analysis is quite easy. We have a linear board of length $n$ with coordinates $0, \dots, n-1$, an unknown unary function $f \colon \mathbb{N} \to \mathbb{N}$, a starting position $s$, and an exit label $B$. For two different positions $a, b \in \{0, \dots, n-1\}$ on board we know that $a < b$ implies $f(a) < f(b)$. Hence the exit label $B$ can occur only once on a linear board. The starting point $s$ is one of the extremities of the board: either $s = 0$ or $s = n - 1$.

The search algorithm proceeds as follows. First, we set $x \leftarrow s$ and compute the value $f(x)$. If $f(x) = B$ holds, then we are already at the exit room. If $f(x) > B$ we must decrease $x$, else if $f(x) < B$ we must increase $x$. Set $x \leftarrow x - 1$ or $x \leftarrow x + 1$, respectively, and repeat the loop. No flags are necessary to reach the exit, since there is no necessity to make choices.

In the worst case, the starting point is at one extremity of the board (say 0), and the exit at the other $(n - 1)$. Hence the path to the exit must contain $n$ rooms in the worst case. On average, each position from $0, \dots, n-1$ is equally likely to contain the exit. The probability $p_i$ that a position $i$ contains the exit is $p = 1/n$. We denote by $X$ the random variable equal to the length of the path from 0 to the exit and set $\Pr[X = i] = p_{i-1} = 1/n$. If the starting point is one of the extremities ($s = 0$ or $s = n - 1$, but these two cases are mirror images of each other), the expected length of the path to the exit is

$$E[X|s=0] \;=\; E[X|s=n-1] \;=\; \sum_{i=1}^{n} i \cdot \Pr[X=i] \;=\; \sum_{i=1}^{n} \left(i \cdot \frac{1}{n}\right) \;=\; (n+1)/2.$$

Not really a surprise, this position is near the middle of the linear board.

### 2.2 Matrix Board

A $m \times n$ matrix $A$, whose elements $A[x, y]$ are equal to a binary function $f(x, y) \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, satisfying the relations$f(x, y) < f(x + 1, y)$ and $f(x, y) < f(x, y + 1)$, is a full Young tableau over natural numbers [2, Problem 6-3, page 143]. The starting position $s$ is usually $(0, 0)$, but for simplification reasons we will consider the coordinate $(0, n - 1)$ as the starting point. Just consider the matrix horizontally flipped. For two different positions $a = (a_1, a_2)$ and $b = (b_1, b_2)$ there exist two positions $c = (c_1, c_2)$ and $d = (d_1, d_2)$, such that $c_i = \min\{a_i, b_i\}$

[1] Just for your information, the exit is located in the room with coordinates $(14, 15, 16)$ and the minimal number of visited rooms is therefore 39, including the starting room and the exit.

**Algorithm 1** Search in a 2D Maze.

**Input:** Function $f\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ satisfying $f(x, y) < f(x + 1, y)$ and $f(x, y) < f(x, y + 1)$, and a value $B \in \mathbb{N}$.
**Output:** Coordinates $(x, y)$ for which $f(x, y) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function 2D_SEARCH($f, B$)
2:      $x \leftarrow 0$
3:      $y \leftarrow n - 1$
4:      while $x \leq m - 1$ & $y \geq 0$ do
5:          if $f(x, y) = B$ then
6:              return $(x, y)$
7:          else if $f(x, y) < B$ then
8:              $x \leftarrow x + 1$
9:          else if $f(x, y) > B$ then
10:             $y \leftarrow y - 1$
11:         end if
12:     end while
13:     return $\perp$
14: end function
```

and $d_i = \max\{a_i, b_i\}$, satisfying the relations

$$f(c_1, c_2) \leq f(a_1, a_2), f(c_1, c_2) \leq f(b_1, b_2), f(a_1, a_2) \leq f(d_1, d_2), \text{ and } f(b_1, b_2) \leq f(d_1, d_2).$$

All four relations are strict if the positions $a$ and $b$ do not share the same row or column. Hence, the exit label $B$ can occur only once in each row and only once in each column.

Algorithm 1 proceeds as follows. The starting position is the room $s$ with coordinates $(0, n - 1)$, therefore we set $x \leftarrow 0$ and $y \leftarrow n - 1$. While $f(x, y) > B$ holds, decrease the second coordinate: $y \leftarrow y - 1$. When we arrive at a position where $f(x, y) < B$, we increase the first coordinate: $x \leftarrow x + 1$. We repeat this loop until we find a room labeled by $B$. No flags are necessary to reach the exit, since there is no necessity to make choices.

The correctness of Algorithm 1 is easily proved. If $f(x, y) > B$ holds, then we have $f(x', y) > B$ for each $x' > x$. Hence the exit room labeled by $B$ cannot be located at any position $(x', y')$ for $x' \geq x$ and $y' \geq y$. Therefore there is no need to increase the first coordinate if the $f(x, y) > B$ holds. Only the second coordinate can be decreased to move towards the exit room. If $f(x, y) < B$ holds, then we have $f(x, y') < B$ for each position $y' < y$. Hence the exit room labeled by $B$ cannot be located at any position $(x', y')$ for $x' \leq x$ and $y' \leq y$. Therefore there is no need to decrease the second coordinate. Only the first coordinate can be increased to move towards the exit room.

In the worst case, the starting point is at the south-east extremity $(0, n - 1)$ of the maze and the exit at the north-western extremity $(m - 1, 0)$. Algorithm 1 proceeds by a zig-zag, which never returns back. Neither the coordinate $y$ (columns) is increased, nor the coordinate $x$ (rows) is decreased. There are $m$ rooms in each row and $n$ rooms in each column. Therefore the path to the exit must contain $m + n$ rooms in the worst case. This is a considerable improvement against a brute force algorithm going through all $m \cdot n$ rooms.

## 3 The *Cube* and Beyond

Let us extend the previous ideas to a cubic maze. We have $\ell \times m \times n$ cube $A$, whose elements $A[x, y, z]$ are equal to a ternary function $f(x, y, z)\colon \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, satisfying the inequalities

$f(x,y,z) < f(x+1,y,z)$, $f(x,y,z) < f(x,y+1,z)$, and $f(x,y,z) < f(x,y,z+1)$. The starting position is, once again, $(0,0,n-1)$. If you wish to start at the origin $(0,0,0)$, you can just flip the cube and arrange the subsequent computation according to this flip.

The basic idea of the algorithm remains the same. We set $x \leftarrow 0$, $y \leftarrow 0$, and $z \leftarrow n-1$ at the beginning. While $f(x,y,z) > B$ holds, we decrease the last coordinate: $z \leftarrow z-1$. This is correct, because we have $f(x',y',z) > B$ for any $x' \geq x$ and $y' \geq y$ when $f(x,y,z) > B$ holds. This implies that the value $B$ cannot be in the cube slice with the fixed $z$. When we reach a coordinate $z$ with $f(x,y,z) < B$, then we have $f(x',y',z') < B$ for any position with $x' \leq x$, $y' \leq y$, and $z' \leq z$. However, we now have the choice to increase either $x$ or $y$, contrary to the two-dimensional case, where we were forced to increase only one variable. Here, we have the choice to continue either to the room $(x+1,y,z)$ or $(x,y+1,z)$. This is the point where the flags must be applied, i.e., where we must apply some limited backtrack. The other strategy is to proceed in a chosen direction without a possibility to return to the choice position.

Let us analyze four possible strategies. We will do it for the general case with $k$ coordinates, where $k \geq 3$. We will place the continuations onto a stack or into a queue, then proceed further. Those strategies, with the possibility to return to a remembered room, face another problem. In the two-dimensional case, the path from the starting point to the exit was exactly determined and there were no two or more paths possible to any room in the maze. However, with the possibility to return to a choice room, we have the possibility to reach another choice room by two or more different paths. Therefore we must memorize the choice rooms in which we have been before. This does not count for teleportation returns, but simple arrivals by a path from another room. If we arrive in a choice room $r$ the second time by a different path, we can stop the search and ask to be teleported back to a previous choice room $r'$, since all possible path from the room $r$ must have been already explored, i.e., all choices for a continuation from the room $r$ have been already placed on the stack or into the queue. When we are in dimension $k$ for $k \geq 3$, there are $k-1$ possible continuations from a choice room. Therefore we must allow to return by teleportation to a choice room $(k-2)$-times. This is compatible with the situation in Section 1, where we allow only one teleportation return to a choice room.

A $k$-dimensional maze has the shape of a $n_1 \times \cdots \times n_k$ hypercube $A$, whose elements $A[x_1,\ldots,x_k]$ are equal to the values of a function $f\colon \mathbb{N}^k \to \mathbb{N}$„ satisfying the inequality $f(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_k) < f(x_1,\ldots,x_{i-1},x_i+1,x_{i+1},\ldots,x_k)$ for each $i = 1,\ldots,k$. The starting point will be $(0,\ldots,0,n_k-1)$.

The first strategy will be completely *sequential*, presented in Algorithm 2. In each choice room $r$ with coordinates $(x_1,\ldots,x_k)$ it will put on stack all possible continuations $(x_1,\ldots,x_{i-1},\ldots,x_i+1,x_{i+1}\ldots,x_k)$ from $i=1$ to $i=k-1$. This implies, that the continuation $(x_1,\ldots,x_{k-2},x_{k-1}+1,x_k)$ will be popped first. However, only the continuation rooms will be put on stack, which have not been visited yet. Of course, if $k=2$ then only one continuation room will be put on stack, but it will be popped and considered immediately during the next turn of the outer while-loop on Line 6. Hence, Algorithm 2 is compatible with Algorithm 1. Moreover, the use of visited rooms is superfluous for $k=2$, but introducing another if-statement would just unnecessarily complicate the algorithm. A successful path to the exit contains in the worst case at most $N = \sum_{i=1}^{k} n_i$ rooms, without counting the possible backtracks.

This version of our algorithm does not use the concept of flags, or allows to use an unbounded number of flags. The version using flags would require Algorithm 2 to be called with the parameters $(f,B,F)$ on Line 1, where $F$ is the number of allowed flags, followed by

**Algorithm 2** Sequential Search in a $k$-Dimensional Maze.

```
Input: Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.
Output: Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.
 1: function SEQ_kD_SEARCH($f, B$)
 2:     $s \leftarrow \emptyset$                                        ▷ Initialize stack $s$
 3:     $m \leftarrow \emptyset$                         ▷ Initialize memory $m$ of visited rooms
 4:     $r \leftarrow (0, \dots, 0, n_k - 1)$                            ▷ Initialize room $r$
 5:     $s$.push($r$)                                        ▷ Put room $r$ on stack $s$
 6:     while $s \neq \emptyset$ do                              ▷ While stack $s$ is nonempty
 7:         $r \leftarrow$ top($s$)                           ▷ Get room from top of stack $s$
 8:         pop($s$)                                                    ▷ Pop stack $s$
 9:         while $r[j] < n_j$ for $j = 1, \dots, k - 1$ & $r[k] \geq 0$ do
10:             if $f(r) = B$ then
11:                 return $r$
12:             else if $f(r) > B$ then
13:                 $r[k] \leftarrow r[k] - 1$
14:             else if $f(r) < B$ then
15:                 for $i \leftarrow 1$ to $k - 1$ do    ▷ For each potential continuation coordinate
16:                     $r' \leftarrow r$                            ▷ Copy room coordinates
17:                     $r'[i] \leftarrow r'[i] + 1$                  ▷ Go to a neighboring room
18:                     if $r'[i] < n_i$ & $r' \notin m$ then  ▷ If room within limits and not visited
19:                         $s$.push($r'$)                  ▷ Put continuation room $r'$ on stack $s$
20:                         $m$.insert($r'$)                        ▷ Label room $r'$ as visited
21:                     end if
22:                 end for
23:             end if
24:         end while
25:     end while
26:     return $\perp$
27: end function
```

an extension of the condition on Line 18 to "$r'[i] < n_i$ & $r' \notin m$ & $F > 0$", an introduction of the statement "$F \leftarrow F - 1$" between Lines 18 and 19, plus inserting a test "**else if** $F = 0$" with a subsequent failure command after Line 20.

The second strategy is based on a greedy heuristic to always choose first the continuation room nearest to the exit. For this, a priority queue is maintained to include the coordinates of the continuation room together with their "distance" to the exit. Since we do not know the coordinates of an exit room – which we are supposed to find – we use the difference between the label $f(x_1, \dots, x_k)$ of a continuation room $r$ and the label $B$ of an exit room as the priority key. This priority strategy is presented in Algorithm 3. We assume that the priority queue is implemented by a heap, therefore no code concerning an implementation of the priority queue is presented. An interested reader can find more information on priority queues and their implementation by a heap for instance in [6].

This version of our algorithm is similar to Dijkstra's shortest path algorithm [6, Section 4.4] in a graph. Dijkstra's algorithm applied directly on regular multidimensional Cartesian

**Algorithm 3** Priority Search in a $k$-Dimensional Maze.

**Input:** Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.

**Output:** Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function PRIORITY_kD_SEARCH($f, B$)
2:      $q \leftarrow \emptyset$                                  ▷ Initialize priority queue $q$
3:      $m \leftarrow \emptyset$                                  ▷ Initialize memory $m$ of visited rooms
4:      $r \leftarrow (0, \dots, 0, n_k - 1)$                     ▷ Initialize room $r$
5:      $s$.insert($(r, 0)$)                                      ▷ Insert room $r$ into queue $q$ with dummy key 0
6:      while $q \neq \emptyset$ do                               ▷ While queue $q$ is nonempty
7:          $r \leftarrow$ front($q$).first                       ▷ Get room coordinates from front of queue $q$
8:          pop($q$)                                              ▷ Pop queue $q$
9:          while $r[j] < n_j$ for $j = 1, \dots, k-1$ & $r[k] \geq 0$ do
10:             if $f(r) = B$ then
11:                 return $r$
12:             else if $f(r) > B$ then
13:                 $r[k] \leftarrow r[k] - 1$
14:             else if $f(r) < B$ then
15:                 for $i \leftarrow 1$ to $k - 1$ do            ▷ For each potential continuation coordinate
16:                     $r' \leftarrow r$                         ▷ Copy room coordinates
17:                     $r'[i] \leftarrow r'[i] + 1$              ▷ Go to a neighboring room
18:                     if $r'[i] < n_i$ & $r' \notin m$ then     ▷ If room within limits and not visited
19:                         $c \leftarrow |f(r') - B|$            ▷ Compute key $c$
20:                         $q$.insert($(r', c)$)                 ▷ Insert room $r'$ with key $c$ into queue $q$
21:                         $m$.insert($r'$)                      ▷ Label room $r'$ as visited
22:                     end if
23:                 end for
24:             end if
25:         end while
26:     end while
27:     return $\perp$
28: end function
```

grids, where each cell represents a node and each pair of neighboring cells is connected by an edge of length 1, would potentially place each cell into the priority queue of explored nodes. Algorithm 3 places a room into the priority queue only if it matters, namely when it is a split room from where we have more than one possibility to continue. All other rooms $r$ where $f(r) > B$ do not need to be inserted into the priority queue for the same reason as it was already mentioned in an aforementioned discussion on Algorithm 2. Moreover, we know the goal node in Dijkstra's algorithm, whereas in Algorithm 3 the exit room is unknown and must be discovered. Hence, we cannot minimize the path leading to the exit room. Therefore the distance $|f(r') - B|$ from a continuation room $r'$ to the exit is the only value which we can minimize. This is the reason for which Algorithm 3 does not preclude backtracks, even if they are reduced to the minimum. Although in principle it is only a pseudo-problem, the use of a priority queue implies uncontrolled jumps around a maze.

The third strategy is similar to the first strategy, but instead of pushing the continuations on stack in a fixed predefined way, it randomly permutes the sequence of continuations before placing them on stack. For this reason we call this strategy *randomized.* The strategy is implemented by a Las Vegas algorithm, therefore it always produces a correct answer. However, produced results may vary, provided that there is more than one solution, depending on the random permutation of the continuation sequence. Nevertheless, even if there is only one solution, depending on different random permutations of the continuation sequences subsequently pushed on the stack, the algorithm can follow different paths, potentially with some backtracks, to find the exit.

**Algorithm 4** Randomized Search in a $k$-Dimensional Maze.

**Input:** Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots,x_i,\dots) < f(\dots,x_i+1,\dots)$ for each $i = 1,\dots,k$, and a value $B \in \mathbb{N}$.
**Output:** Coordinates $(x_1,\dots,x_k)$ for which $f(x_1,\dots,x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function RAND_kD_SEARCH($f, B$)
2:      $s \leftarrow \emptyset$                                  ▷ Initialize stack $s$
3:      $m \leftarrow \emptyset$                                  ▷ Initialize memory $m$ of visited rooms
4:      $r \leftarrow (0,\dots,0,n_k-1)$                          ▷ Initialize room $r$
5:      $s$.push($r$)                                             ▷ Put room $r$ on stack $s$
6:      while $s \neq \emptyset$ do                               ▷ While stack $s$ is nonempty
7:          $r \leftarrow$ top($s$)                               ▷ Get room from top of stack $s$
8:          pop($s$)                                              ▷ Pop stack $s$
9:          while $r[j] < n_j$ for $j = 1,\dots,k-1$ & $r[k] \geq 0$ do
10:             if $f(r) = B$ then
11:                 return $r$
12:             else if $f(r) > B$ then
13:                 $r[k] \leftarrow r[k]-1$
14:             else if $f(r) < B$ then
15:                 $v \leftarrow \emptyset$                      ▷ Initialize auxiliary vector $v$
16:                 for $i \leftarrow 1$ to $k-1$ do              ▷ For each potential continuation coordinate
17:                     $r' \leftarrow r$                         ▷ Copy room coordinates
18:                     $r'[i] \leftarrow r'[i]+1$                ▷ Go to a neighboring room
19:                     if $r'[i] < n_i$ & $r' \notin m$ then     ▷ If room within limits and not visited
20:                         $v$.push($r'$)                        ▷ Put continuation room $r'$ in vector $v$
21:                         $m$.insert($r'$)                      ▷ Label room $r'$ as visited
22:                     end if
23:                 end for
24:                 permute($v$)                                  ▷ Permute vector $v$ uniformly at random
25:                 for all $r' \in v$ do                         ▷ For the permuted sequence of continuation rooms
26:                     $s$.push($r'$)                            ▷ Put each continuation room $r'$ in $v$ on stack $s$
27:                 end for
28:             end if
29:         end while
30:     end while
31:     return $\perp$
32: end function
```

The fourth and last strategy is purely *probabilistic.* It does not store the potential continuation rooms in a structure – a stack, a queue, or others – but it makes a probabilistic choice among possible continuations to advance, without the possibility to return back when a dead end is subsequently discovered. For this reason, this strategy is a Monte Carlo algorithm, which can produce a failure answer $\perp$ even if there exists a solution. The probability $p$ to find an exit is equal to $E/(k-1)^S$, where $S$ is the number of splits and $E$ the number of exits in the maze. Potentially any room on the path from the start $s$ to the exit can be a splitting room, therefore the (very coarse) lower bound to find an exit is equal to $E/(k-1)^N$, where $N = \sum_{k=1}^{k} n_i$ is the sum of all bounds. Recall that in the two-dimensional case ($k = 2$) there is only one exit ($E = 1$) and no splits ($S = 0$), therefore the probabilistic algorithm applied to the two-dimensional case becomes totally deterministic with the probability $p = 1$ to find the exit. This probabilistic strategy is presented in Algorithm 5.

**Algorithm 5** Probabilistic Search in a $k$-Dimensional Maze.

**Input:** Function $f\colon \mathbb{N}^k \to \mathbb{N}$ satisfying $f(\dots, x_i, \dots) < f(\dots, x_i + 1, \dots)$ for each $i = 1, \dots, k$, and a value $B \in \mathbb{N}$.

**Output:** Coordinates $(x_1, \dots, x_k)$ for which $f(x_1, \dots, x_k) = B$ or $\perp$ if such coordinates do not exist.

```
1:  function PROBA_kD_SEARCH($f, B$)
2:      $r \leftarrow (0, \dots, 0, n_k - 1)$                          ▷ Initialize room $r$
3:      while $r[j] < n_j$ for $j = 1, \dots, k-1$ & $r[k] \geq 0$ do
4:          if $f(r) = B$ then
5:              return $r$
6:          else if $f(r) > B$ then
7:              $r[k] \leftarrow r[k] - 1$
8:          else if $f(r) < B$ then
9:              $i \leftarrow$ choose$(1, \dots, k-1)$     ▷ Choose a coordinate uniformly at random
10:             $r[i] \leftarrow r[i] + 1$                    ▷ Go to a neighboring room
11:         end if
12:     end while
13:     return $\perp$
14: end function
```

## 4 Applications

A well-suited possibility to implement the function $f\colon \mathbb{N}^k \to \mathbb{N}$ is to use variadic polynomials over natural numbers. A variadic polynomial $p(x_1, \dots, x_k) \in \mathbb{N}[x_1, \dots, x_k]$ ensures the inequality

$$p(x_1, \dots, x_i, \dots, x_k) < p(x_1, \dots, x_i + 1, \dots, x_k) \quad \text{for each } i = 1, \dots, k,$$

since all coefficients of $p$ are natural numbers. The bound will be $b_i = \lceil \sqrt[d_i]{B}/a_i \rceil$ for each variable $x_i$, where $d_i$ is the minimum exponent of $x_i$ and $a_i$ is the coefficient of the monomial where the variable $x_i$ occurs with this exponent in the polynomial $p$. The bounds $b_i$ can be further reduced along the edges of the hypercube. More precisely, for each $i = 1, \dots, k$, the bound $b_i$ can be still reduced if $p(0, \dots, 0, b_i, 0, \dots, 0) > B$ holds.

For the two-dimensional case, we can for instance use the polynomial $f(x, y) = x^3 + y^3$. In fact, this polynomial is the basis for *Taxicab numbers* [10] $\mathrm{Ta}(t)$ for $t = 1, 2, 3, \dots$ If we set the value $B$ to one of Taxicab numbers $\mathrm{Ta}(t)$, the bounds to $m = n = \lceil \sqrt[3]{B} \rceil$, and the starting point to $(0, \lceil \sqrt[3]{B} \rceil)$, Algorithm 1 finds a pair of values $x$ and $y$ whose sum of cubes is equal to $\mathrm{Ta}(t)$. An easy modification of Algorithm 1 produces *all* solutions for a Taxicab number $\mathrm{Ta}(t)$: extend the while-loop condition on Line 4 to "$x \leq m-1$ & $y \geq 0$ & $x \leq y$", replace Line 6 by "**print** $(x, y)$", add the instruction "$y \leftarrow y-1$" between Lines 6 and 7, and finally delete Line 13. In the same way, we can solve other problems mentioned by Silverman in [10], like the problem $x^4 + y^4 = 635318657$ solved by Euler.

For the three-dimensional case, we can use for instance the polynomial $x^3 + y^3 + z^3 = B$ with the bound $\lceil \sqrt[3]{B} \rceil$ for each variable. This problem was considered by Heath-Brown in [3], not only over natural numbers $\mathbb{N}$, but over integers $\mathbb{Z}$.

Another interesting case comes from Gauss' theorem, showing that any natural number can be written as a sum of three triangle numbers, which is equivalent to the statement that any natural number of the form $8n + 3$ can be written as a sum of squares of three odd natural numbers. This is expressed formally by $x^2 + y^2 + z^2 = 8n + 3$ for any $n \in \mathbb{N}$, where the bound for each variable is $\lceil \sqrt{8n+3} \rceil$. Hirschhorn and Sellers studied in [5] the number of *all* solutions for this problem.

An excellent example for an application is the famous Lagrange's theorem [7], showing that any natural number can be written as a sum of four squares. Formally, this can be expressed as $x^2 + y^2 + z^2 + w^2 = B$ for each $B \in \mathbb{N}$, with the bound $\lceil \sqrt{B} \rceil$ for each variable. We should mention here, that our method of generalized Young tableaux is *not* the best algorithm to compute the four squares in Lagrange's sum. More efficient algorithms exist, namely three randomized algorithms of Rabin and Shallit [9] the fastest of which has the running time $O(\log^2 B)$ provided that the Extended Riemann Hypothesis holds, their modification by Pollack and Treviño [8] with running time $O(\log^2 B / \log\log B)$, or that of Bumby [1]. It is just an interesting application for a more general setting. We can also play with Hilbert–Waring theorem [4], which says that for each natural number $d$ there exists an associated natural number $q(d)$ such that every natural number $B$ can be expressed as a sum of at most $q(d)$ natural numbers raised to the power $d$.

The coordinates of the exit in the puzzle from Section 1 are the solution of the equation $3x^{14} + 5y^{15} + 7z^{16} = 1313500139883478322 35$. There is only one solution to this equation.

## 5 Implementation and Benchmarks

All five *Generalized Young Tableaux* algorithms (respective seven if we count the modifications producing all solutions) have been implemented in `C++`. All implementations have a variant with the GNU Multiple Precision Arithmetic Library (GMP) which allows to treat numbers $B$ of any precision. These implementations, together with data files, can be found at the `github` repository `github.com/miki-hermann/gyt`. This directory contains the individual `C++` sources, as well as the data, and a `Makefile`. There are two directories. The first one, entitled `src`, contains the `C++` sources and a `Makefile` which allows to compile the sources without typing the whole compiler command. The correspondence between the algorithms presented in this paper and their implementations is described in Table 1. The second directory, entitled `data`, contains the data files for the implemented algorithms. All implemented algorithms expect the input from `STDIN`, either typed from the keyboard after being prompted, or being piped from a file, for instance by a command like "`gyt < ../data/lagrange01.data`".

The `C++` sources have been compiled by the `g++`, version 10.2.1, with the optimization option "`-O4`". The software has been run with the benchmarks on a Dell computer with an Intel® Core™ i7-9700 CPU @ 3.00GHz × 8 processor, with 16GB RAM, running under Fedora 33. The performance of the software is quite surprizing. For instance, `gyt-2d-gmp`

**Table 1** Corresondence between algorithms and `C++` implementation sources.

| | | |
|---|---|---|
| Algorithm 1 (2-dimensional) | `gyt-2d.cpp` | `gyt-2d-gmp.cpp` |
| Algorithm 1 all solutions | `gyt-2d-all.cpp` | `gyt-2d-all-gmp.cpp` |
| Algorithm 2 (sequential) | `gyt.cpp` | `gyt-gmp.cpp` |
| Algorithm 2 all solutions | `gyt-all.cpp` | `gyt-all-gmp.cpp` |
| Algorithm 3 (priority) | `gyt-pq.cpp` | `gyt-pq-gmp.cpp` |
| Algorithm 4 (randomized) | `gyt-rand.cpp` | `gyt-rand-gmp.cpp` |
| Algorithm 5 (probabilistic) | `gyt-proba.cpp` | `gyt-proba-gmp.cpp` |

computing the first solution of the 7th taxicab number (data file `taxi7.data`) takes only 31.83 seconds and `gyt-2d-all-gmp` computing *all* solutions of the same takes only 1607.15 seconds, i.e., not even 27 minutes.

The performance of the individual algorithms is measured in terms of a maximal stack or queue size, number of splits or choices, number of backtracks, and number of continuation rooms double reached. For the randomized and probabilistic versions, the most advantageous outcome out of 20 runs is presented. Table 2 summarizes the performance of the most interesting data sets. Measuring the execution time would not give a clear picture in this case, since all of them except `lagrange07.data` execute very fast. For instance, the sequential algorithm needs only 4.91 seconds on `ex01.data`, which is quite an involved data set, whereas the priority algorithm needs only 0.57 seconds to execute it, and the randomized algorithm squeezes it down even to 0.25 seconds in the best case.

## 6 Concluding Remarks

When we look at Table 2, we cannot decide which of the four algorithms is the clear winner. As a rule of thumb, the priority algorithm almost always outperforms the sequential algorithm. Notable exceptions are the data sets `ex01.txt` and `ex04.txt`, where the sequential algorithm pushes only 10628 or 329 rooms on stack, whereas the priority driven algorithm inserts 72910 or 213064 rooms into the queue, which indicates that the algorithm "dances" around the search space. Of course, the sequential algorithm makes in the first case more splits and backtracks, as well as it encounters more doubles. However, except for the backtracks, the priority algorithm looses in any category against the sequential version in the second case.

The priority algorithm is a clear winner for `lagrange07.data`. Both the sequential and randomized algorithms terminate with a timeout due to memory exhaustion. The priority algorithm found a solutions without backtracks and only with 16214 splits, beating even the probabilistic algorithm which does not memorize continuation rooms, but needed an incredible number of 19636633 choices. The search space must be densely populated by exits in this case, since the probabilistic algorithm almost always returns a positive anser for this data set. However, the priority algorithm looses against everybody, even against the sequential algorithm, for `ex04.data`.

The randomized and probabilistic algorithms should in principle make the same number of splits, respective choices. This is true many times, but there are cases like `lagrange09.data`, where the search space is small and populated with many solutions (there are actually 1260), but the probabilistic algorithm was not able to reach the minimum achieved by the randomized version. Note, that the priority algorithm beats everybody in this case. Neither doubles are encountered, nor backtracks are triggered.

If we do not count the probabilistic algorithm, when we wish always to receive a correct

**Table 2** Performance of algorithms on chosen data sets.

| | | Algorithm | | | |
|---|---|---|---|---|---|
| Data set | measure | 2:seq | 3:priority | 4:random | 5:proba |
| `gauss-triangle4.data` | max stack/queue | 49465 | 1858 | 1555 | — |
| $x^2+y^2+z^2=$ | splits/choices | 51118 | 1857 | 1554 | 1554 |
| 2446610011 | backtracks | 1659 | 0 | 0 | — |
| | doubles | 2180 | 95152 | 0 | — |
| `gauss-triangle6.data` | max stack/queue | 264651 | 48777 | 16661 | — |
| $x^2+y^2+z^2=$ | splits/choices | 269208 | 49399 | 16660 | 13024 |
| 70039266307 | backtracks | 4563 | 623 | 0 | — |
| | doubles | 6462 | 1521415 | 0 | — |
| `lagrange02.data` | max stack/queue | 22256 | 315 | 173 | — |
| $x^2+y^2+z^2+w^2=$ | splits/choices | 11143 | 218 | 86 | 86 |
| 123456789 | backtracks | 32 | 0 | 0 | — |
| | doubles | 0 | 146 | 0 | — |
| `lagrange07.data` | max stack/queue | timeout | 32383 | timeout | — |
| $x^2+y^2+z^2+w^2=$ | splits/choices | timeout | 16214 | timeout | 19636633 |
| 83461523083775142 | backtracks | timeout | 0 | timeout | — |
| | doubles | timeout | 46 | timeout | — |
| `lagrange09.data` | max stack/queue | 92 | 23 | 45 | — |
| $x^2+y^2+z^2+w^2=$ | splits/choices | 46 | 11 | 22 | 25 |
| 2021 | backtracks | 2 | 0 | 0 | — |
| | doubles | 0 | 0 | 0 | — |
| `ex01.data` | max stack/queue | 10628 | 72910 | 254 | — |
| $2x^3y^2+3y^3z^2+5z^3w^2=$ | splits/choices | 2058405 | 99665 | 74513 | failure |
| 84662255 | backtracks | 2138825 | 50299 | 101210 | — |
| | doubles | 7078935 | 129088 | 331188 | — |
| `ex02.data` | max stack/queue | 6 | 9 | 7 | — |
| $2x^3y^2+3y^3z^2+5z^3w^2=$ | splits/choices | 5 | 5 | 3 | 3 |
| 10 | backtracks | 5 | 2 | 0 | — |
| | doubles | 0 | 0 | 0 | — |
| `ex04.data` | max stack/queue | 329 | 213064 | 731 | — |
| $2x^4+3y^4+5z^4+7w^4+13u^4=$ | splits/choices | 6262 | 180091 | 386 | failure |
| 253930575 | backtracks | 14179 | 0 | 395 | — |
| | doubles | 29559 | 3658440 | 524 | — |
| `maze01.data` | max stack/queue | 16 | 51 | 27 | — |
| $x^{10}+y^{10}+z^{10}=$ | splits/choices | 39 | 51 | 26 | 26 |
| 413575475547 | backtracks | 26 | 11 | 0 | — |
| | doubles | 24 | 22 | 0 | — |
| `maze05.data` | max stack/queue | 3120 | 130 | 36 | — |
| $3x^3y^2+5y^3z^2+7x^2z^3=$ | splits/choices | 3555 | 168 | 35 | 35 |
| 29177953 | backtracks | 3537 | 44 | 0 | — |
| | doubles | 6875 | 184 | 0 | — |
| `maze06.data` (using GMP) | max stack/queue | 21 | 30 | 30 | — |
| $3x^{14}+5y^{15}+7z^{16}=$ | splits/choices | 60 | 29 | 29 | 29 |
| 131350013988347832235 | backtracks | 45 | 0 | 0 | — |
| | doubles | 102 | 0 | 0 | — |

answer, the randomized algorithm beats all others in most cases. However, this is mainly due to the best performance among those 20 runs. Even if the randomized algorithm outperforms the priority one, the advantage is measured only in terms of a constant, provided we do not count the doubles. For these two algorithms, the ratio of maximal stack/queue size ranges within an interval from 1.0 to 3.6, with three notable exceptions, two in favor of the randomized algorithm (`ex01.data` and `ex04.data`) and the other in favor of the priority algorithm (`langrange02.data`). In the same spirit, the ratio of splits ranges within an interval from 0.51 to 4.8. Nevertheless, in the case of `ex04.data`, the randomized algorithm massively surpasses the priority version.

The probabilistic version works well only when the search spaces is populated with many exits. If there is only a small number of exits, namely 1 or 46, respectively, and the search space is huge, as in `ex01.data` and `ex04.data`, the probabilistic algorithm fails to find the exit. The respective randomized algorithm needed 74513 splits for the first one, but only 386 in the second. Given that there are 4 or 5 variables, respectvely, in that problem, which means the correct continuation room is chosen with probability 1/3 or 1/4, respectively, exactly 74513 or even only 386 times, the chances to find the exit by a probabilistic algorithm are practically equal to 0.

The performance of the algorithms essentially depends on the bounds. The data set `maze06.data` has a horribly big exit label $B$, but actually its calculated bound is relatively small: the maximal bound for `maze06.data` is 26. However, the bounds for `lagrange07.data` are all equal to 288897081.

Interested readers are invited to write their own examples and try it out with this software.

## References

1 Richard T. Bumby. Sums of four squares. In David V. Chudnovsky, Gregory V. Chudnovsky, and Melvyn B. Nathanson, editors, *Number Theory: New York Seminar 1991–1995*, pages 1–8. Springer, 1996.

2 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, 2nd edition, 2001.

3 D. R. Heath-Brown. Searching for solutions of $x^3 + y^3 + z^3 = k$. In S. David, editor, *Séminaire de Théorie des Nombres, Paris, 1989–90*, volume 102 of *Progress in Mathematics*, pages 71–76. Birkhäuser, 1992.

4 David Hilbert. Beweis für die Darstellbarkeit der ganzen Zahlen durch eine feste Anzahl $n$-ter Potenzen (Waringsches Problem). *Mathematische Annalen*, 67:81–300, 1909.

5 Michael D. Hirschhorn and James A. Sellers. Partitions into three triangular numbers. *Australasian Journal of Combinatorics*, 30:307–318, 2004.

6 Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison Wesley, 2006.

7 Joseph-Louis Lagrange. Démonstration d'un théorème d'arithmétique. *Nouveaux mémoires de l'Académie royale des sciences et belles-lettres de Berlin*, 123-133, 1770.

8 Paul Pollack and Enrique Treviño. Finding the four squares in Lagrange's theorem. *Integers*, 18A:A15, 2018.

9 Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. *Communications on Pure and Applied Mathematics*, 39:S239–S256, 1986.

10 Joseph H. Silverman. Taxicabs and sums of two cubes. *American Mathematical Monthly*, 100(4):331–340, 1993.

# Force-Directed Embedding of Scale-Free Networks in the Hyperbolic Plane

**Thomas Bläsius**
Karlsruhe Institute of Technology, Germany

**Tobias Friedrich**
Hasso Plattner Institute, University of Potsdam, Germany

**Maximilian Katzmann**
Hasso Plattner Institute, University of Potsdam, Germany

**Abstract**

Force-directed drawing algorithms are the most commonly used approach to visualize networks. While they are usually very robust, the performance of Euclidean spring embedders decreases if the graph exhibits the high level of heterogeneity that typically occurs in scale-free real-world networks. As heterogeneity naturally emerges from hyperbolic geometry (in fact, scale-free networks are often perceived to have an underlying hyperbolic geometry), it is natural to embed them into the hyperbolic plane instead. Previous techniques that produce hyperbolic embeddings usually make assumptions about the given network, which (if not met) impairs the quality of the embedding. It is still an open problem to adapt force-directed embedding algorithms to make use of the heterogeneity of the hyperbolic plane, while also preserving their robustness.

We identify fundamental differences between the behavior of spring embedders in Euclidean and hyperbolic space, and adapt the technique to take advantage of the heterogeneity of the hyperbolic plane.

**2012 ACM Subject Classification** Theory of computation → Random projections and metric embeddings

**Keywords and phrases** force-directed drawing algorithms, spring embedding, hyperbolic space

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.22

**Supplementary Material** *Software (Source Code)*: `https://github.com/maxkatzmann/hyperbolic-spring-embedder.git`
archived at `swh:1:dir:7b9445f64fae3be4bbe3a692c2f94ded0bc600d1`

**Funding** This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Grant No. 390859508.

## 1 Introduction

Network science is an increasingly popular field that ties in with many different research areas such as biology or social science, where researchers examine real-world networks in order to explain observed phenomena. While the goal is typically a mathematical analysis of these graphs, more often than not the first step to understanding the structure of a network, is to gain an intuition by *looking* at it, using a suitable visualization. The most natural way to visualize a graph is to draw its vertices as points and edges as lines between them. In such a drawing, it is typically desirable to have short edges while non-adjacent vertices should be farther apart. On the one hand, this reduces visual clutter. On the other hand, it preserves the typical interpretation of edges as a representation of similarity.

In Euclidean space, the approach that is most commonly used to embed graphs in such a way are *spring embedders* or *force-directed drawing algorithms* [11]. Starting with a random position for each vertex, they simulate physical forces between vertices. *Attractive forces* pull adjacent vertices together and *repulsive forces* push non-adjacent vertices apart.

Due to their basic nature, spring embedders can be applied to all types of graphs. However, they typically struggle if a network contains high-degree vertices that tie together otherwise loosely connected parts of the graph. In the Euclidean plane, there is not enough space close to the high-degree vertices, to make all their edges short while keeping non-connected parts away from each other, often leading to a visualization that resembles a ball of wool. This can be resolved by embedding a network in the hyperbolic plane instead. There, space expands exponentially, i.e., the area and circumference of a disk grows exponentially with its radius. This makes it possible to have many vertices with pairwise large distance being close to a single high-degree vertex.

In fact, a heterogeneous degree distribution (few vertices of high degree and many low-degree vertices) emerges naturally from a hyperbolic geometry which is therefore perceived to be underlying these so-called *scale-free* graphs [3, 13]. In particular, choosing an origin in the hyperbolic plane, one can imagine a vertex's distance to that origin (the vertex's *radius*) to be a measure of popularity: high-degree vertices are placed near the origin and low-degree vertices are farther away. Additionally, the angular distance (around the origin) between two vertices measures their similarity: the *angular coordinates* of adjacent vertices are close. If we now distribute the vertices of a network uniformly within a hyperbolic disk, we obtain few very popular vertices (near the disk center) that are connected to many unpopular vertices (near the disk's boundary) as a result of the exponential expansion of space.

To actually present a hyperbolic drawing to a user, one has to project the hyperbolic plane to the Euclidean plane. This naturally results in a nice fish-eye view that highlights what is currently in the center of the projection [14]. With these advantages of the hyperbolic plane and the popularity of spring embedders, it is not surprising that a spring embedder has been adapted to work for the hyperbolic plane [12]. This approach already produces good results when the embedding is constrained to a small portion of the hyperbolic plane and it showcases the nice fish-eye view effect obtained by the projection. Our goal is to extend their work by considering larger portions of the plane in order to utilize the natural heterogeneity of hyperbolic space to embed scale-free networks. Unfortunately, spring embedders encounter some fundamental problems when confronted with this heterogeneity.

Due to the exponential expansion of space, geodesic lines between pairs of points are bend towards the origin. Thus, moving towards another vertex almost always means moving towards the origin first. Therefore, a less popular vertex (one with low degree) has to be moved closer to the origin to get to where it actually belongs. However, this brings it closer to every other vertex (the smaller the radius, the higher the popularity), which is prevented by the repulsive forces. Thus, even bad embeddings with very long edges are rather stable.

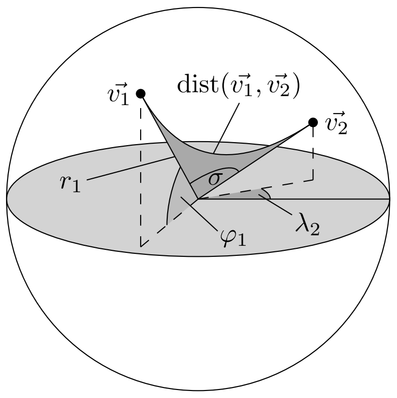
Beyond spring embedders, other approaches have been proposed to generate hyperbolic embeddings. Some of them determine hyperbolic coordinates for the vertices using a spanning tree of the graph, for example to perform greedy routing [5, 10] or to visualize hierarchical data in three-dimensional hyperbolic space [15]. In order to embed graphs with an underlying hyperbolic geometry, the following techniques have been proposed.

An often used approach are maximum likelihood estimation embedders, which try to find the coordinates for the vertices that maximize the probability of the network being generated by an underlying hyperbolic model [18]. In the *step model*, such a hyperbolic random graph is generated by placing $n$ vertices in a hyperbolic disk of radius $R$ and connecting any two vertices with hyperbolic distance at most $R$. Thus, the probability of a graph being produced by the model is 0, if it has edges longer than $R$ or non-edges shorter than $R$. Essentially, the goal of the embedder is to find an embedding such that adjacent vertices are close to each other and non-adjacent vertices are farther apart, which is exactly, what spring embedders pursue as well. *HyperMap* [18] tries to solve this problem, by replaying the network's geometric growth in the hyperbolic plane: starting with high-degree vertices near the origin, each vertex is placed close to its neighbors such that the probability is maximized that the currently embedded graph emerged from the model. An improved version called *HyperMap-CN* was later obtained by additionally taking the common neighborhood of two vertices into account [17]. A further adaption yielded an algorithm with quasilinear running time [2]. Additionally, a novel approach to embedding networks into the hyperbolic plane are *coalescent* embeddings [16]. There, non-linear dimensionality reduction is applied to a matrix representing distances between vertices in the graph. The result is a Euclidean embedding of the network, in which metric distances between vertices match the corresponding distances in the input matrix. The hyperbolic embedding is then obtained by deriving a circular order of the vertices from the Euclidean embedding and combining it with information about the degree of a vertex to approximate its position in the hyperbolic disk.

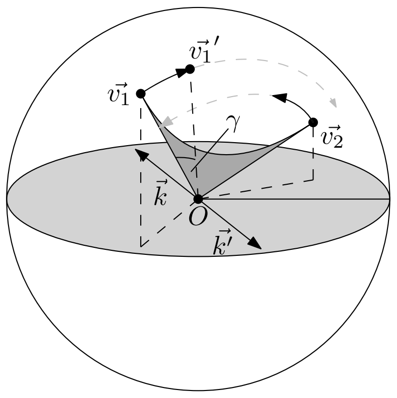
While the above techniques mostly produce good embeddings, they are not very robust. For example, maximum likelihood embedders rely on a good initial embedding of the core (the high-degree vertices) and place vertices with larger distance to the center near their higher-degree neighbors. If the initial embedding of the core is bad (which can happen if there are not enough high-degree vertices), the overall embedding will be bad as well. The coalescent embedder encounters a similar issue if the initial Euclidean embedding is not good.

In the Euclidean plane, spring embedders have proven to be very robust and to quickly produce good embeddings for non-complex graphs. It is still an open problem to adapt this approach to work in the hyperbolic plane in a way that exploits the geometry to better visualize heterogeneous networks. To answer this question, we provide a proof of concept which shows that good hyperbolic embeddings of heterogeneous networks can be obtained using a spring embedder. Our experiments indicate that the quality of the resulting embeddings is on par with the one obtained using the previously mentioned embedding techniques. As a consequence, we believe that our proof of concept lays the groundwork for transferring the ensemble of techniques that have been developed to improve Euclidean spring embedders into the hyperbolic setting.

**Outline and Contribution.** After a brief introduction into the hyperbolic space in Section 2, we describe our embedding process in Section 3. First, we identify a fundamental difference between Euclidean and hyperbolic spring embedders. Basically, the forces simulate a *region of influence* around each vertex: attractive forces pull neighbors into this region and repulsive forces push non-adjacent vertices out of it. In the hyperbolic plane this region of influence has a very different impact on the embedding than in the Euclidean plane. In order to adapt to this difference, we split the forces into two types: one that affects a vertex's popularity and one that tunes the similarity aspect. This, however, reduces the dimensionality of the spring embedder, which makes it harder to escape local optima. We propose to overcome this issue by embedding a network in the three-dimensional hyperbolic space first (Sections 3.1 to 3.4), and transitioning the resulting embedding to the plane afterwards (Section 3.5). In order to evaluate this technique, we conducted experiments on different kinds of networks. In Section 4 our results are compared to the existing embedding techniques mentioned above.



**(a)** The central angle $\sigma$ is used to determine the hyperbolic distance between the two vertices $\vec{v_1}$ and $\vec{v_2}$.



**(b)** Vertex $\vec{v_1}$ is rotated towards $\vec{v_2}$ around $\vec{k}$, which goes through the origin $O$ and is perpendicular to the plane defined by $\vec{v_1}$, $\vec{v_2}$, and $O$.

**Figure 1** Distances and vertex movement in three-dimensional hyperbolic space.

## 2 Preliminaries

**Hyperbolic Plane.** While space expands polynomially in the Euclidean geometry, the expansion is exponential in the hyperbolic geometry. In the hyperbolic plane $\mathbb{H}_2$ a circle with radius $r$ has area $2\pi(\cosh(r) - 1)$ and circumference $2\pi \sinh(r)$, with $\cosh(x) = (e^x + e^{-x})/2$ and $\sinh(x) = (e^x - e^{-x})/2$, both growing as $e^x/2 \pm o(1)$.

A point $p \in \mathbb{H}_2$ is identified using polar coordinates $p = (r, \varphi)$, where $r$ is the *radius* and defines the distance to a designated origin $O$ and $\varphi \in [0, 2\pi)$ is the *angular coordinate* and denotes the angular distance to a reference ray starting at $O$. Given two points $p_1 = (r_1, \varphi_1)$ and $p_2 = (r_2, \varphi_2)$, the hyperbolic distance between them is given by

$$\cosh(\text{dist}(p_1, p_2)) = \cosh(r_1)\cosh(r_2) - \sinh(r_1)\sinh(r_2)\cos(\Delta(\varphi_1, \varphi_2)),$$

where $\Delta(\varphi_1, \varphi_2) = \pi - |\pi - |\varphi_1 - \varphi_2||$ is the *angular distance* between $p_1$ and $p_2$. Finally, given two points with radii $r_1, r_2 \leq R$, respectively, the maximum angular distance such that their hyperbolic distance is still at most $R$ [8, Lemma 3.1], is given by

$$\theta(r_1, r_2) = \arccos\left(\frac{\cosh(r_1)\cosh(r_2) - \cosh(R)}{\sinh(r_1)\sinh(r_2)}\right) = 2e^{\frac{R - r_1 - r_2}{2}}(1 + \Theta(e^{R - r_1 - r_2})). \tag{1}$$

**Three-Dimensional Hyperbolic Space.** In $\mathbb{H}_3$ the coordinates of a vertex $v$ are represented by a tuple $\vec{v} = (r, \lambda, \varphi)$, which describes its radius, latitude, and longitude, respectively. As can be seen in Figure 1a, the hyperbolic distance between two vertices $\vec{v_1}$ and $\vec{v_2}$ is obtained by first determining the central angle $\sigma$ between them, which is given by

$$\cos(\sigma) = \sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\Delta\lambda),$$

for $\Delta\lambda = |\lambda_1 - \lambda_2|$. Afterwards, as in the two-dimensional case, the distance is obtained as

$$\cosh(\text{dist}(\vec{v_1}, \vec{v_2})) = \cosh(r_1)\cosh(r_2) - \sinh(r_1)\sinh(r_2)\cos(\sigma). \tag{2}$$

The rotation of a vertex $\vec{v_1}$ towards another vertex $\vec{v_2}$ around the origin works just as it does in Euclidean space. We therefore convert our polar coordinates to Cartesian coordinates and perform the rotation as if it was in Euclidean space. This rotation is defined by a single

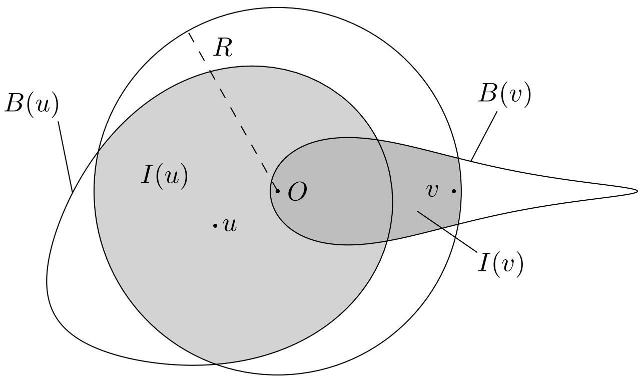


**Figure 2** Two vertices and their regions of influence are shown in a hyperbolic disk of radius $R$. As the distance between $v$ and the origin is larger than the one between $u$ and the origin, $v$'s region of influence $I(v)$ is smaller than $I(u)$, although $B(u)$ and $B(v)$ have the same radius $R$.

vector $\vec{k}$. The direction of $\vec{k}$ denotes the axis that the vertex rotates around. This axis goes through the origin and is perpendicular to the plane defined by the origin $O$ and the points $\vec{v}_1$ and $\vec{v}_2$ (the right-hand rule applies). We obtain $\vec{k} = \vec{v}_1 \times \vec{v}_2$, where $\times$ denotes the cross-product. The length of the vector determines the rotation angle. In Figure 1b one can see how vertex $\vec{v}_1$ is rotated towards $\vec{v}_2$. It is rotated by $|\vec{k}| = \gamma$ around the axis denoted by $\vec{k}$. Note that inverting the rotation axis from $\vec{k}$ to $\vec{k}'$ inverts the direction of the rotation.

Given the Cartesian coordinates of $\vec{v}$ and a rotation vector $\vec{k}$, the rotation $R(\vec{v}, \vec{k})$ is applied using *Rodrigues' formula*, yielding the coordinates of the rotated vector $\vec{v}'$ as

$$\vec{v}' = R(\vec{v}, \vec{k}) = \vec{v} \cos\left(|\vec{k}|\right) + \left(\vec{k} \times \vec{v}\right) \sin\left(|\vec{k}|\right) + \vec{k}\left(\vec{k} \cdot \vec{v}\right)\left(1 - \cos\left(|\vec{k}|\right)\right), \tag{3}$$

where $\times$ and $\cdot$ denote the cross product and dot product, respectively.

## 3 Embedding Process

In a force-directed embedding, the forces simulate a *region of influence* around a vertex $v$, denoted by $B(v)$ (a ball around $v$). Attractive forces pull $v$'s neighbors into $B(v)$ and repulsive forces push non-adjacent vertices out of it.

In the Euclidean plane the size of the region of influence is an input parameter that determines the preferred length of the edges, essentially scaling the embedding. In hyperbolic space the region of influence has a very different impact on the embedding. Recall that a scale-free network emerges naturally, if we assume that its vertices are distributed uniformly within a hyperbolic disk. Since there are no vertices outside of the disk, the region of influence $I(v)$ of a vertex $v$ can be seen as a ball around $v$ that is *constrained* to the disk. This is depicted in Figure 2, where we interpret the polar coordinates in hyperbolic space as polar coordinates in Euclidean space. The size of the region of influence $I(v)$ changes with $v$'s distance to the origin, even though the radius of the ball denoting $I(v)$ is fixed. The closer $v$ is to the origin, the larger is the portion of the disk that is covered by $I(v)$. Consequently, the popularity of $v$ is high. With increasing distance between $v$ and the origin, the size of $I(v)$ decreases, i.e., $v$ becomes less popular (again see Figure 2). The exponential expansion of space now leads to an interesting phenomenon: the heterogeneous degree distribution of scale-free networks emerges naturally by using *the same radius* for the hyperbolic disk (that the vertices are distributed in) and the ball that denotes the region of influence [13].

Unfortunately, this property of hyperbolic geometry impedes the successful application of force-directed embedding algorithms. As mentioned in the introduction, moving a vertex towards another vertex decreases its distance to (almost) all other vertices. Consequently, the resulting repulsive forces prevent this movement, leading to bad stable embeddings. We overcome this problem by dividing the forces into two types, one effecting the popularity (i.e., the radii), and the other only the similarity (i.e., the angular coordinates). That way, vertices no longer move on their geodesic lines. This division, however, reduces the movement of the vertices to one dimension, which decreases the chances of escaping local optima. We circumvent this issue by first embedding the graph in three-dimensional hyperbolic space. In this way, the forces affecting the similarity move the vertices on a two-dimensional surface of a sphere, leading to a similar behavior as in the Euclidean plane.

The general process can now be described as follows. Starting with a random initial embedding of the graph in three-dimensional hyperbolic space, we iteratively apply forces to move adjacent vertices close to each other and non-adjacent vertices farther apart, and in the process adapt the radii of the vertices to tune their region of influence. Once this embedding is stable, a plane is identified, that minimizes the distance to all vertices. Finally, forces are applied to pull the vertices towards this designated plane, resulting in a two-dimensional embedding of the network. In the following sections, we explain each step in greater detail.

## 3.1 Initial Embedding

Recall that we imagine the vertices of our network to be evenly distributed in a disk lying in the hyperbolic plane that has the same radius as the region of influence around each vertex. This radius $R$ can be estimated such that it best fits the size of the given network [3].

We start with an initial embedding $\mathcal{E}$ that assigns each vertex a point in a three-dimensional hyperbolic sphere of radius $R$. To this end, we draw a point uniformly distributed on the surface of the three-dimensional unit sphere, and a radius uniformly at random from $[0, R]$.

## 3.2 Forces

Recall that we have two types of movements. Changing the radial coordinate only affects the size of the region of influence. Rotating a vertex around the origin moves the region of influence without changing its size. To accommodate for these two different movements, we apply two types of forces separately: *popularity forces* affect the radius of a vertex, and *similarity forces* affect the latitude and longitude of a vertex within the sphere.

**Popularity Forces.** A single vertex $v$ with radius $r_v$ is adjusted by comparing its degree $\deg(v)$ with the expected number of vertices in its region of influence, assuming that the current radii $r_1, \ldots, r_n$ of all vertices are fixed. Consider the following random experiment: $n$ vertices with radii $r_1, \ldots, r_n \leq R$ are placed in a hyperbolic disk by choosing their angular coordinates uniformly at random. Without loss of generality we can assume that $\varphi_v = 0$. Now we observe the random variable $X$ which denotes the number of vertices in $I(v)$. Recall that a vertex $u$ is contained in $I(v)$ if its distance to $v$ is at most $R$. Moreover, $\theta(r_u, r_v)$ denotes the maximum angular distance between $u$ and $v$ such that this is true (Equation (1)). Since the probability for this to happen is $\Pr[u \in I(v)] = 2\theta(r_u, r_v)/2\pi$, we can compute $\mathbb{E}[X]$ as

$$\mathbb{E}[X] = \frac{1}{\pi} \sum_{u \in V \setminus \{v\}} \theta(r_v, r_u).$$

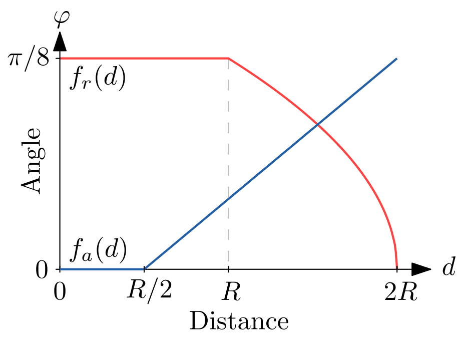


**Figure 3** The functions $f_a(d)$ and $f_r(d)$ determine the magnitude of the attractive and repulsive forces, respectively.

When applying the popularity force on a vertex $v$, we compare $\mathbb{E}[X]$ with $\deg(v)$. If $\mathbb{E}[X] > \deg(v)$, the region of influence of $v$ is too large and $r_v$ is increased. If $\deg(v) = \mathbb{E}[X]$, $r_v$ is not changed. Otherwise it is decreased. In preliminary experiments we determined that adjusting the radii using a fixed, small step size delivered the best results.

We note, that the radius of a vertex is not bounded from above, i.e., the radius $R$ of our disk (and therefore the size of the region of influence) can change during the embedding process and is set to be the maximum radius of a vertex in the current embedding. That way we can accommodate for potential errors that were made during the initial estimation of $R$.

**Similarity Forces.** The similarity forces move vertices without changing their radii. This allows us to move vertices close to each other, without getting closer to all other vertices.

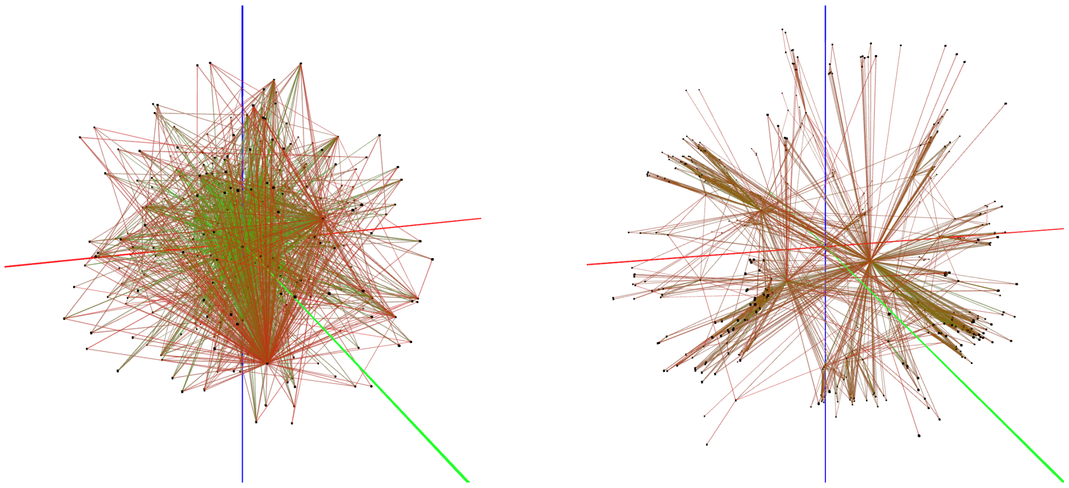
Let $u \neq v$ be two vertices with coordinates $\vec{u}$ and $\vec{v}$, respectively. In the following, we observe the force that is caused by $u$ and acts on $v$. Formally, the force is determined using a function $f\colon \mathbb{H}_3 \times \mathbb{H}_3 \to \mathbb{R}^3$. The resulting vector describes the axis around which the vertex, that the force acts on, is rotated. The direction of the force $f(\vec{u}, \vec{v})$ is perpendicular to the plane containing $u, v$ and the origin, and is given by $\vec{k} = (\vec{v} \times \vec{u})/(|\vec{v} \times \vec{u}|)$. If the force is repulsive, we invert the direction of the rotation by rotating around $-\vec{k}$. The length of $f(\vec{u}, \vec{v})$ describes the magnitude of the force and was chosen to match the angle $\varphi$ that $v$ is rotated by. It depends on the hyperbolic distance $\mathrm{dist}(\vec{u}, \vec{v})$ between the two vertices (Equation (2)). We define two functions $f_a$ (for attractive forces) and $f_r$ (for repulsive forces) that map the distance $d \in [0, 2R]$ to the magnitude $\varphi \in [0, \pi/8]$ of the force. In preliminary experiments, the upper bound of $\pi/8$ for $\varphi$ proved to be useful in avoiding very large jumps. As they delivered the best results in preliminary experiments, we chose the two functions to be

$$f_a(d) = \begin{cases} 0, & d \leq R/2, \\ \pi/8 \cdot \left(\frac{2d-R}{3R}\right), & \text{otherwise} \end{cases} \quad \text{and} \quad f_r(d) = \begin{cases} \pi/8, & d \leq R, \\ \pi/8 \cdot \left(2 - \frac{d}{R}\right)^{1/2}, & \text{otherwise,} \end{cases}$$

which are depicted in Figure 3. Note that the repulsive force is as strong as possible if the distance between the two vertices is less than $R$, i.e., when $v$ is in the region of influence of $u$.

## 3.3 Force Application

After the initial random embedding we alternatingly apply batches of popularity and similarity forces. In each iteration we compute the forces that act on the vertices and rotate them accordingly. Additionally, some precautions are taken that help stabilize the embedding. In a single iteration $i$, the total force that acts on a vertex $v$ is determined, by first summing



**(a)** The initial embedding with random coordinates for all vertices.

**(b)** A stable embedding after applying popularity and similarity forces.

**Figure 4** Two three-dimensional embeddings (before and after applying forces) of a hyperbolic random graph with 492 vertices.

the forces that are caused by all other vertices, which we denote with $\vec{k}_i(v)$. In order to stabilize the embedding, $\vec{k}_i(v)$ is then scaled using a *temperature* $\tau_i \in (0, 1]$ that decreases as $\tau_i = 0.975 \cdot \tau_{i-1}$ in every iteration. Additionally, to prevent oscillations, a *velocity* is simulated by adding a portion $\nu$ (we use $\nu = 1/2$) of the forces that acted on $v$ in the previous iteration. Taken together, we obtain the rotation vector $\vec{\kappa}_i(v)$, describing the total force that acts on $v$ in iteration $i$ as $\vec{\kappa}_i(v) = \nu\vec{\kappa}_{i-1}(v) + \tau_i\vec{k}_i(v)$ and rotate $v$ accordingly.

## 3.4 Stability

After every iteration $i$ we obtain a new embedding $\mathcal{E}_i$ describing the current positions of all vertices. From these positions we can derive a *potential* $\rho_i$ which describes how unstable the embedding is after iteration $i$. The potential is defined as the sum of the strengths of all forces and can be computed as $\rho_i = \sum_{v \in V} |\vec{\kappa}_i(v)|$. After every iteration we compute the potential $\rho_i$ and compare it with previous $\rho_j$ for $j < i$ to detect whether the potential is decreasing, meaning the embedding is getting more stable. Figure 4 compares the initial embedding of a graph with a stable one, obtained after iteratively applying the forces.

After iteration $i$ the process stops if the potential has decreased in the last couple of iterations, meaning $\rho_j > \rho_{j+1}$, for $i - 10 < j < i$ and the difference to the previous potential $\rho_{i-1}$ drops below a given stability threshold. Additionally, the process stops if the number of iterations has exceeded a predefined threshold. Usually, the potential fluctuates in the beginning, since the temperature is high, but as the temperature decreases over time, so does the potential and the embedding becomes stable eventually.

## 3.5 Transition to the Plane

Once the three-dimensional embedding is stable, we convert it to a two-dimensional embedding, by first determining the plane that contains the origin and minimizes the distance to all vertices, using principal component analysis. Then, the embedding is rotated such that this plane aligns with the plane $P = \{(r, \lambda, \varphi) \mid \varphi = 0\}$. Afterwards, in addition to the forces

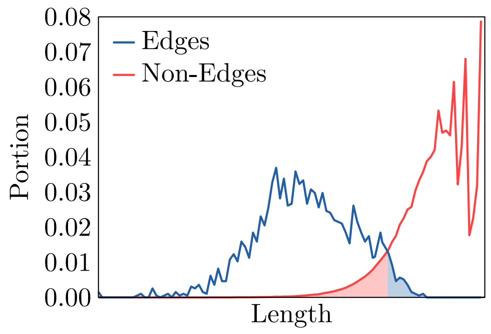


**Figure 5** An edge-length histogram that is used to measure the embedding quality. It shows the portion of edges (blue) and non-edges (red) of a given length. The filled regions denote the errors.

that were used so far, we apply forces that pull the vertices towards $P$. To this end, for every vertex $v$ with coordinate $\vec{v} = (r, \lambda, \varphi)$, we introduce a virtual vertex $v'$ with coordinate $\vec{v'} = (r, \lambda, 0)$. The total force $\vec{\kappa}_i(v)$ that acts on $v$ is now determined as before, with the addition of a force that attracts $v$ towards $v'$. In particular, we obtain the new position of $v$ by rotating it around $\vec{\kappa}_i(v) + \vec{\kappa}'_i(v)$. The additional force $\vec{\kappa}'_i(v) = \tau_i \cdot \pi/15 \cdot (\vec{v}' \times \vec{v})/(|\vec{v}' \times \vec{v}|)$ is independent of the distance between $v$ and $v'$, constantly pulling $v$ towards the plane. By still applying the popularity and similarity forces, we try to preserve the quality of the existing embedding while transitioning it towards the plane. This process ends as soon as the average distance between the vertices and the plane drops below a certain threshold or a maximum number of iterations is reached. Then, all vertices are moved onto the plane, yielding the final two-dimensional embedding.

## 4 Experiments

In order to evaluate whether the adapted spring embedder works and how it compares to existing embedding techniques, we implemented it in C++[1], using *Eigen* [7] to perform the principal component analysis needed for the transition to the two-dimensional plane. Afterwards we ran experiments on the largest component of 16 real-world networks [19] and 18 hyperbolic random graphs with different parameter configurations. In addition to our spring embedder, we considered the Euclidean spring embedder *FMMM* [9] (using the implementation found in *OGDF* [4]), as well as the maximum likelihood estimation embedder HyperMap-CN [17], the previously mentioned quasilinear adaptation, which we abbreviate with *BFKL* [2], and the coalescent embedder [16].

**Embedding Quality.** Recall, that the goal of an embedding technique is to place adjacent vertices closely together and non-adjacent ones farther apart. In order to measure how well this goal is achieved we introduce the *edge-length histogram*, as can be seen in Figure 5. It is based only on a graph's adjacency information and an embedding. This allows us to evaluate embeddings of real-world networks, where ground truth data is usually not available. The *edge curve* (blue) denotes the relative number of edges of a given length. The *non-edge curve* (red) does the same for non-edges. We measure the error of the embedding by determining

[1] Our code is available at `https://github.com/maxkatzmann/hyperbolic-spring-embedder.git` (archived at `swh:1:dir:7b9445f64fae3be4bbe3a692c2f94ded0bc600d1`).

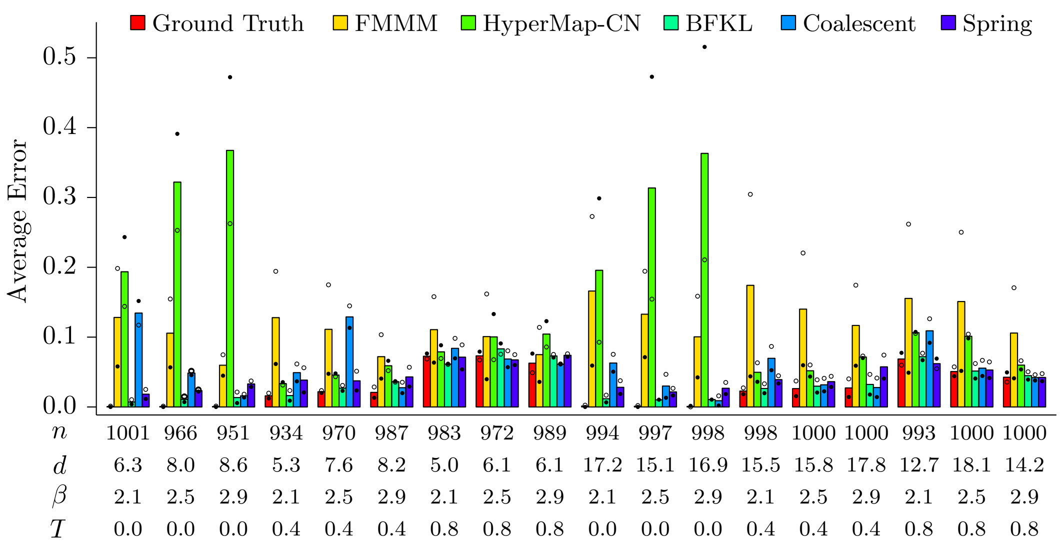


**Figure 6** Errors obtained using different embedding techniques on hyperbolic random graphs. Filled circles indicate edge errors. Hollow circles indicate non-edge errors. Bars denote their average. The graphs differ in the number of vertices $n$, the average degree $d$, the power-law exponent of the degree distribution $\beta$, and the temperature $T$.

the area under the minimum of the two curves. The *edge error* is the corresponding area whenever the edge curve takes on the minimum (filled blue region). The *non-edge error* (denoted by the filled red region) is defined analogously. The *average error* is the average of the edge error and the non-edge error and the *balancing error* is the difference between them.

In a perfect embedding, the edge-length histogram would show a peak in the edge curve to the left of another peak in the non-edge curve without any overlap, meaning all edges are shorter than all non-edges. In other words, all vertices have their neighbors inside their region of influence and all non-neighbors outside of it. Consequently, the average error and the balancing error are both 0%. Whenever the two curves overlap, errors were made. The average error gives some general hints about the overall quality of the embedding. But assume an embedding has an average error of 50% (the worst error possible). This could mean, for example, that the edge error and the non-edge error are both close to 50%, or (in the extreme case where all vertices are placed on the same point) that the edge error is 0% and the non-edge error is 100%. In that case the latter embedding is arguably worse than the first one, which is revealed by looking at the balancing error.

**Hyperbolic Random Graphs.** Figure 6 shows the errors obtained in our experiments on hyperbolic random graphs. These graphs are sampled by placing $n$ vertices uniformly at random in a hyperbolic disk of radius $R = 2\log n + C$, where $C$ controls the average degree $d$ of the network. Furthermore, the power-law exponent $\beta \in (2, 3)$ impacts the degree distribution. The smaller $\beta$, the denser is the core of the network. In the step model, any two vertices are connected, if the distance between them is at most $R$. In a relaxed version, a temperature $T$ (typically between 0 and 1) is introduced, which allows for long-range edges (and short non-edges) by smoothing the step function. This influences the clustering coefficient of the generated network: the smaller $T$, the higher the clustering. We note that this notion of temperature is independent of the one used to stabilize the embedding.

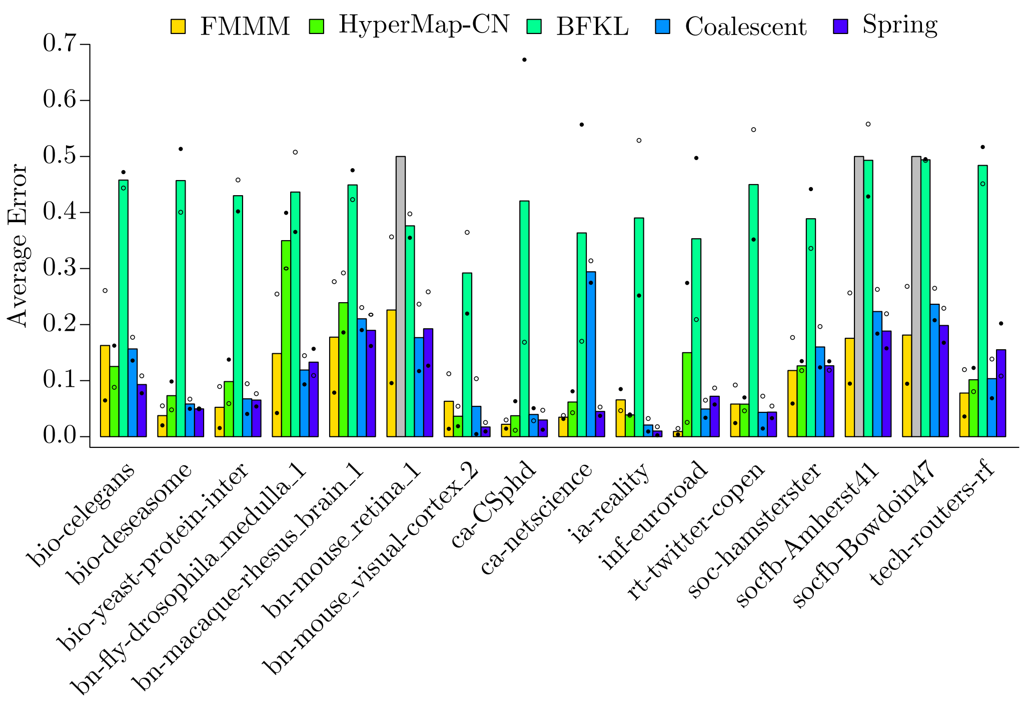


**Figure 7** Errors obtained using different embedding techniques on the largest component of several real-world networks. Filled circles indicate edge errors. Hollow circles indicate non-edge errors. Bars denote their average. Grey bars indicate that the embedding process did not finish within 96 hours.

On these networks the considered techniques that generated embeddings in the hyperbolic plane delivered mostly good results. The mean errors of the different techniques over all generated graphs are:

| | Ground Truth | FMMM | HyperMap-CN | BFKL | Coalescent | Spring |
|---|---|---|---|---|---|---|
| Average Error | 2.8% | 11.9% | 14.5% | 3.4% | 5.9% | 4.3% |
| Balancing Error | 1.0% | 13.4% | 8.2% | 1.0% | 2.2% | 1.8% |

The BFKL embedder was the best one, delivering results that are very close to the ground truth (the coordinates sampled during the graph generation). With the exception of FMMM and HyperMap-CN, the remaining embedders also produced very good results.

As expected, FMMM had trouble fitting the hyperbolic random graphs into the Euclidean plane. Figure 6 shows that on most networks FMMM obtained a small edge error at the expense of a large non-edge error, yielding a balancing error of 13.4% on average. This does not come as a surprise as there is simply not enough space in the Euclidean plane to keep non-edges long while trying to obtain short edges. Unfortunately, HyperMap-CN seemed to have issues with low temperatures. Excluding graphs with $T = 0$, HyperMap-CN obtained an average error of 7.2%.

The hyperbolic spring embedder produced embeddings that are on par with the ground truth and sometimes even better. The average error and the balancing error are small on average, with 4.3% and 1.8%, respectively. This shows that spring embedders can be adapted to take advantage of the intrinsic heterogeneity of the hyperbolic plane to produce good embeddings of heterogeneous networks.

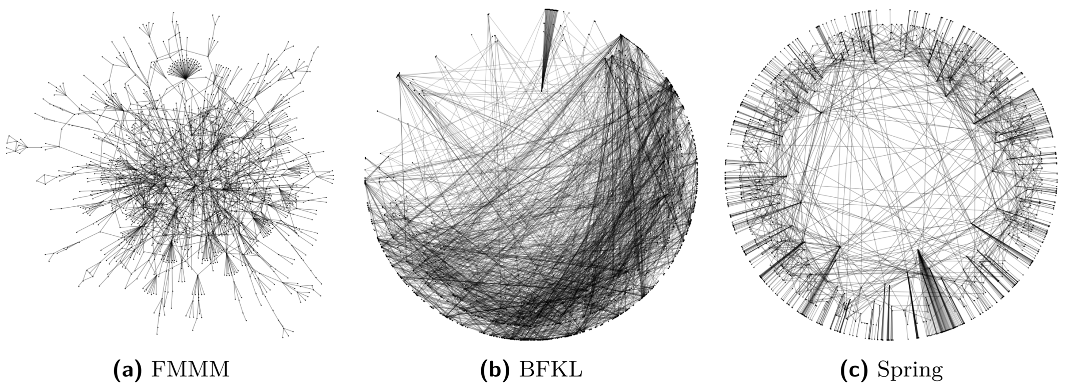


**Figure 8** Embeddings of the largest component (containing 1458 vertices) of the *bio-yeast-protein-inter* network, obtained using different techniques.

**Real-World Networks.** Figure 7 shows the errors of the embeddings obtained using the different techniques on real-world networks. The mean errors over all graphs are:

| | FMMM | HyperMap-CN | BFKL | Coalescent | Spring |
|---|---|---|---|---|---|
| Average Error | 10.1% | 11.5% | 42.1% | 12.6% | 10.1% |
| Balancing Error | 10.9% | 6.6% | 15.8% | 5.5% | 4.1% |

Overall the embedders produced good results, with the exception of BFKL with an average error of 42.1%. As noted by the authors, this technique excels on larger graphs [2]. Thus, one explanation for the bad performance is the constraint to smaller graph sizes ($< 7000$ vertices) that was necessary to compensate for the running time of other techniques. In fact, while HyperMap-CN produced good results in general, it also encountered three instances where it did not finish the embedding process within 96 hours. The remaining techniques performed well with an average error of a little over 10%.

Figure 7 shows that FMMM encountered the same difficulties as on the hyperbolic random graphs and was often not able to maintain long non-edges while making the edges short (except for *inf-euroroad*, a road-network where the underlying geometry is arguably rather Euclidean). Figure 8 depicts embeddings of the *bio-yeast-protein-inter* network.[2] In the FMMM embedding (Figure 8a) one can see how higher degree vertices are placed near the center of the embedding and with them their low-degree neighbors. As a result, the center part is very cluttered and the structure of the network is unclear.

Likewise, the maximum likelihood estimation techniques were sometimes not able to produce embeddings that convey the structure of the network. They usually start the embedding process by first embedding the core (high-degree vertices that are supposed to be close to the origin) and determine the positions of later vertices based on the already placed ones. As can be seen in Figure 8b, this fails if there are not enough vertices in the core. In that case, the earlier stages of the embedding are rather arbitrary and later vertices can not compensate for the initial errors.

On the other hand, the natural approach of applying forces between the vertices to obtain embeddings that reflect the structure of the network well, proved to be very robust. In 11 of the 16 embeddings the hyperbolic spring embedder delivered the best results and was not far off on the remaining instances, yielding the best average and balancing errors

[2] Further embedding evaluations can be found in Appendix A.

of 10.1% and 4.1%, respectively. Figure 8c shows the hyperbolic spring embedding of the *bio-yeast-protein-inter* network. One can observe a good trade-off between edges being too long and non-edges being too short, resulting in a drawing where higher degree vertices are placed towards the center and their low-degree neighbors are close to them near the disk's boundary. This separation between high- and low-degree vertices helps in reducing the clutter, making it easier to see which vertices actually belong together.

## 5 Conclusion

It was previously observed that spring embedders encounter a fundamental problem when being subjected to the natural heterogeneity of hyperbolic space. After explaining the core difficulties we proposed a way to circumvent them: The application of the forces typically has close to no impact on the position of a vertex as movement along geodesic lines simultaneously affects its region of influence in two ways (popularity and similarity), which can be overcome by differentiating between two types of forces and increasing the dimensionality in order to better escape local optima. Our experiments indicate that the resulting approach produces embeddings that are on par with other commonly used hyperbolic embedding techniques.

Adapting the standard force-directed embedding approach to work in hyperbolic space paves the way for translating well known techniques that improve the quality of Euclidean spring embedders into the hyperbolic setting. These include the use of geometric data structures [1], the multilevel strategy of the previously mentioned FMMM embedder [9], and the application of random sampling in order to improve the running time [6]. We believe that these techniques can also be used to extend our proof of concept in order to further improve its performance.

## References

1 Josh Barnes and Piet Hut. A hierarchical $O(NlogN)$ force-calculation algorithm. *Nature*, 324(6096):446–449, 1986. `doi:10.1038/324446a0`.

2 Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *24th Annual European Symposium on Algorithms, ESA 2016*, pages 16:1–16:18, 2016. `doi:10.4230/LIPIcs.ESA.2016.16`.

3 Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, 1(62), 2010. `doi:10.1038/ncomms1063`.

4 Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In *Handbook of Graph Drawing and Visualization*, chapter 17. CRC Press, 2014.

5 D. Eppstein and M. T. Goodrich. Succinct greedy geometric routing using hyperbolic geometry. *IEEE Transactions on Computers*, 60(11):1571–1580, 2011. `doi:10.1109/TC.2010.257`.

6 R. Gove. A Random Sampling $O(n)$ Force-calculation Algorithm for Graph Layouts. *Computer Graphics Forum*, 38(3):739–751, 2019. `doi:10.1111/cgf.13724`.

7 Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

8 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering. In *Automata, Languages, and Programming*, pages 573–585, 2012. `doi:10.1007/978-3-642-31585-5_51`.

9 Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing, 12th International Symposium, GD 2004*, pages 285–295, 2004. `doi:10.1007/978-3-540-31843-9_29`.

10 Robert Kleinberg. Geographic routing using hyperbolic space. In *26th IEEE International Conference on Computer Communications*, pages 1902–1909, 2007. `doi:10.1109/INFCOM.2007.221`.

11 Stephen G. Kobourov. Force-directed drawing algorithms. In *Handbook of Graph Drawing and Visualization*, chapter 12. CRC Press, 2014.

12 Stephen G. Kobourov and Kevin Wampler. Non-euclidean spring embedders. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):757–767, 2005. doi:10.1109/TVCG.2005.103.

13 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, 2010. doi:10.1103/PhysRevE.82.036106.

14 John Lamping, Ramana Rao, and Peter Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 401–408, 1995. doi:10.1145/223904.223956.

15 Tamara Munzner. H3: Laying out large directed graphs in 3d hyperbolic space. In *Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, pages 2–10, 1997. doi:10.1109/INFVIS.1997.636718.

16 Alessandro Muscoloni, Josephine Maria Thomas, Sara Ciucci, Ginestra Bianconi, and Carlo Vittorio Cannistraci. Machine learning meets complex networks via coalescent embedding in the hyperbolic space. *Nature Communications*, 8(1):1615, 2017. doi:10.1038/s41467-017-01825-5.

17 Fragkiskos Papadopoulos, Rodrigo Aldecoa, and Dmitri Krioukov. Network geometry inference using common neighbors. *Physical Review E*, 92(2):022807, 2015. doi:10.1103/PhysRevE.92.022807.

18 Fragkiskos Papadopoulos, Constantinos Psomas, and Dmitri Krioukov. Network mapping by replaying hyperbolic growth. *IEEE/ACM Transactions on Networking*, 23(1):198–211, 2015. doi:10.1109/TNET.2013.2294052.

19 Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL: http://networkrepository.com.

## A Additional Embedding Comparisons

On the following pages we show a few more comparisons of embeddings produced using the various techniques as explained in Section 4.

### Hyperbolic Random Graph (Figure 9)

Hyperbolic random graphs are sampled by distributing vertices uniformly at random in a hyperbolic disk and connecting any two that are sufficiently close. Figure 9a depicts such a graph using the sampled vertex positions. Since the underlying geometry of the network is hyperbolic and not Euclidean, FMMM has no chance in finding an embedding that represents the structure of the graph well (Figure 9b). The densely connected high-degree vertices as well as their lower-degree neighbors are all placed close to each other, yielding a cluttered embedding. Apart from the coalescent embedding (Figure 9e) the other techniques obtained embeddings that resemble the ground truth well. This is reflected by the corresponding error diagram in Figure 6 for the network with $n = 970$, $d = 7.6$, $\beta = 2.5$, and $T = 0.4$.

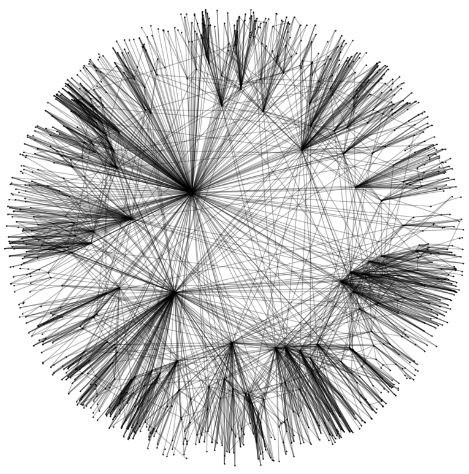
### rt-twitter-copen (Figure 10)

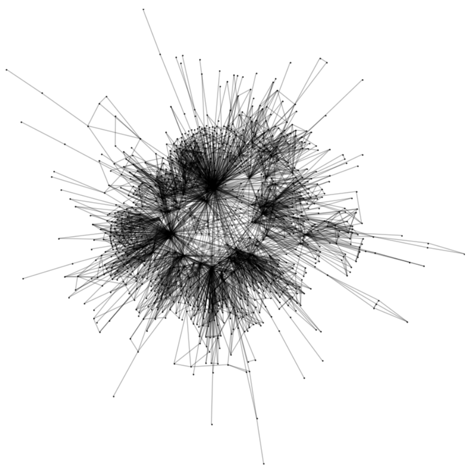
Similarly to the embeddings in Figures 8a and 9a, FMMM succeeds in making the edges short, which leads to a small average error. However, in order to achieve this, many of the non-adjacent lower-degree vertices are placed close to the high-degree neighbors and, thus, close to each other. Consequently, the small edge-error is counteracted by a comparatively large non-edge error (denoted by the hollow-circle in Figure 7). While BFKL seemed to be unable to find a good initial placement of the higher-degree vertices, the remaining techniques produced reasonable results. Most notably, the spring embedder distributed the low-degree vertices near the boundary of the embedding and placed them close to their higher-degree neighbors. As a result, it obtained the best combination of average and balancing error.

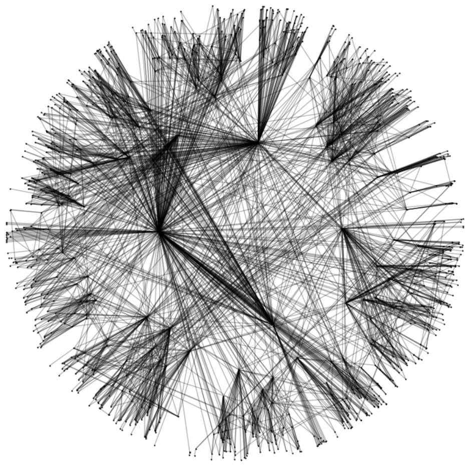
### inf-euroroad (Figure 11)

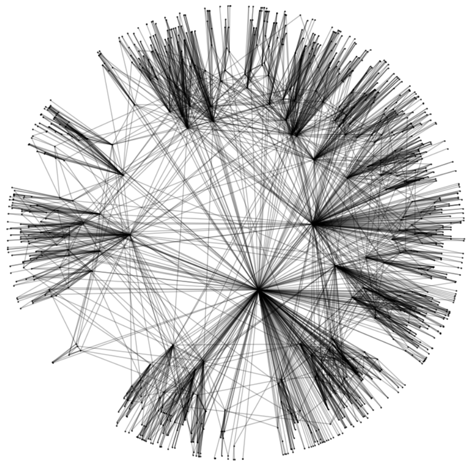
The underlying geometry of the road network *inf-euroroad* is arguably rather Euclidean than hyperbolic. Therefore, it is no surprise that FMMM, the only considered embedder that works with Euclidean geometry, obtained the best results here. The corresponding embedding in Figure 11a resembles that of a road network and is, therefore, a good representation of the structure of the graph. This is reflected by very small average and balancing errors, as shown in Figure 7.

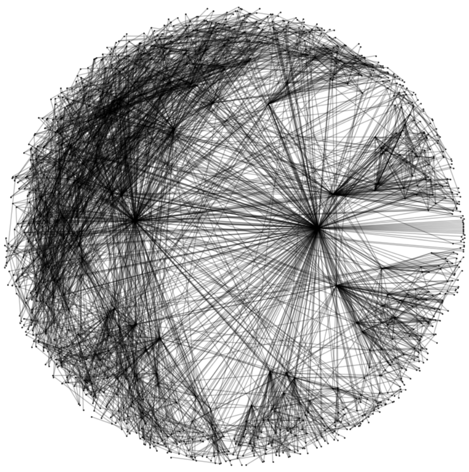
In heterogeneous networks we interpret the degrees of the vertices as a measure of their popularity. In hyperbolic embeddings this is represented by the radial coordinates of the vertices: a vertex that is closer to the center has a larger popularity (higher degree). However, road networks are rather homogeneous, meaning most vertices have similar degree and, thus, similar popularity. The hyperbolic embedders represent this by placing all vertices in a ring close to the boundary of the embedding. The differences in the qualities of the embeddings is, thus, revealed by how well the techniques captured the similarities of the vertices, i.e., how close adjacent vertices are positioned in the ring. This is represented by the amount of edges that go through the center of the embedding. Clearly, the coalescent embedder captured this best, followed closely by the spring embedder, with HyperMap-CN and BFKL rather far off. The exact same ranking is reflected in the corresponding average errors in Figure 7.


**(a)** Ground Truth


**(b)** FMMM


**(c)** HyperMap-CN


**(d)** BFKL


**(e)** Coalescent

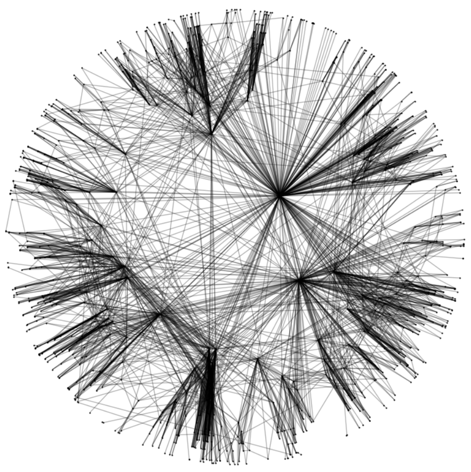

**(f)** Spring

**Figure 9** Embeddings of the largest component (containing 970 vertices) of a hyperbolic random graph (with average degree 7.6, power-law exponent 2.5, and temperature 0.4), obtained using different techniques.

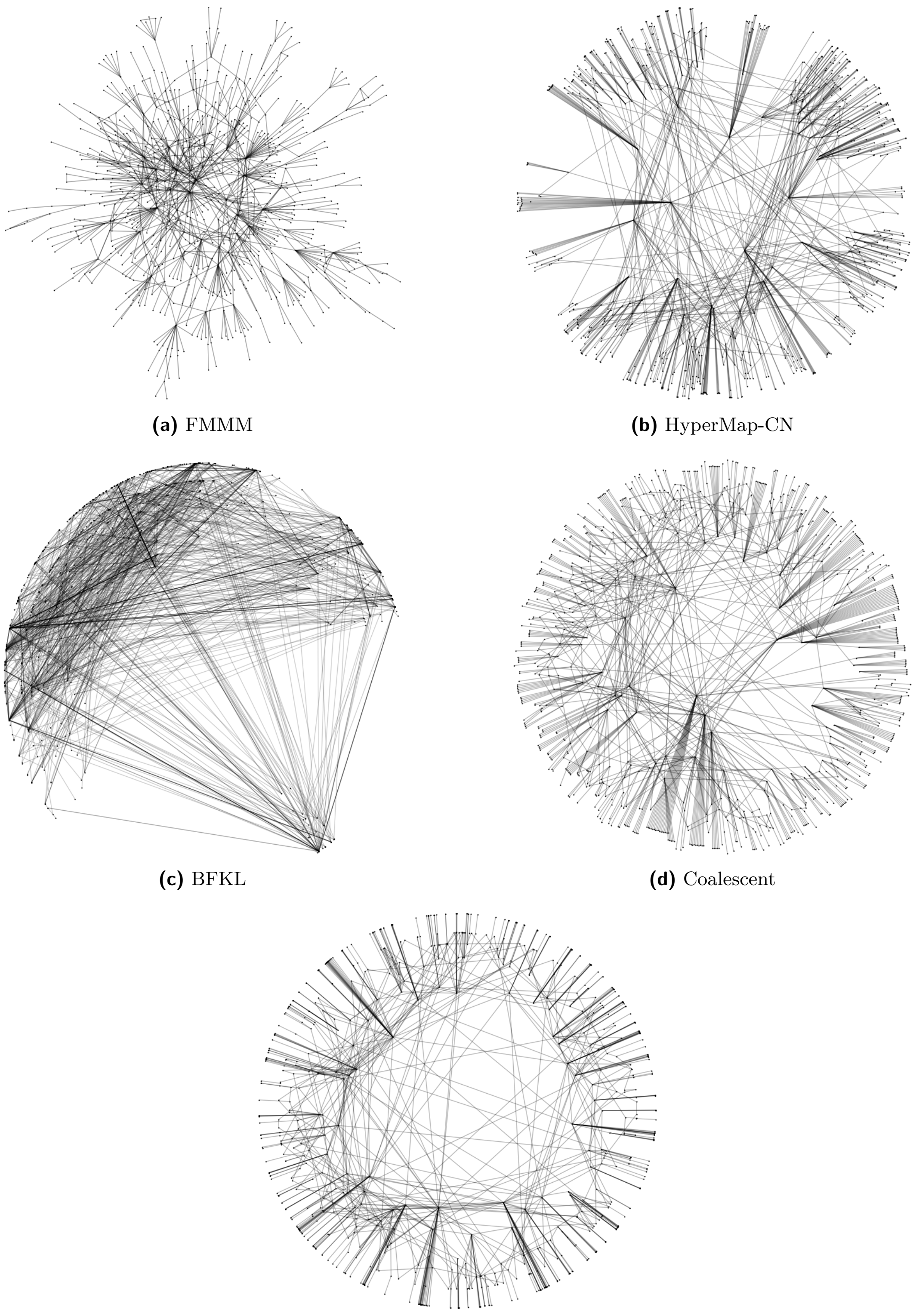
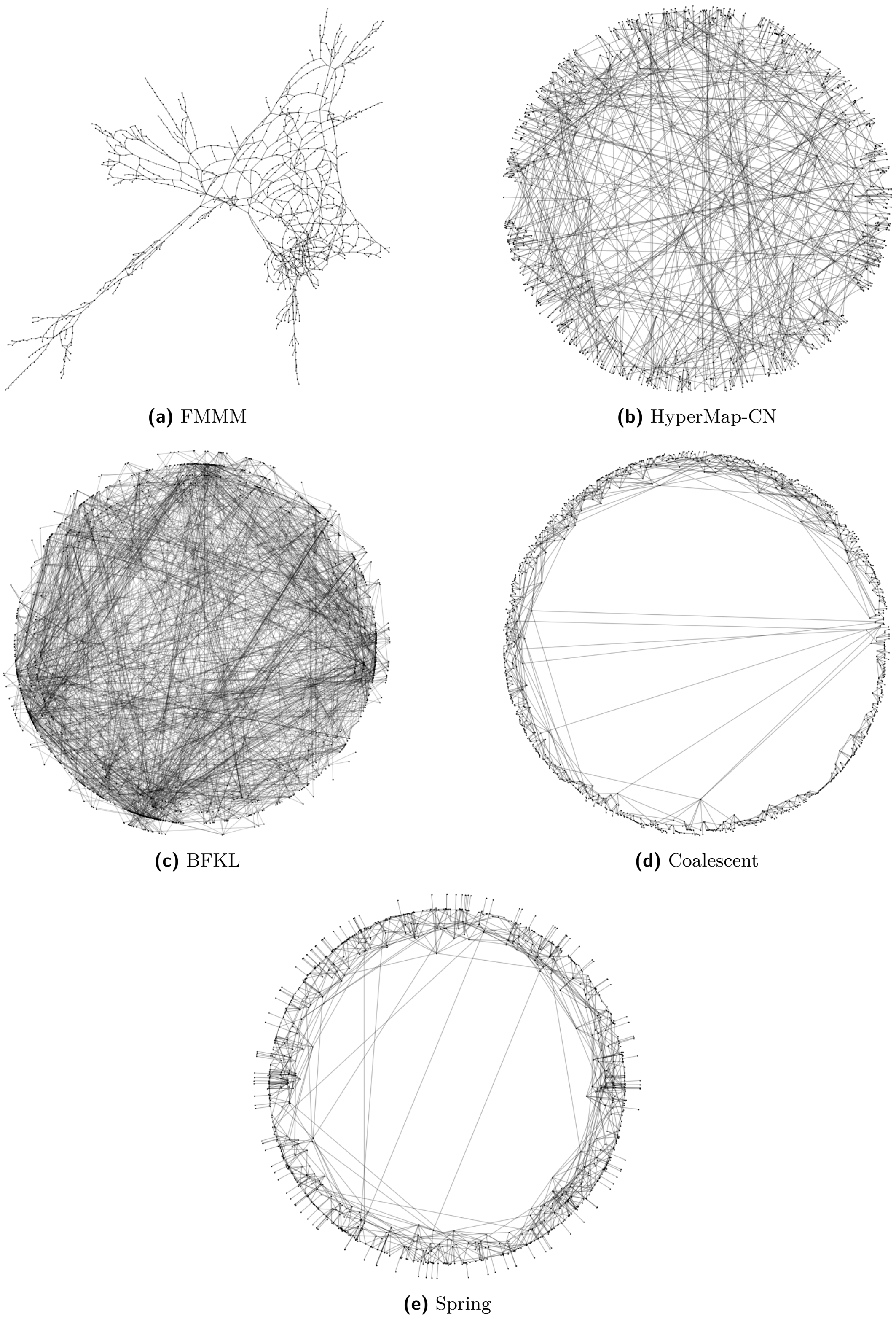

**(a)** FMMM **(b)** HyperMap-CN **(c)** BFKL **(d)** Coalescent **(e)** Spring

**Figure 10** Embeddings of the largest component (containing 761 vertices) of the *rt-twitter-copen* network, obtained using different techniques.



**(a)** FMMM

**(b)** HyperMap-CN

**(c)** BFKL

**(d)** Coalescent

**(e)** Spring

**Figure 11** Embeddings of the largest component (containing 1174 vertices) of the *inf-euroroad* network, obtained using different techniques. Here, FMMM obtained the best representation of the structure of the network.

# An Experimental Study of External Memory Algorithms for Connected Components

**Gerth Stølting Brodal** ✉
Aarhus University, Denmark

**Rolf Fagerberg** ✉
University of Southern Denmark, Odense, Denmark

**David Hammer** ✉
Goethe Universität Frankfurt, Germany
University of Southern Denmark, Odense, Denmark

**Ulrich Meyer** ✉
Goethe Universität Frankfurt, Germany

**Manuel Penschuck** ✉
Goethe Universität Frankfurt, Germany

**Hung Tran** ✉
Goethe Universität Frankfurt, Germany

## Abstract

We empirically investigate algorithms for solving Connected Components in the external memory model. In particular, we study whether the randomized $O(\mathrm{Sort}(E))$ algorithm by Karger, Klein, and Tarjan can be implemented to compete with practically promising and simpler algorithms having only slightly worse theoretical cost, namely Borůvka's algorithm and the algorithm by Sibeyn and collaborators. For all algorithms, we develop and test a number of tuning options. Our experiments are executed on a large set of different graph classes including random graphs, grids, geometric graphs, and hyperbolic graphs. Among our findings are: The Sibeyn algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the Connected Components setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. Borůvka's algorithm is not competitive with the two others.

**2012 ACM Subject Classification** Mathematics of computing → Paths and connectivity problems; Theory of computation → Graph algorithms analysis

**Keywords and phrases** Connected Components, Experimental Evaluation, External Memory, Graph Algorithms, Randomization

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2021.23

**Supplementary Material** *Software (Source Code)*: `https://ae.cs.uni-frankfurt.de/r/p/em-cc` archived at `swh:1:dir:a11f45c7dd725022c9898f14f1b2a9afe1136b9a`

**Funding** Based upon work partially performed while attending AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, organized by Nodari Sitchinava and in part supported by the National Science Foundation under Grant No. CCF-1930579. Gerth Stølting Brodal and Rolf Fagerberg have been supported by the Independent Research Fund Denmark under grants 9131-00113B and DFF-7014-00041, respectively. David Hammer, Ulrich Meyer, Manuel Penschuck, and Hung Tran have been supported by the Deutsche Forschungsgemeinschaft (DFG) under grants ME 2088/4-2 (SPP 1736 Algorithms for Big Data) and ME 2088/5-1.

**Acknowledgements** Extensive calculations on the Goethe-HLR high-performance computer of the Goethe University Frankfurt were conducted for this research. The authors would like to acknowledge the CSC team for their support. We would also like to thank Peter Sanders for valuable feedback on an earlier draft of this paper.

## 1 Introduction

The Connected Components (CC) problem is a fundamental algorithmic task on undirected graphs and has a large number of applications including web graph analysis, communication network design, image analysis, and clustering in computational biology. CC may be viewed as a smaller sibling of the Minimum Spanning Forest (MSF) problem defined on weighted, undirected graphs – any algorithm solving MSF and able to return the trees of the forest one by one can be used to solve CC by first assigning arbitrary edge weights.

In internal memory, CC is simple to solve in linear time by DFS or BFS. A long-standing open problem is whether MSF can also be solved deterministically in linear time. The large body of work devoted to the question (see e.g. the references in [22]) indicates that in internal memory, MSF is harder to tackle than CC, at least in terms of the algorithmic sophistication needed (and potentially also in terms of the asymptotic complexity of the problem).

In external memory (see Section 2 for the definition of the model and its parameters), the I/O-complexity of CC and MSF is bounded from below by $\Omega(E/V \cdot \text{Sort}(V))$ [18] and a number of algorithms come within at most a logarithmic factor of $O(\text{Sort}(E))$. No deterministic algorithm is known to match the lower bound, but a randomized algorithm with $O(\text{Sort}(E))$[1] expected cost exists [15, 8]. Unlike in internal memory, the known external memory CC algorithms are essentially the same as the known algorithms for MSF, either exactly or as close variants. The largest discrepancy between the two settings is for the randomized $O(\text{Sort}(E))$ algorithm, where a fairly involved subroutine in its MSF variant becomes straight-forward for CC.

It seems that the randomized $O(\text{Sort}(E))$ external memory algorithm was never empirically investigated. One aim of this paper is to carry out such an investigation in the CC setting where the discrepancy mentioned above gives the algorithm the largest opportunity of being competitive in practice. Due to the large size of internal memory in most current computer systems, it is not clear whether a small asymptotic advantage of at most a logarithmic factor will materialize in practice for graphs of very large, but still plausible, sizes. In more detail, we want to investigate implementations and tuning options for the randomized $O(\text{Sort}(E))$ CC algorithm, as well as for the practically most promising of the remaining (asymptotically slightly worse, but often simpler) external CC algorithms, and then compare the best implementations of each algorithm on a broad range of graph classes. More generally, the aim of this paper is to investigate the best algorithmic choices for solving the CC problem in external memory.

**Previous work.** In the semi-external case, where $V \leq M$, scanning the edges and maintaining the components via a Union-Find data structure in internal memory will solve CC in $O(\text{Scan}(E))$ I/Os. The classic Borůvka MSF algorithm was externalized by Chiang et al. [8] by showing how to implement a Borůvka step in $O(\text{Sort}(E))$ I/Os, leading to $O(\log(V/M) \cdot \text{Sort}(E))$ I/Os for the entire algorithm. A simpler method for implementing a Borůvka step in $O(\text{Sort}(E))$ I/Os was later given by Arge et al. [4]. Munagala and Ranade [18] gave a CC algorithm using $O(\log\log(VB/E) \cdot \text{Sort}(E))$ I/Os and also proved the above-mentioned lower bound. The algorithm was generalized to MSF by Arge et al. [4], keeping the I/O bound. The algorithm of [4] was further developed by Bhushan and Gopalan [7], slightly improving the I/O bound.

[1] Using sparsification, the algorithm can be implemented to use $O(E/V \cdot \text{Sort}(V))$ I/Os [8], matching the lower bound exactly. In this paper, we will consider its $O(\text{Sort}(E))$ version as the two bounds are very close and in practice their difference is unlikely to outweigh the added algorithmic complication.

Karger, Klein, and Tarjan [15] gave an internal MSF algorithm with expected $O(E)$ running time using a linear time MSF verification algorithm as its central subroutine. The algorithm can be externalized to use expected $O(\text{Sort}(E))$ I/Os [8] by using external Borůvka steps and the external MSF verification algorithm by Chiang et al. [8]. For CC, it is an easy observation (already made by [2]) that the MSF verification can be substituted by a contraction step, which simplifies the implementation considerably. To the best of our knowledge, neither the CC nor the MSF variant of this external memory algorithm has been studied empirically.

A very simple randomized MSF algorithm using expected $O(\log(V/M) \cdot \text{Sort}(E))$ I/Os was developed by Sibeyn and Meyer. It was first reported by Schultes [24], and further described and empirically tested by Dementiev et al. [26]. A CC variant was theoretically and empirically studied by Sibeyn [27] (and to a lesser extent by Schultes [25]). Due to its simplicity, the algorithm is likely to have very competitive constants in its I/O bound, which is argued theoretically in [27] and substantiated by the experiments in [24, 25, 26, 27]; however, none of these experiments include comparisons to other external memory algorithms.

**Our contribution.** We implement the CC version of the $O(\text{Sort}(E))$ randomized and external algorithm by Karger, Klein, and Tarjan [15] and develop and investigate a number of tuning options. We then compare it to tuned versions of what we consider the practically most promising other algorithms for the CC, namely external Borůvka and the algorithm by Sibeyn et al. [24, 25, 26, 27]. Our experiments are executed on numerous graph classes, including $\mathcal{G}(n, p)$ graphs, grids, geometric graphs, and hyperbolic graphs (see Section 6).

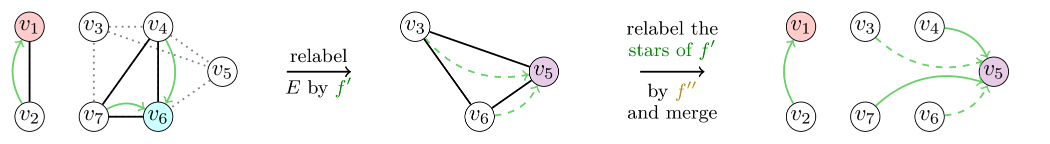
Among our findings are: Sibeyn's algorithm is a very strong contender due to its simplicity and due to an added degree of freedom in its internal workings when used in the CC setting. With the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive. Higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn, as does larger graph sizes. The latter observation is in line with its better (expected) asymptotic I/O bound. Borůvka's algorithm is not competitive compared to its contenders.

## 2 Definitions

The Connected Components (CC) problem on an undirected graph $G = (V, E)$ is to partition $V$ such that two nodes are in the same subset iff they are connected by a path in $G$. We overload the symbols $V$ and $E$: depending on the context, $V$ may represent either the *set* or the *number* of nodes, and $E$ may similarly represent either the set or the number of edges.

We analyze the cost of algorithms in the I/O-model of Aggarwal and Vitter [1] where $M$ denotes the size of the internal memory, $B$ denotes the block size, and $\text{Scan}(N) = \Theta(N/B)$ and $\text{Sort}(N) = \Theta(N/B \log_{M/B}(N/M))$ denote the costs of scanning and sorting $N$ elements.

As input, we assume the standard external memory representation of a graph as a list of its edges. This means that isolated nodes cannot be represented and should be handled separately by the user, which is straight-forward as they constitute their own connected components. We denote by $V(E)$ the set of nodes contained in an edge set $E$. Hence, an input is formally a graph $G = (V(E), E)$, but for simplicity we denote it just by $E$. We require the input $E$ to be given in lexicographical order, as all our algorithms need this. We thereby avoid an initial sorting step in the algorithms, which would only make their relative differences in running times less clear. Unless otherwise stated, we also assume that each unordered edge $\{u, v\}$ is stored only once in its *normalized* form $(\min(u, v), \max(u, v))$.



| $v$ | $v_1$ | $v_2$ | $v_4$ | $v_6$ | $v_7$ |
|---|---|---|---|---|---|
| $f'(v)$ | $v_1$ | $v_1$ | $v_6$ | $v_6$ | $v_6$ |

**(a)** Obtain $f'$ by solving CC on $E'$

| $v$ | $v_3$ | $v_5$ | $v_6$ |
|---|---|---|---|
| $f''(v)$ | $v_5$ | $v_5$ | $v_5$ |

**(b)** Obtain $f''$ by solving CC on $E$ relabeled by $f'$

| $v$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ |
|---|---|---|---|---|---|---|---|
| $f(v)$ | $v_1$ | $v_1$ | $v_5$ | $v_5$ | $v_5$ | $v_5$ | $v_5$ |

**(c)** Obtain $f$ by merging the star of $f''$ and the star of $f'$ relabeled with $f''$

**Figure 1** Relabeling and contraction. The input $E$ and the subset $E' \subseteq E$ are illustrated in **(a)** where $E'$ corresponds to solid black edges and $E - E'$ to dotted lines. Solving CC on $E'$ yields $f'$ which represents the two CCs $\{v_1, v_2\}$ and $\{v_4, v_6, v_7\}$ by $v_1$ and $v_6$, respectively. This corresponds to the two stars indicated by the directed green edges. Since $v_3$ and $v_5$ are not covered by $E'$, they are also not included in $f'$. The result of the contraction $E/E'$ is shown in **(b)** with solid lines and is obtained by relabeling $E$ by $f'$. Solving CC on $E/E'$ yields $f''$ indicated by the dashed directed edges. In **(c)**, we merge the stars of $f'$ relabeled with $f''$ (solid) together with the stars of $f''$ (dashed) and obtain the final result $f$. Observe that the star of $f$ may contain edges (e.g. $(v_7, v_5)$) that were not part of the original input $E$.

As output, we require a mapping $f\colon V(E) \to V(E)$ where $f(v) = f(u)$ iff $u$ and $v$ are in the same connected component. In other words, for each connected component one node is chosen as its representative. Concretely, the mapping shall be returned as the list of pairs $\{(v, f(v)) \mid v \in V(E)\}$, except that all identities $(v, v)$ are omitted. Note that we can interpret this output as the edge list of a directed graph composed of disjoint *stars*, where a star is a set of nodes pointing to a common center node. Each star represents a connected component in $E$.

A *relabeling* of a graph $E$ by a mapping $f\colon V(E) \to V(E)$ means applying $f$ to all edge endpoints and then removing parallel edges and self-loops in the resulting edge list. If $f$ is given by a graph of oriented stars as described above, a relabeling can be implemented in $O(\mathrm{Sort}(E))$ I/Os by $O(1)$ sorting and scanning steps on $E$. A *contraction* $E/E'$ of a graph $E$ by a subset $E' \subseteq E$ of its edges means solving CC on $E'$ and then relabeling $E$ by the returned mapping $f'$. The concepts of relabeling and contraction are illustrated in Figure 1 **(a)** and **(b)**, respectively.

Note that if we next obtain a mapping $f''$ by solving CC on the contracted graph $E/E'$, we can solve CC on the original graph $E$ as follows: use the mapping $f''$ to relabel the graph of stars representing $f'$ (only the target of each star edge is affected by the relabeling) and then return the union of those relabeled edges and the edges of the graph of stars representing $f''$. The process is illustrated in Figure 1 **(c)**. It is easy to verify that it will produce a graph of stars representing the solution $f$ to CC on the original graph $E$. All recursive algorithms in the current paper use this process as their framework.

## 3 Algorithms

In this section, we describe the basic versions of the implemented algorithms.

**Union-Find.** In the semi-external case, where $V(E) \leq M$, scanning the edges once while maintaining a Union-Find data structure on $V(E)$ in internal memory solves CC in $O(\mathrm{Scan}(E))$ I/Os and $O(E\alpha(E, V(E)))$ time [28], where $\alpha$ is the inverse Ackermann function. We use this as a base case.

**Borůvka.** A Borůvka step in the MSF setting means letting each node choose an incident edge of minimum weight and then contracting the graph by the set $E'$ of chosen edges. In $E'$, each node is in a connected component of size at least two, so the number of nodes is at least halved in the step. As a Borůvka step requires $O(\text{Sort}(E))$ I/Os (see below), this leads to a recursive algorithm which will use $O(\log_2(V/M)\text{Sort}(E))$ I/Os before the semi-external base case is reached. This constitutes Borůvka's algorithm.

The first part of a Borůvka step finds $E'$ with $O(\text{Sort}(E))$ I/Os as follows: double $E$ during a scan to make it contain both directions of each undirected edge. Then for all nodes choose an incident edge of minimum weight via a single sort and scan of this version of $E$.

To implement the remainder of a Borůvka step, one can exploit that $E'$ is a graph where each connected component has exactly one cycle, as seen by repeatedly following paths of chosen edges until all nodes have been visited. Assuming that all edge weights are unique (otherwise, use node IDs as tie-breakers), the weights along any such path are strictly decreasing, except when traversing the lightest undirected edge $\{u, v\}$ of the component in two directions $(u, v), (v, u)$, implying that the cycle is a two-cycle. Both directions have the same normalized representation, hence can be identified and de-duplicated by sorting $E'$, after which the connected component corresponds to a tree rooted in $v$. This can be done for all such pairs in the same sorting step, making the edges $E'$ form a forest where each tree coincides with a connected component. We select the roots as the components' IDs.

In order to return the star graph of the mapping $f'$, we have to inform each node of its tree's root. Early external methods [8, 2] used algorithms for Euler tours of trees based on list ranking. We use a simpler method described in [4]. It requires $O(\text{Sort}(V(E)))$ I/Os and is based on the fact that edge weights are strictly increasing on root-to-leaf paths in the trees, i.e., if we address messages to nodes by the weight of their incoming edge, parents will be processed before their children. This allows edge weights to be used as a "time line" in a general technique known as time-forward processing [17]. The propagation is done for all trees simultaneously by maintaining a set of signals in an external priority queue. The data structure is initialized by inserting signals for all children of all roots. Using sorting steps, we also create a list $L$ of tree edges not incident to a root. In $L$, all child edges of a node $v$ are grouped together, and the order between groups is determined by the weight of the parent edge of $v$. We then repeatedly remove the signal with smallest key from the priority queue and forward the information contained to the next block of children from $L$.

In the CC setting, the above algorithm for a Borůvka step can be implemented by (formally) assigning to all edges their unique normalized identity as their weight. Note that in the first part of the step, this is equivalent to each node simply choosing the edge to the neighbor with the lowest ID.

**Karger-Klein-Tarjan.** The CC version [2] of the $O(\text{Sort}(E))$ randomized, external algorithm based on Karger, Klein, and Tarjan [15] has the following recursive structure:

1. Perform three Borůvka steps on the input graph. Let the result be $E$.
2. Let $E'$ contain each edge of $E$ independently with probability $1/2$.
3. Compute the connected components of $E'$ recursively.
4. Form the contraction $E'' = E/E'$.
5. Compute the connected components of $E''$ recursively.
6. Relabel the result of step 3 by the result of step 5 and merge with the result of step 5, as detailed in Section 2.
7. Perform the relabelings and merges corresponding to the contraction in each of the initial Borůvka steps (as detailed in Section 2) and return the result.

In step 4, only the edges in $E - E'$ need to be processed as contraction by $E'$ eliminates all edges in $E'$. The crux of the Karger-Klein-Tarjan algorithm is that the number of edges in $E''$ is $O(V(E))$ in expectation. The argument for this is as follows (adapted from [15] to the CC setting).

Consider building a spanning tree $F$ for $E'$ by the standard Union-Find based algorithm *while* performing the sampling. That is, consider each edge $e$ of $E$ sequentially and include it in $F$ *iff* it is sampled *and* it does not form a cycle with edges already in $F$. Case 1: $e$ forms a cycle. Then $e$ will not appear in $E''$ due to the contraction. Case 2a: $e$ does not form a cycle, and is sampled. Then $e$ will not appear in $E''$ due to the contraction (as it is included in $F$). Case 2b: $e$ does not form a cycle, and is not sampled. Then $e$ may appear in $E''$. Since the final $F$ is a spanning tree of $E'$, we have $|F| \leq V(E') - 1$ and hence $|F| < V(E)$. Thus, the number of Case 2b edges is a stochastic variable upper-bounded by a negative binomial distribution with $p = 1/2$ and $r = V(E)$ (the number of tails before $V(E)$ heads have appeared when flipping a fair coin). Therefore the expected number of Case 2b edges is at most $V(E)$, implying the same for the expected number of edges in $E''$.

This statement is analogous to Lemma 2.1 of [15] for the MSF version. The rest of the argument in [15] for the expected cost carries over[2] almost verbatim, with $O(E)$ time substituted by $O(\text{Sort}(E))$ I/Os.

**Sibeyn.** The MSF algorithm presented in [26] is a surprisingly simple I/O-efficient algorithm. It works by repeatedly letting some node select its minimum incident edge and contracting that edge. These contractions are done in a lazy fashion using the time-forward processing method with node IDs as the “time” dimension. The original algorithm is described in two versions: one using buckets and the other using a priority queue.

We here describe the version based on priority queues. The algorithm represents the undirected edges only in their normalized form (oriented from lower to higher ID). All edges are initially inserted into a priority queue (PQ) which is ordered by source first and edge weight second. This ordering allows the algorithm to perform node contractions by repeatedly extracting the minimum edge in the PQ. When the extracted edge $(u, v, w)$ has a new source $u$ compared to the previous extracted edge, $\{u, v\}$ is the lightest edge incident to $u$ (after the contractions done so far) and is output as an MSF edge. The edge $\{u, v\}$ is then contracted and $u$'s remaining edges are forwarded to (i.e., taken over by) $v$. In detail, all subsequent edges $(u, v', w')$ with source $u$ extracted from the PQ become $\{v, v'\}$ by inserting $(\min\{v, v'\}, \max\{v, v'\}, w')$ into the PQ, except that edges with $v' = v$ (i.e., self-loops) are skipped. In this MSF version of the algorithm, forwarded edges need to be annotated with the original node IDs of their endpoints, in order for the output to be a correct MSF. When the number $V'$ of source IDs remaining in the PQ can fit in internal memory, i.e., when $V' \leq M$, the rest of the edges in the PQ are extracted and a semi-external version of Kruskal's algorithm is run on them. If using randomized node IDs, the algorithm requires expected $O(E \log(V/V'))$ priority queue operations to contract the original node set $V$ to a smaller node set $V'$ (i.e. for contracting $V - V'$ nodes) [26]. This implies a total cost of $O(\log(V/M) \cdot \text{Sort}(E))$ I/Os.

In our setting, the goal is to compute connected components. This allows the algorithm to be simplified in a number of ways (some described in [25]). The tree that the algorithm outputs should only capture connectivity, hence its edges need not be edges from the original

[2] The argument in [15] allows for using only two initial Borůvka steps. We here follow the description of the CC algorithm in [2], which uses three.

input $E$, so there is no need to annotate forwarded edges with original node IDs. Additionally, one can choose an *arbitrary* edge out of the "current" source $u$ as the new target to forward edges to. A natural heuristic is to send the information as far forward in time as possible. This is achieved by simply ordering the PQ by source in increasing order and by target in decreasing order as the first edge out of each new source will then go to the furthest neighbor (or known reachable node due to forwarded edges) immediately.

As the final CC information should be represented as a set of stars, some post-processing has to be done on the rooted trees output by the modified node contraction algorithm. As node IDs give a topological ordering of the tree edges, one can simply reverse the tree edges and use time-forward processing in the opposite direction relative to the node contraction phase. This post-processing only incurs $O(\mathrm{Sort}(V))$ additional I/Os.

The bucket version of the algorithm replaces the priority queue with a set of unsorted buckets. Two variants are described in the CC setting in [27, Section 3.4]: one which processes each bucket in internal memory and one which uses the semi-external Union-Find algorithm on each bucket. Choosing bucket sizes ahead of time for the former variant is non-trivial as the density tends to increase during computation. We therefore focus on the latter variant in this paper.

**Randomized Borůvka.** A standard Borůvka step has a first part where each node selects an incident edge, and a second part where the connected components of this edge set $E'$ are found via time-forward processing and returned as a mapping represented by a star graph.

We now describe a novel randomized method for the second part which is simpler than time-forward processing, at the cost of a worse bound on the contraction factor. In Section 7, we empirically investigate whether this trade-off is beneficial for the overall I/O cost when using Borůvka steps (as part of Borůvka's or Karger-Klein-Tarjan's algorithm).

We consider the selected edge of a node as an outgoing oriented edge. The method is simple: 1) Let each node keep its selected edge with probability $p$, resulting in the edge set $E''$. 2) Mark all edges $(u, v)$ in $E''$ for which $E''$ contains an edge $(w, u)$, then remove all marked edges to give the final edge set $E'''$. Step 1) can be done during the edge selection process at no cost, and step 2) can be done in one additional sort and scan step. No (oriented) path in $E'''$ has length more than one, hence $E'''$ is a star graph itself (it represents its own connected components) and can just be returned. Note that while the star-graph computation discussed for the original Borůvka algorithm requires the cycle of a connected component to be a two-cycle, and therefore requires nodes to choose minimum incident edges according to some assigned unique edge weights, this is not the case for our randomized variant.

▶ **Lemma 1.** *$E'''$ has expected size of at least $p(1-p)V(E)$.*

**Proof.** $E'$ has size $V(E)$, so the expected size of $E''$ is $pV(E)$. If we for each edge $(w, u)$ in $E'$ count a mark whenever $(w, u)$ was kept *and* $(u, v)$ was kept (where $(u, v)$ is $u$'s chosen edge), then we have an upper bound on the total number of marks (it is an upper bound, as $(u, v)$ could also be counted as marked via another edge $(w', u)$, but $(u, v)$ can only hold one mark). Hence, the expected number of edges removed from $E''$ to $E'''$ is less than $p^2V(E)$. Thus, the expected size of $E'''$ is at least $p(1-p)V(E)$, which is maximized for $p = 1/2$. ◀

When contracting using the star graph $E'''$, each edge of $E'''$ will remove at least one node, so at least $1/2(1-1/2)V(E) = V(E)/4$ nodes are removed in expectation. Thus, the expected contraction factor is at least $1/(1-1/4) = 4/3$. The contraction for a given graph may be larger than this (just as for standard Borůvka steps and its lower bound of two on the contraction factor). In Section 7, we empirically study contraction factors.

## 4 Tuning Options

We suggest and experimentally evaluate several variations of the algorithms with potential for impact on their practical running times and I/O costs.

**Pipelining.** Pipelining is the concept of one algorithmic sub-routine handing its output directly to another sub-routine without storing the intermediate data on disk. Applying this where possible can save I/Os, and our implementation platform STXXL offers tools for this type of programming. Before settling on using it, however, we want to investigate its impact.

**Contraction sub-routine.** In Borůvka's algorithm, and in the first step of the Karger-Klein-Tarjan algorithm, nodes are contracted. We investigate if time-forward processing based Borůvka steps or the proposed randomized version will be the fastest. The general form of the I/O cost argument in [26, 24] states that if Sibeyn's algorithm is run until the number of nodes has been contracted from $V$ to $V'$, it uses expected $O(\log(V/V') \cdot \mathrm{Sort}(E))$ I/Os. Thus, another possible contraction sub-routine in Karger-Klein-Tarjan is to use Sibeyn.

**Omitting node contractions at the root in Karger-Klein-Tarjan.** From the details of the cost analysis of Karger-Klein-Tarjan [15], it seems likely that the initial contraction in the root node of its recursion tree will dominate the running time in practice. The asymptotic result of expected $O(\mathrm{Sort}(E))$ cost still holds if this contraction (but not the contractions in other nodes of the recursion tree) is omitted. Then the algorithm will simply start with a scan of the input edge list when sampling edges before the first recursive call. If the returned mapping happens to contract nodes and edges well, the second recursive call will not contribute much to the total I/O cost, either. In this case, the dominating part will be the contraction after the first recursive call, which comprises two sorting steps and two scannings steps on $E$ (if we enter the base case in the second recursive call, we can even save one of the sorting steps, because the edges do not need to be sorted before making the call).

**Sampling parameter in Karger-Klein-Tarjan.** The original sampling probability for edges before the first recursive call in Karger-Klein-Tarjan was set to $p = 1/2$, but other values are possible. Lowering $p$ makes the first recursive call cheaper, and for denser graphs, we may still have a good effect of the contraction before the second recursive call, because a sparser subset of edges may still span large portions of the connected components. If this turns out to be true, one could make $p$ depend on the density (lower $p$ when the density is higher).

**Approximate counting algorithms for size estimation.** In the recursive algorithms, there is a need to estimate $V$ in order to know when the semi-external base case can be entered. One idea is to use approximate counting algorithms [3, 5, 10] from the streaming community to determine an estimate on the number of unique nodes in the edge list. In the streaming model this problem is referred to as the *Distinct Elements* problem and most solutions only provide a $(\delta, \varepsilon)$ guarantee, meaning that the estimate is within a $(1 + \varepsilon)$-multiplicative error with probability at least $(1 - \delta)$. As smaller values of $\varepsilon$ and $\delta$ require more internal work (mostly in the form of more evaluations of independent hash-functions), we investigate if we can benefit from these methods while staying I/O-bound.

**Which neighbor to contract in Sibeyn.** In each step of Sibeyn, the MSF version of the algorithm must choose to contract the current node and its neighbor given by its incident edge of minimum weight. In the CC version, it is free to choose any neighbor. As argued in [27], it may be beneficial to choose the neighbor with largest node ID. We investigate what is the best choice and the gains possible, and we empirically compare choosing a neighbor with largest node ID, a neighbor with smallest node ID, and a random neighbor (which corresponds to the MSF version).

**Minimizing the PQ in Sibeyn.** When running the PQ version of Sibeyn, we may exploit that the input edges are sorted. This allows us to skip the initial insertion of all edges into the PQ: while running the algorithm, the list of original edges can just be merged with the output of the PQ, which then only needs to contain reinserted edges, not original edges.

**Influence of relinking in Sibeyn with buckets.** In the bucket version of Sibeyn's algorithm, the connected components for a bucket are computed and signals are sent to later buckets. Sibeyn [27] introduces a *relinking* variant which restructures the signals before sending them to reduce the number of signals between buckets.

## 5 Implementation

All algorithms are implemented in C++ using the STXXL library [9], which offers highly tuned external memory versions of fundamental algorithmic building blocks like sorting and priority queues. It also supports pipelining, as described in Section 4. The external priority queue of STXXL, which we use in several places in the algorithms implemented, is based on [23].

In order to accommodate different contraction schemes in the contraction of the recursive Karger-Klein-Tarjan algorithm, we implemented a generic framework for performing the sampling, contraction, relabeling and merging during the algorithm's execution. The supported contraction schemes are Sibeyn, Karger-Klein-Tarjan and randomized Borůvka contractions. The framework comes in two flavors: a purely vector-based and a pipelined stream-based implementation. This allows us to evaluate to what degree pipelining is beneficial.

**Edge representation.** In our implementation we store each undirected edge by its ordered pair $(u, v)$ where $u < v$. For sorted edges we additionally employ a more I/O-efficient data structure: consecutive edges with the same source $u$ are compressed to a single entry $u$ followed by all its adjacent nodes and a delimiter.

**Data structures.** The pipelined implementations make use of several STXXL data structures. In these, generated data is not saved in an explicit vector but fed to a container which then functions as a data stream with read-only access. An example of this is STXXL's *sorter*: in the first phase, items are pushed into the write-only sorter in an arbitrary order by some algorithm. After an explicit switch, the filled data structure becomes read-only and the elements are provided as a sorted stream which can be rewound at any time. While a sorter is functionally equivalent to filling, sorting and reading back an external memory vector, the restricted access model reduces constant factors in the running time and I/O cost [6].

**Semi-external base case.** While we assume that the number of nodes in the original input is known exactly, this is not necessarily true for recursive calls. As we aim to switch to a semi-external base case algorithm, keeping track of the number of remaining nodes is essential. For the Karger-Klein-Tarjan algorithm, the node contraction step contracts a specified number of nodes and as such, a good estimate for the number of nodes remaining after initial contraction is simply the original node count minus the number contracted[3]. The same holds for the contracted edge set passed on to the second recursive call: the number of connected components returned from the first recursive call is known and corresponds to the number additional nodes contracted. This leaves the first recursive call operating on the sampled edges $E'$. The number of nodes here is trivially bounded both by the bound known before sampling and by $2|E'|$. The latter bound can be improved somewhat; as edges are kept sorted, the number of unique sources can be counted while sampling and only $|E'|$ is added for an upper bound. Taking the best of these bounds at different stages, we maintain an upper bound on the node count. By using these upper bounds we can save I/Os in the relabeling as relabeled edges may immediately be piped into the semi-external base case without the otherwise required final sorting step. Note that while computing the exact number of nodes requires only a few scanning and sorting steps, this is too costly in practice for competitive results.

## 6 Graph classes

For our experiments, we use a variety of different synthetic graph models. We consider four types: the Gilbert type classic random graphs, random geometric and random hyperbolic graphs, both belonging to the class of spatial network models, and finally deterministically generated grid graphs. Using scalable graph generators, we generate fully external ($M < V$) graphs with a range of different parameters. For a recent overview of such generators, see [21].

**Gilbert graphs.** In the $\mathcal{G}(n, p)$ model of Gilbert [12], each edge is present independently with probability $p$. The $\mathcal{G}(n, p)$ model can generate graphs with a varying number of connected components for sufficiently small $p$. It is widely used in empirical work, but its degree distribution is often considered atypical compared to real-world instances.

**Random Geometric graphs.** Random Geometric graphs (RGGs) [13, 19] are a simple case of spatial networks where graphs are projected onto Euclidean space. In RGGs $n$ points are placed uniformly at random into a $d$-dimensional unit-cube $[0, 1)^d$ where any two points are connected if their Euclidian distance is below a given threshold $r$. To generate graphs in this model, we use the generator available in KaGen [11].

**Random Hyperbolic graphs.** Random Hyperbolic graphs (RHGs) [16, 14] are a special case of spatial networks where graphs are projected onto hyperbolic space. We describe the threshold model, the simplest RHG variant [14]. The points are randomly placed onto a two-dimensional disk in hyperbolic space where the radial probability density function increases exponentially towards the border. The angular coordinate is sampled uniformly at random from $[0, 2\pi)$ and points are connected if their hyperbolic distance is less than a given threshold $R$. The density of points near the center is controllable by setting a dispersion parameter $\alpha$. One interesting feature of RHGs is that the node degrees follow a power law distribution which is often found in real-world graph instances, in particular when generated via human activities and choices. In the threshold model the exponent is $\gamma = 1 + 2\alpha$ with high probability [14]. To generate graphs in this model, we use the HyperGen generator [20].

[3] This gives an exact count except when a connected component is contracted to a singleton – at which point it will not appear in the edge list.

**Grid graphs.** We consider two different types of square grid graphs. In both versions, the nodes are seen as points in a two-dimensional grid; $(x, y)$ for $1 \le x \le w$ and $1 \le y \le h$. For the simpler version, nodes are connected horizontally and vertically to their neighbors. All nodes except for boundary nodes thus have degree 4. To achieve higher degree, we additionally consider generalized grids in which nodes are connected to all nodes within distance $d$ under the infinity norm. That is, node $(x, y)$ has edges to nodes $(x + i, y + j)$ where $-d \le i \le d$ and $-d \le j \le d$, except where this exceeds the grid boundary. Internal nodes in these graphs have degree $4d(d + 1)$. To investigate the effects of increasing the number of components, we additionally generate graphs which we refer to as *cubes* consisting of multiple disjoint layers, each of which is a generalized grid graph.

## 7 Experiments

Our experiments were carried out in two phases. In the first phase, we investigated the impact of the various algorithmic variants and proposals for tuning described in Sections 3–5. This was done on subsets of the test graphs of Section 6 and selected other test cases. The aim of this phase was to develop a set of well-engineered implementations of the most promising contenders. In the second phase, we then compared those on a large set of test graphs of Section 6 – the compute time of this phase alone comprised one third of a year. Below, we describe our experimental setup and our learnings from each of the two phases. For space reasons, we mainly include plots for the second phase. The full set of plots are in Appendix A (in the plots, the numbers $V$ and $E$ are denoted by $n$ and $m$, respectively).

### 7.1 Experimental setup

The experiments were run on individual nodes of the Goethe-HLR cluster at Goethe University Frankfurt, as this allowed us to run many experiments simultaneously (note that our algorithms all are sequential, parallel algorithms for CC are beyond the scope of this paper). The nodes each have Intel Xeon Skylake Gold 6148 CPUs and 192 GB of RAM. Each node has a HGST Ultrastar HUS726020ALA610 hard drive which was used for the STXXL disk file. The code was compiled using GCC version 8.3.1 with the optimization parameters `O3` and `march=native`.

In each run, the input graph was first loaded onto the local hard drive in the appropriate STXXL data structure: an edge stream for the stream-based implementations and an STXXL vector for the vector-based implementations. The threshold for switching to the semi-external base case was for all the algorithms set to 33,554,432 nodes which corresponds to 256 MiB of node IDs. To capture wall-time and I/O volume, we used the `iostats` module provided by the FOXXLL library (a component of STXXL). The main timing plots in Figures 10–17 show the wall time (bars) and total I/O volume (bytes read plus bytes written during the execution of the algorithm) reported by the `iostats`.

To keep the combined compute time of the experiments from becoming infeasible (even when executing experiments in parallel on a cluster), we reduced the RAM used by the CPUs to a few GB, which allowed us values of $V/M$ up to 80 and graph densities $E/V$ up to 20 (although not both maximal values at the same time) while keeping individual experiments under half a day of compute time. Our hypothesis was that if the algorithms are I/O-bound, the relative running times of the algorithms would stay approximately the same even if moving to larger sizes of RAM and from the hard disks of the cluster nodes to solid-state disks. With the set of final contenders, we conducted experiments on selected graph classes on a single machine having 16 GB of RAM and a RAID with six solid-state disks of 480 GB each. Those experiments confirmed our hypothesis, as the relative running times changed less than 20% in almost all cases tested.

To limit the amount of memory used on the cluster nodes, we limited the internal memory allowed for STXXL primitives used (sorting streams and priority queues were limited to 1 GB of RAM each). With the base case threshold (accounting for overhead), and the above limits, the implementations should be able to run with approximately 2 GB of memory. We did not have a mechanism to enforce a strict bound on the memory actually allocated, but monitored the amount of RAM actually used, which was in the range of 2 GB to 5 GB. To force disk accesses rather than additional buffering, the `direct` flag was used for the STXXL disk file.

## 7.2 Phase 1 – Initial Findings

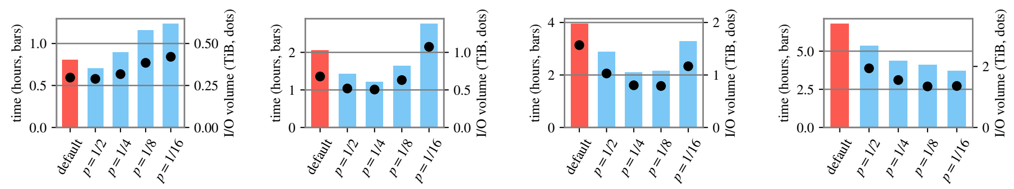
We now describe our main findings in phase one of our experiments. Unless otherwise mentioned, the measure compared is wall clock time.

### Randomized Borůvka and Borůvka

For our suggestion for randomized Borůvka steps, we first investigated the impact on the observed contraction ratio of a number of different edge representations and of various sampling parameters. On most graph classes, sampling parameters much closer to one than to 1/2 gave better contraction ratios (see Figure 5), in line with Lemma 1 only giving a lower bound. There was correlation among the graph classes between increased contraction efficiency of the randomized Borůvka steps and increased contraction efficiency of standard Borůvka (past the lower bound of two on the ratio). However, the ratio was consistently worse for the randomized version, and its simpler code did not make up for this when considering the total time of Borůvka's algorithm. Additionally, both of the two versions of Borůvka's algorithm were clearly worse than Sibeyn's algorithm based on PQs, both before and after adding pipelining. For instance, when doing node contraction until the base case is reached, we found that Sibeyn's algorithm was approximately 59% faster than ordinary Borůvka and we likewise found that one variant of our Karger-Klein-Tarjan implementation using Sibeyn's algorithm for node contraction was around 58% faster than one using the randomized Borůvka steps (results vary across graphs, numbers given here are averages). We therefore left Borůvka's algorithm out of the final race.

### Pipelining

Adding pipelining in STXXL turned out to improve our implementations of Sibeyn and of Karger-Klein-Tarjan. Introducing pipelining (including compressed edge streams) reduced the running time of one of our Karger-Klein-Tarjan variants approximately 73% on average over a simple version based on STXXL vectors. Even for our Sibeyn implementation (where even a simple implementation incurs much less copying), introducing pipelining improved the running time by approximately 10% on average. The tunings to the PQ based version of Sibeyn suggested in Section 4, e.g. reducing the processed volume of edges in the PQ, turned out to be beneficial, lowering running time by an additional approximately 29% on average.



**(a)** $m/n = 2.04$ **(b)** $m/n = 5.00$ **(c)** $m/n = 10.00$ **(d)** $m/n = 20.00$

**Figure 2** (Subset of Figure 10) Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 5GiB and varying density. The *default* variant always contracts and has a sampling probability of $p = 1/2$. The remaining variants skip contraction in the root and have fixed sampling probabilites. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.

## Approximate counting

For the estimation of $V$ using approximate counting algorithms, we tested the FM algorithm by Flajolet and Martin [10]. In essence, the FM algorithm computes for a given input stream an estimate of its number of distinct elements. For this, every input element is mapped by a hash function and incorporated into a later modified and returned proxy value. Due to the output variance being intolerably large, standard median-of-means techniques are employed which in turn, however, require more independent hash functions.

For several graph classes, we employed the FM algorithm in the sampling step of the Karger-Klein-Tarjan algorithm with an increasing number of hash functions. To accurately assess the returned estimates we separately ran the Karger-Klein-Tarjan algorithm with the same seed and explicitly counted the correct number of nodes in each sampling step.

We found that the number of hash functions needed in order to get a useful precision in the estimate was so high that it impacted the running time. Additionally, the errors in the estimation are two-sided, which fits badly with the fact that invoking a Union-Find based base case when not actually being semi-external will have disastrous effects on the running time. Combined, this made us decide not to include this method in the final experiments.

## Karger-Klein-Tarjan

For the contraction steps in the Karger-Klein-Tarjan algorithm we tried both standard and randomized Borůvka steps, as well as the PQ based version of Sibeyn, and the latter proved to be the better option.

When varying the sampling parameter $p$ in Karger-Klein-Tarjan, we observed a rather clear correlation (see Figure 2 and Figures 10–16): the best choice for both I/O volume and running time seems to be $p$ equal to the inverse density $V/E$ of the input graph, likely for the reasons conjectured in Section 4: a sampled subset of edges containing around $V$ nodes will often by itself contract the node set considerably, while the left recursion will be cheap if this is achieved for small $p$, which may happen more often for high densities.

Also, when visualizing the recursion trees, a clear pattern was a balanced tree for this value of $p$, whereas quite strongly left-leaning and right-leaning trees appeared for larger and smaller values, respectively. Profiling of the distribution of time spent in the nodes of the recursion trees showed the root to be dominating, which is aligned with the analysis in [15]. Often, the contraction step was dominating (as can also be seen in Figure 2). There was also a small tendency for Karger-Klein-Tarjan to improve relative to the other algorithms when $V$ grew compared to $M$ (for fixed density and graph class). These observations (which are

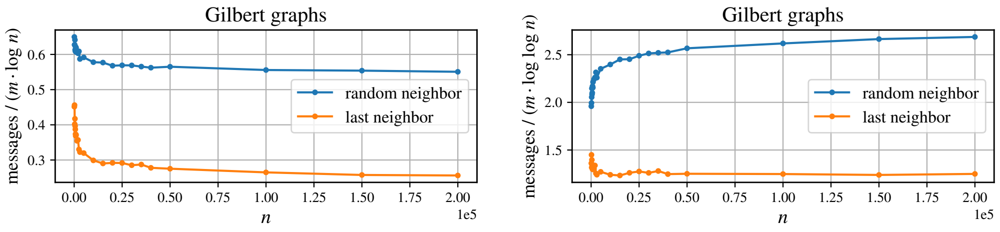


**Figure 3** (Copy of Figure 7) Number of forwarded messages divided by $m \log n$ (left) or $m \log \log n$ (right) for Gilbert graphs for increasing values of $n$. The value $p$ is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log \log n$ if messages are forwarded to the last neighbor.

visible in the plots in Figures 10–17) inspired us to implement variants of the Karger-Klein-Tarjan which do not use contraction at the root, and adaptive variants which in all recursion tree nodes choose contractions only for low (estimated) density and also choose a sampling parameter close to $V/E$.

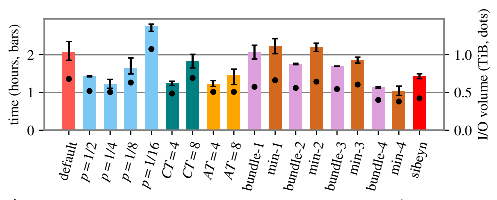
### Choice of contraction target in Sibeyn

Perhaps our most interesting observation in phase one was the influence on Sibeyn of the choice of which neighbor to contract (see Section 4). We tried the choices of nearest, random, and farthest in node-ID order (i.e., the "timeline" in the time-forward processing by the PQ). Of these, the random choice intuitively can be expected to behave like the MSF variant of the algorithm (where each node must choose the neighbor of its lightest incident edge, and where the node IDs are randomly permuted). As exemplified by the first plot of Figure 6, where a message is a PQ entry (i.e., an edge in its original form or a later replaced form) each inducing $O(1)$ PQ operations, the choice of nearest is by far the worst and was not considered again. On the other hand, farthest is always better than random (see rest of plots in Figure 6).

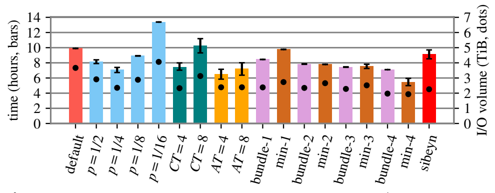
This effect was first studied in [27], where an expected bound of $O(E \log \log(V))$ messages was claimed (but the proof omitted) for Gilbert graphs. As seen in Figure 3, we here verify that claim empirically, and also demonstrate that it does not hold for the random choice. Even more interesting, for random grid graphs and random hyperbolic graphs, the empirical evidence even suggests a better bound of expected $O(E)$ (Figures 8 and 9). These findings suggest that Sibeyn in practice is strictly faster for CC than for MSF, and that it for the former may often run in cost $O(\mathrm{Sort}(E))$. Additionally, the bulk of the messages seem concentrated very late in the time-forward process, which in the external version is preempted by entering the semi-external Union-Find case, which in turn has lower overhead per edge/message than a PQ. Combined with the general simplicity of Sibeyn, these findings indicate that it may be very hard to surpass. For our final Sibeyn implementations, we naturally used the farthest neighbor choice.

## 7.3 Phase 2 – Final Algorithms

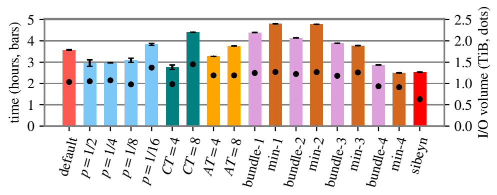
For the second and final phase of experiments we selected the following algorithms (with implementation choices fixed as described above).



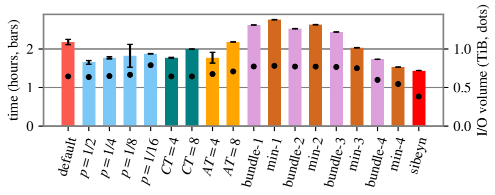
**(a)** Gilbert graph, smallest node set, $m/n = 5$



**(b)** Gilbert graph, largest node set, $m/n = 5$



**(c)** Random Geometric graph



**(d)** Random Hyperbolic graph

**Figure 4** (Copies of Figure 10b, Figure 13b, Figure 16d and Figure 17c)
Running times and I/O volumes for two Gilbert graphs and the two largest generated RGG and RGH instances.

- Karger-Klein-Tarjan in several versions: One with contractions in all recursion tree nodes and fixed sampling parameter $p = 1/2$. Four versions omitting contraction at the root of the recursion tree and having fixed sampling parameters of 1/2, 1/4, 1/8, and 1/16, respectively. Two adaptive versions which in each node of the recursion tree choose (among the values above) the sampling parameter closest to the estimated inverse density $V/E$ of the input graph of the node, and also omit contraction if the estimated density $E/V$ is below a fixed threshold of 4 or 8, respectively. Two similar adaptive versions where instead the threshold is 4 or 8, respectively, when the estimated $V$ is close to the base case, but tends to 2 for growing $V$. These nine algorithms are denoted *default*, $p = 1/2$, $p = 1/4$, $p = 1/8$, $p = 1/16$, $CT = 4$, $CT = 8$, $AT = 4$, and $AT = 8$, respectively.
- Sibeyn's algorithm based on buckets, using Union-Find for solving CC in buckets, as described in [27] (where buckets are called bundles). We tried four increasing bucket sizes, all without and with relinking to minimize edges straddling buckets (Section 4 and [27]). These eight algorithms are denoted *bundle-x* and *min-x* for $x = 1, 2, 3, 4$.
- The basic Sibeyn using a PQ. This algorithm is denoted *sibeyn*.

## Comparing Karger-Klein-Tarjan variants

We find based on Figures 10–17 that among the Karger-Klein-Tarjan variants the adaptive ones are either winning or performing close to the best variant. This can be observed for all considered graph classes (see Figure 4 for an overview). In almost all cases, fixed contraction thresholds tend to perform better than adaptive ones. Further, setting the threshold to a small value seems preferable. This behaviour is consistent when increasing the number of nodes while keeping the density fixed (see for instance Figure 10b, Figure 11b, Figure 12b and Figure 13b) where it is clear that relative performances remain unchanged.

The good performance of these adaptive variants and the comparatively generally weak performance of the *default* variant support our claim that contractions can be intolerably costly.

### Comparing Sibeyn variants

We find that both versions of Sibeyn's algorithm are strong contenders. While the PQ based Sibeyn algorithm generally performs better on low density graphs (see Figure 4c and Figure 4d), its relative performance gets worse with increasing $V$ (see Figure 4a, Figure 4b and Figures 10–13). Additionally, while the overall I/O volume may be near optimal (see Figure 4d), the achieved wall clock time does not always reflect this, indicating that the I/Os incurred by the PQ may be more costly than those for sorting.

In comparison, the bucket based Sibeyn algorithm performs consistently among the studied graph classes (see Figure 4 and Figures 10–17). We notice two clear trends, larger buckets generally perform better and adding relinking typically improves performance for graphs with higher densities.

## 8 Conclusion

The results of our experiments in phase two on the above set of algorithms can be seen in Figures 10–17. Sibeyn's algorithm is a strong contender. One reason is that it is very simple, using essentially only a priority queue (or repeated Union-Find in the bucket version). A tuned implementation of external priority queues can be highly efficient: our measurements on STXXL show that sorting by its priority queue is less than a factor of 2.5 slower than its sorting routine. Another reason is that for its CC variant, the choice of farthest neighbors seems to lower the number of messages generated to essentially linear (with the exact observed bound depending on the graph class) in $E$, which translates into a similar number of priority queue operations. Very few sorting and scanning steps on the input edge list can be performed by a competing algorithm before it will lose to Sibeyn.

Still, with the right tunings, the Karger-Klein-Tarjan algorithm can be implemented to be competitive in many cases. The best Karger-Klein-Tarjan variant often either wins over PQ based Sibeyn, but not bucket based Sibeyn, or vice versa. If nothing is known about the graph type and density, an adaptive variant such as $CT = 4$ may be a robust choice. In general, higher graph density seems to benefit Karger-Klein-Tarjan relative to Sibeyn. If choosing the bucket based Sibeyn variant, using the largest bucket size is clearly preferable (and often the min variant has a slight advantage). Borůvka's algorithm was not able to compete with neither Sibeyn nor Karger-Klein-Tarjan.

Natural future work suggested by this work include: 1) To investigate theoretically the observed positive effects on Sibeyn of the farthest neighbors choice. As demonstrated in Figures 7–9, different results seem plausible for different graph classes. 2) To compare empirically also the MSF versions of the algorithms.

## References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

2 Susanne Albers, Andreas Crauser, and Kurt Mehlhorn. Lecture notes on algorithms for very large data sets. https://web.archive.org/web/19970816002522/http://www.mpi-sb.mpg.de/~crauser/Plan.ps.gz, 1997.

3 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

4 Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004. doi:10.1016/j.jalgor.2004.04.001.

5 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In José D. P. Rolim and Salil P. Vadhan, editors, *Randomization and Approximation Techniques, 6th International Workshop, RANDOM 2002, Cambridge, MA, USA, September 13-15, 2002, Proceedings*, volume 2483 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berling Heidelberg, 2002. doi:10.1007/3-540-45726-7_1.

6 Andreas Beckmann, Roman Dementiev, and Johannes Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10. IEEE Computer Society, 2009. doi:10.1109/IPDPS.2009.5161001.

7 Alka Bhushan and Sajith Gopalan. An I/O efficient algorithm for minimum spanning trees. In Zaixin Lu, Donghyun Kim, Weili Wu, Wei Li, and Ding-Zhu Du, editors, *Combinatorial Optimization and Applications - 9th International Conference, COCOA 2015, Houston, TX, USA, December 18-20, 2015, Proceedings*, volume 9486 of *Lecture Notes in Computer Science*, pages 499–509. Springer, 2015. doi:10.1007/978-3-319-26626-8_36.

8 Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA*, pages 139–149. SIAM, Philadelphia, PA, USA, 1995. URL: http://dl.acm.org/doi/10.5555/313651.313681.

9 Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: standard template library for XXL data sets. *Softw. Pract. Exp.*, 38(6):589–637, 2008. doi:10.1002/spe.844.

10 Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985. doi:10.1016/0022-0000(85)90041-8.

11 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. doi:10.1016/j.jpdc.2019.03.011.

12 Edgar N. Gilbert. Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144, December 1959. doi:10.1214/aoms/1177706098.

13 Edgar N. Gilbert. Random plane networks. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):533–543, 1961. doi:10.1137/0109045.

14 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II, ICALP 2012, Warwick, UK, July 9-13, 2012*, page 573–585. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-31585-5_51.

15 David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, March 1995. doi:10.1145/201019.201022.

16 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.

17 Anil Maheshwari and Norbert Zeh. A survey of techniques for designing I/O-efficient algorithms. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer, 2002. doi:10.1007/3-540-36574-5_3.

18 Kamesh Munagala and Abhiram G. Ranade. I/O-complexity of graph algorithms. In Robert Endre Tarjan and Tandy J. Warnow, editors, *Proceedings of the 10th Annual ACM-SIAM*

*Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*, pages 687–694. ACM/SIAM, 1999. URL: `https://dl.acm.org/doi/10.5555/314500.314891`.

19 Mathew D. Penrose. *Random Geometric Graphs*. Oxford University Press, 2003. `doi:10.1093/acprof:oso/9780198506263.001.0001`.

20 Manuel Penschuck. Generating practical random hyperbolic graphs in near-linear time and with sub-linear memory. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, volume 75 of *LIPIcs*, pages 26:1–26:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.SEA.2017.26`.

21 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. `arXiv:2003.00736`.

22 Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. `doi:10.1145/505241.505243`.

23 Peter Sanders. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics*, 5:7, 2000. `doi:10.1145/351827.384249`.

24 Dominik Schultes. *External Memory Minimum Spanning Trees*. Bachelor thesis, Universität des Saarlandes, 2003. URL: `http://algo2.iti.kit.edu/schultes/emmst/emmst_short.pdf`.

25 Dominik Schultes. External memory spanning forests and connected components, 2003. URL: `http://algo2.iti.kit.edu/dementiev/files/cc.pdf`.

26 Jop Sibeyn, Roman Dementiev, Peter Sanders, and Dominik Schultes. Engineering an external memory minimum spanning tree algorithm. In Jean-Jacques Levy, John C. Mitchell, and Ernst W. Mayr, editors, *Exploring New Frontiers of Theoretical Informatics*, volume 155, pages 195–208. Kluwer Academic Publishers, Boston, 2004. `doi:10.1007/1-4020-8141-3_17`.

27 Jop F. Sibeyn. External connected components. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebæk, Denmark, July 8-10, 2004*, volume 3111, pages 468–479. Springer Berlin Heidelberg, 2004. `doi:10.1007/978-3-540-27810-8_40`.

28 Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. `doi:10.1145/321879.321884`.

## A Plots

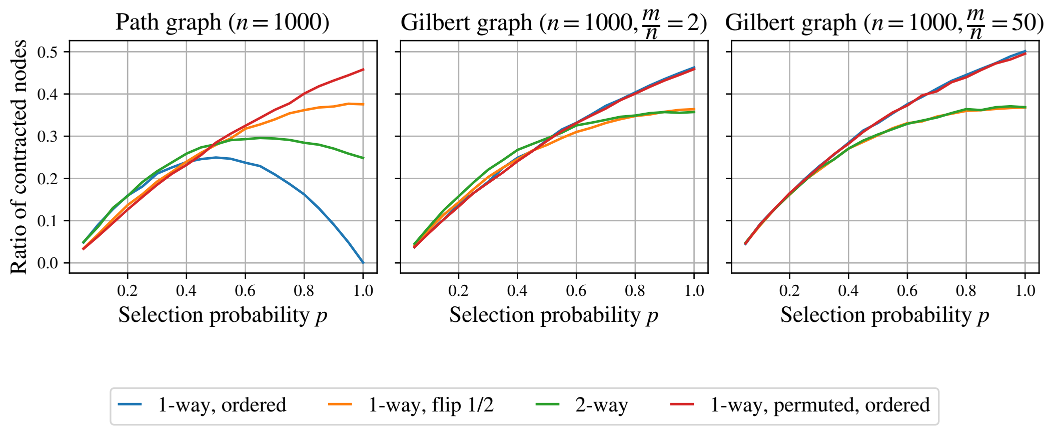


**Figure 5** Contraction ratios achieved by variants of a randomized Borůvka step for varying $p$ and varying edge representations. For Gilbert graphs, the contraction ratio increases with increasing selection probability $p$ where the best candidates are the ordered variants. For path graphs, the variants without randomness peak and start to perform worse.

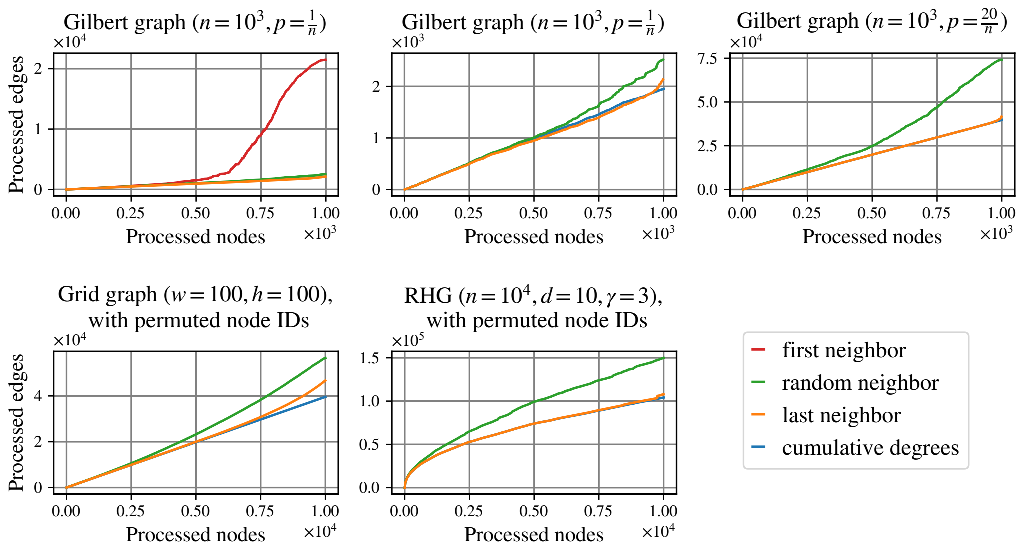


**Figure 6** Message volume of forwarded messages for different graphs depending on the contraction strategy. Sibeyn's algorithm processes significantly more edges (priority queue messages) when messages are sent to the first neighbor (see first plot). In comparison, sending messages to the last neighbor produces volumes very close to the baseline (cumulative degrees) and always performs better than sending to a random neighbor.

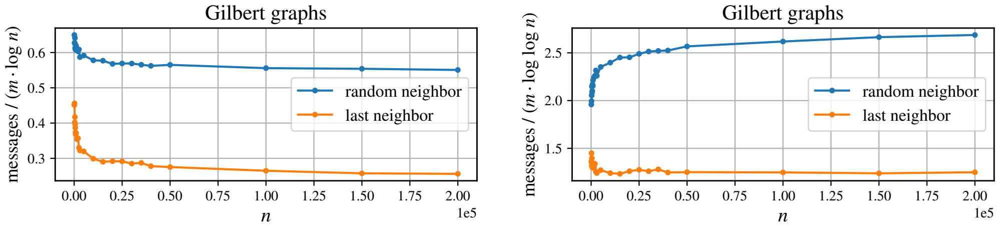


**Figure 7** Number of forwarded messages divided by $m \log n$ (left) or $m \log\log n$ (right) for Gilbert graphs for increasing values of $n$. The value $p$ is chosen s.t. a density of five is fixed. In (left) we observe that the total number of produced messages is dominated by $m \log n$ whereas in (right) we see that the volume asymptotically matches with $m \log\log n$ if messages are forwarded to the last neighbor.

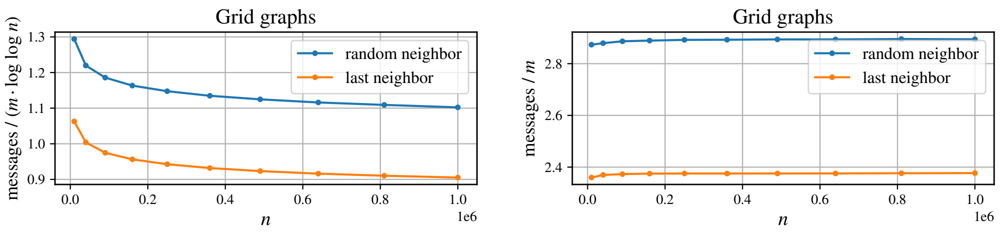


**Figure 8** Number of forwarded messages divided by $m \log\log n$ (left) or $m$ (right) for quadratic grid graphs for increasing values of $n$. By construction, these have density approximately two. In (left) we observe that the total number of produced messages is dominated by $m \log\log n$ whereas in (right) we see that the volume asymptotically matches with $m$.

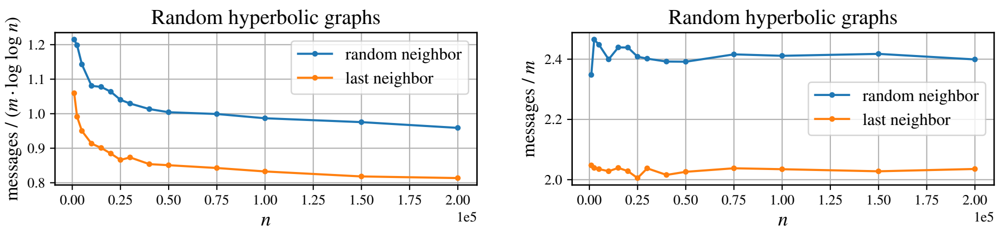
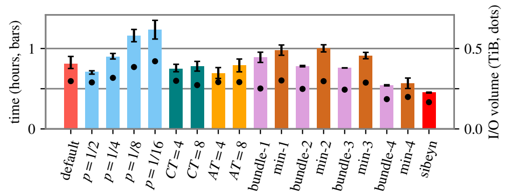
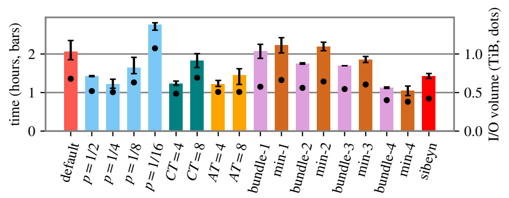


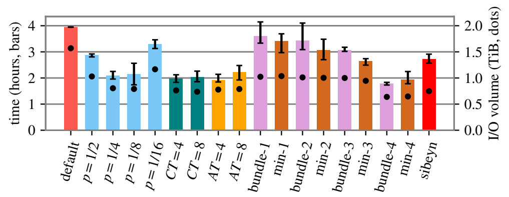
**Figure 9** Number of forwarded messages divided by $m \log\log n$ (left) or $m$ (right) for RHGs for increasing values of $n$. The degree parameter is set to 10 for all of these, yielding an approximate density of five. The degree exponent is set to 3. In (left) we observe that the total number of produced messages is dominated by $m \log\log n$ whereas in (right) we see that the volume asymptotically matches with $m$.



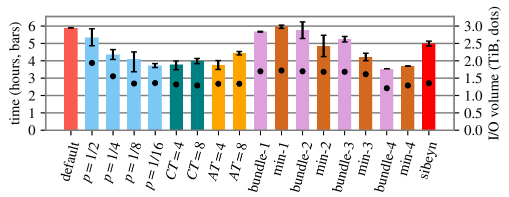
**(a)** $m = 1.34 \times 10^9$, $m/n = 2.04$



**(b)** $m = 3.36 \times 10^9$, $m/n = 5.00$



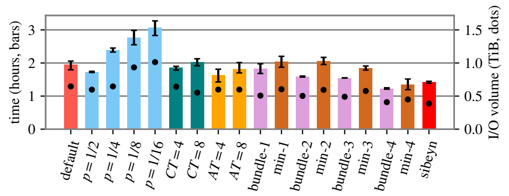
**(c)** $m = 6.71 \times 10^9$, $m/n = 10.00$



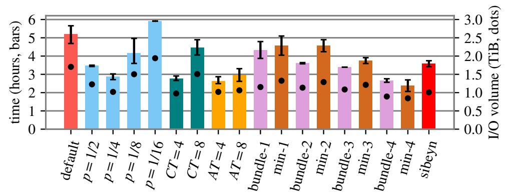
**(d)** $m = 1.34 \times 10^{10}$, $m/n = 20.00$

**Figure 10** Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 5GiB and varying density. For $m/n = 20$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point. The considered algorithms are in fixed order from left to right:

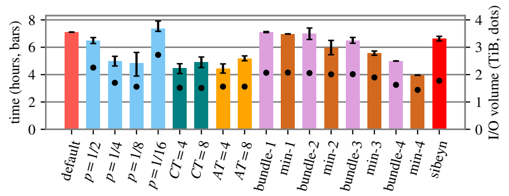
| | |
|---|---|
| *default*: | fixed sampling $p = 1/2$, always contract |
| $p = 1/x$: | fixed sampling $p = 1/x$, always contract except in root |
| $CT = x$: | adaptive sampling, contract if estimated density below fixed threshold $x$ |
| $AT = x$: | adaptive sampling, contract if estimated density below adaptive threshold $x$ . |
| *bundle-x*: | Sibeyn's algorithm based on buckets, without linking |
| *min-x*: | Sibeyn's algorithm based on buckets, with linking |
| *sibeyn*: | Sibeyn's algorithm based on priority-queues |



**(a)** $m = 2.68 \times 10^9$, $m/n = 2.04$

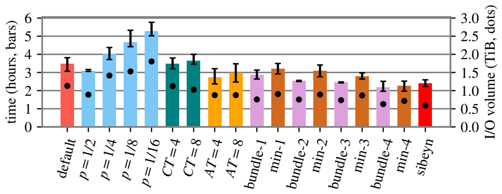


**(b)** $m = 6.71 \times 10^9$, $m/n = 5.00$

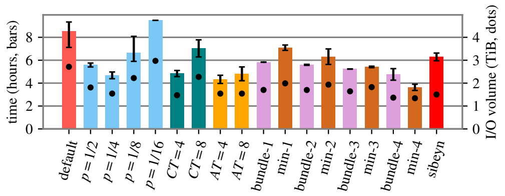


**(c)** $m = 1.34 \times 10^{10}$, $m/n = 10.00$

**Figure 11** Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 10GiB and varying density. For $m/n = 10$, the *default* variant exceeded the local hard disk's capacity leading to a halt in the algorithm's execution. We thus only report the elapsed wall time up until that point.

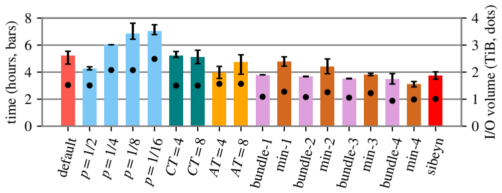


**(a)** $m = 4.03 \times 10^9$, $m/n = 2.04$

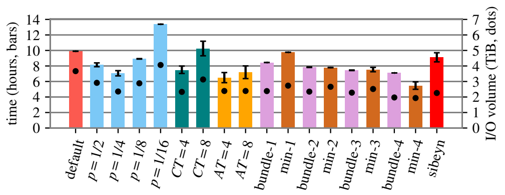


**(b)** $m = 1.01 \times 10^{10}$, $m/n = 5.00$

**Figure 12** Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 15GiB and varying density.

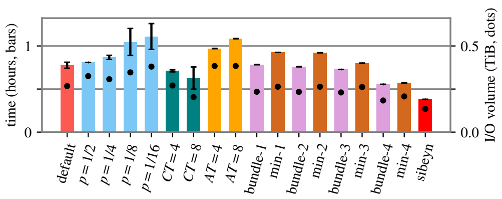


**(a)** $m = 5.37 \times 10^9$, $m/n = 2.04$

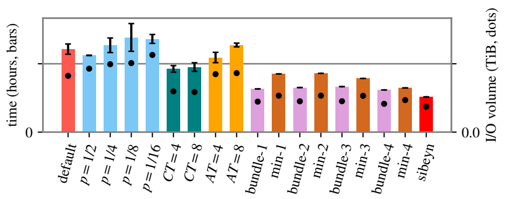


**(b)** $m = 1.34 \times 10^{10}$, $m/n = 5.00$

**Figure 13** Running times and I/O volumes for $\mathcal{G}(n, p)$ graphs with a node set size of 20GiB and varying density.

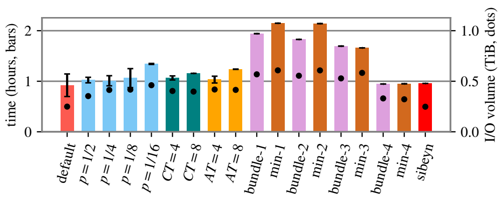


**(a)** $m = 1.34 \times 10^9$, $m/n = 2.00$

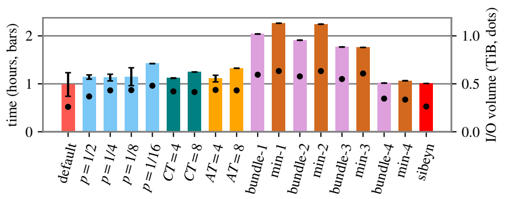


**(b)** $m = 1.34 \times 10^9$, $m/n = 1.00$

**Figure 14** Running times and I/O volumes for (a) a grid graph with $(w, h) = (25\,905, 25\,905)$ and (b) a path graph. For both instances the parameters were chosen to generate a 20GiB graph. Node IDs are permuted.

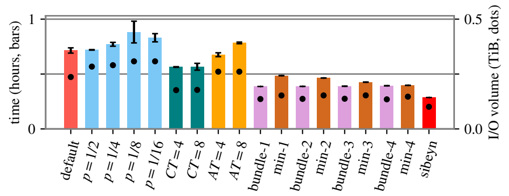


**(a)** $m = 3.89 \times 10^9$, $m/n = 11.96$

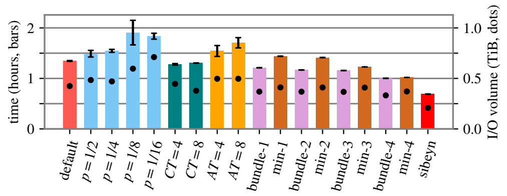


**(b)** $m = 4.05 \times 10^9$, $m/n = 12.00$

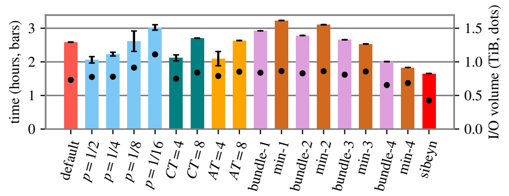
**Figure 15** Running times and I/O volumes for cubes with the parameters (a) one layer and $(w, h, d) = (18\,000, 18\,000, 2)$ and (b) 100 layers and $(w, h, d) = (2600, 1300, 2)$.



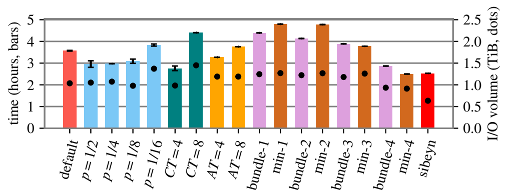
**(a)** $n = 9.18 \times 10^8$, $m = 1.04 \times 10^9$, $m/n = 1.14$



**(b)** $n = 1.06 \times 10^9$, $m = 2.62 \times 10^9$, $m/n = 2.46$

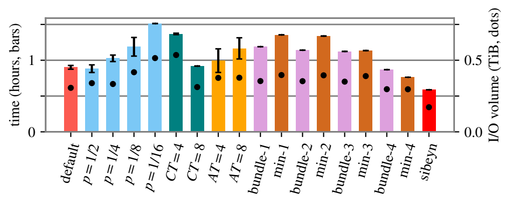


**(c)** $n = 1.07 \times 10^9$, $m = 5.48 \times 10^9$, $m/n = 5.11$

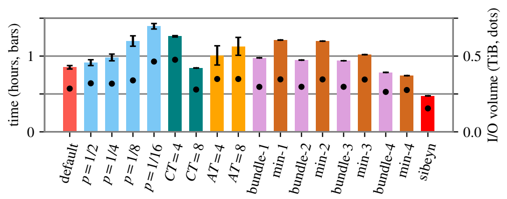


**(d)** $n = 1.07 \times 10^9$, $m = 8.14 \times 10^9$, $m/n = 7.59$

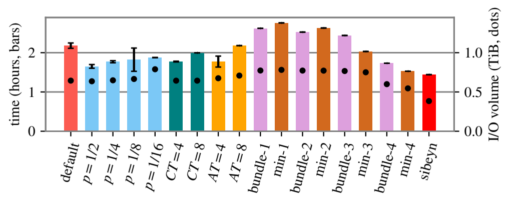
**Figure 16** Running times and I/O volumes for RGGs with roughly $n = 2^{30}$ and varying density. Node IDs are permuted.



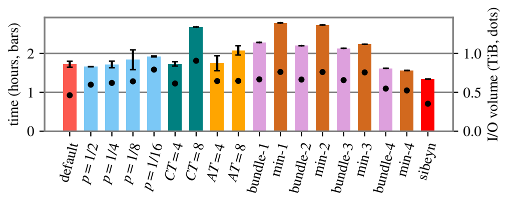
**(a)** $n = 6.54 \times 10^8$, $m = 2.61 \times 10^9$, $m/n = 3.99$, $\gamma = 3$



**(b)** $n = 6.46 \times 10^8$, $m = 2.36 \times 10^9$, $m/n = 3.65$, $\gamma = 4$



**(c)** $n = 6.70 \times 10^8$, $m = 5.57 \times 10^9$, $m/n = 8.30$, $\gamma = 3$



**(d)** $n = 6.70 \times 10^8$, $m = 5.22 \times 10^9$, $m/n = 7.80$, $\gamma = 4$

**Figure 17** Running times and I/O volumes for RHGs with roughly $n = 2^{30}$, degree exponent $\gamma \in \{3, 4\}$ and varying density. Node IDs are permuted.