UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD thesis

Jacob Holm

Efficient Graph algorithms and Data Structures

This thesis has been submitted to the PhD School of The Faculty of Science, University of Copenhagen

Principal Advisor: Mikkel Thorup
Co-Advisor: Stephen Alstrup

Submitted August 31, 2018

Author	Jacob Holm
Affiliation	Department of Computer Science University of Copenhagen
Title	Efficient Graph algorithms and Data Structures
Academic Advisors	Mikkel Thorup (principal) Stephen Alstrup
Submitted	August 31, 2018

Short Abstract

The *graph* is one of the most important abstractions used in computer science. This thesis gives substantial improvements to the state of the art regarding 4 different problems on static or dynamic graphs, namely: Static reachability in planar graphs, Online bipartite matching with recourse, Deterministic fully-dynamic 2-edge connectivity and bridge-finding, and Strong trail orientations of graphs.

Abstract

This work in theoretical computer science covers the following subjects:

Planar Reachability in Linear Space and Constant Time We show how to represent a planar digraph in linear space so that reachability queries can be answered in constant time. The data structure can be constructed in linear time. This representation of reachability is thus optimal in both time and space, and has optimal construction time. The previous best solution used $\mathcal{O}(n \log n)$ space for constant query time [Thorup FOCS'01].

Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements In the online bipartite matching problem with replacements, all the vertices on one side of the bipartition are given, and the vertices on the other side arrive one by one with all their incident edges. The goal is to maintain a maximum matching while minimizing the number of changes (replacements) to the matching. We show that the greedy algorithm that always takes the shortest augmenting path from the newly inserted vertex (denoted the SAP protocol) uses at most amortized $\mathcal{O}(\log^2 n)$ replacements per insertion, where n is the total number of vertices inserted. This is the first analysis to achieve a polylogarithmic number of replacements for *any* replacement strategy, almost matching the $\Omega(\log n)$ lower bound. The previous best strategy known achieved amortized $\mathcal{O}(\sqrt{n})$ replacements [Bosek, Leniowski, Sankowski, Zych, FOCS 2014]. For the SAP protocol in particular, nothing better than the trivial $\mathcal{O}(n)$ bound was known except in special cases. Our analysis immediately implies the same upper bound of $\mathcal{O}(\log^2 n)$ reassignments for the capacitated assignment problem, where each vertex on the static side of the bipartition is initialized with the capacity to serve a number of vertices.

We also analyze the problem of minimizing the maximum server load. We show that if the final graph has maximum server load L, then the SAP protocol makes amortized $\mathcal{O}(\min\{L \log^2 n, \sqrt{n} \log n\})$ reassignments. We also show that this is close to tight because $\Omega(\min\{L, \sqrt{n}\})$ reassignments can be necessary.

Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time We present a deterministic fully-dynamic data structure for maintaining information about the bridges in a graph. We support updates in $\widetilde{\mathcal{O}}((\log n)^2)$ amortized time, and can find a bridge in the component of any given vertex, or a bridge separating any two given vertices, in $\mathcal{O}(\log n / \log \log n)$ worst case time. Our bounds match the current best bounds for deterministic fully-dynamic connectivity up to $\log \log n$ factors.

The previous best dynamic bridge finding algorithm was an $\widetilde{\mathcal{O}}((\log n)^3)$ amortized time algorithm by Thorup [STOC2000], which was a bit-trick-based improvement on the $\mathcal{O}((\log n)^4)$ amortized time algorithm by Holm et al. [STOC98, JACM2001].

Our approach is based on a different and purely combinatorial improvement of the algorithm of Holm et al., which by itself gives a new combinatorial $\widetilde{\mathcal{O}}((\log n)^3)$ amortized time algorithm. Combining it with Thorup's bit-trick, we get down to the claimed $\widetilde{\mathcal{O}}((\log n)^2)$ amortized time.

Essentially the same new trick can be applied to the biconnectivity data structure from [STOC98, JACM2001], improving the amortized update time to $\widetilde{\mathcal{O}}((\log n)^3)$.

We also offer improvements in space. We describe a general trick, which applies both to our new algorithms and to the old ones, to get down to linear space, where the previous best use $\mathcal{O}(m + n \log n \log \log n)$.

Our result yields an improved running time for deciding whether a unique perfect matching exists in a static graph.

One-Way Trail Orientations Given a graph, does there exist an orientation of the edges such that the resulting directed graph is strongly connected? Robbins' theorem [Robbins, Am. Math. Monthly, 1939] asserts that such an orientation exists if and only if the graph is 2-edge connected. A natural extension of this problem is the following: Suppose that the edges of the graph are partitioned into trails. Can the trails be oriented consistently such that the resulting directed graph is strongly connected?

We show that 2-edge connectivity is again a sufficient condition and we provide a linear time algorithm for finding such an orientation.

The generalised Robbins' theorem [Boesch, Am. Math. Monthly, 1980] for mixed multigraphs asserts that the undirected edges of a mixed multigraph can be oriented to make the resulting directed graph strongly connected exactly when the mixed graph is strongly connected and the underlying graph is bridgeless.

We consider the natural extension where the undirected edges of a mixed multigraph are partitioned into trails. It turns out that in this case the condition of the generalised Robbin's Theorem is not sufficient. However, we show that as long as each cut either contains at least 2 undirected edges or directed edges in both directions, there exists an orientation of the trails such that the resulting directed graph is strongly connected. Moreover, if the condition is satisfied, we may start by orienting an arbitrary trail in an arbitrary direction. Using this result one obtains a very simple polynomial time algorithm for finding a strong trail orientation if it exists, both in the undirected and the mixed setting.

Dansk Resumé (Danish Abstract)

Dette arbejde indenfor teoretisk datalogi dækker følgende emner:

Fremkommelighed i Orienterede Plane Grafer Vi viser, hvordan man kan repræsentere en orienteret plan graf, med kun lineært forbrug af plads, så man i konstant tid kan afgøre, om der findes en orienteret vej mellem vilkårlige to knuder. Datastrukturen kan konstrueres i lineær tid, og er derfor optimal både i tid, plads, og konstruktionstid. Den hidtil bedste løsning brugte $\mathcal{O}(n \log n)$ plads for at opnå konstant forespørgselstid.

Online Bipartit Matching med Amortiseret $\mathcal{O}(\log^2 n)$ Ændringer I problemet kaldet "Online bipartit matching med ændringer", er knuderne på den ene side af en bipartit graf givet på forhånd, mens knuderne på den anden side afsløres een ad gangen sammen med alle deres kanter. Målet er, til enhver tid at vedligeholde en "matching" af maksimal størrelse, men at gøre det med færrest mulige ændringer til matchingen undervejs. Vi viser, at enhver grådig algoritme der til enhver tid vælger at ændre langs en korteste vej fra senest ankomne knude (kaldet SAP-protokollen) laver højest amortiseret $\mathcal{O}(\log^2 n)$ ændringer per ny knude, hvor n er det totale antal knuder der indsættes. Dette er den første analyse, der viser et polylogaritmisk antal ændringer for *nogen* protokol, og er tæt på den nedre grænse på $\Omega(\log n)$. Den hidtil bedste analyse opnåede amortiseret $\mathcal{O}(\sqrt{n})$ ændringer [Bosek, Leniowski, Sankowski, Zych, FOCS 2014]. For SAP-protokollen var intet kendt udover den trivielle øvre grænse på $\mathcal{O}(n)$. Det følger umiddelbart af vores analyse, at den samme øvre grænse på $\mathcal{O}(\log^2 n)$ ændringer også gælder for den generaliserede version hvor hver af de statiske knuder har en (på forhånd givet) kapacitet som kan være større end 1.

Vi analyserer også en variant af problemet, hvor målet er at minimere den maksimale belastning på de statiske knuder. Vi viser, at hvis den graf vi slutter med har optimal maksimal belastning L, så laver SAP-protokollen amortiseret $\mathcal{O}(\min\{L \log^2 n, \sqrt{n} \log n\})$ ændringer. Vi viser også, at dette er tæt på optimalt, idet $\Omega(\min\{L, \sqrt{n}\})$ ændringer kan være nødvendige.

Dynamisk Bro-Søgning i $\widetilde{\mathcal{O}}(\log^2 n)$ Amortiseret Tid Vi giver en deterministisk, fuldt-dynamisk datastruktur, som vedligeholder informationer om "broer" i en graf. Strukturen understøtter opdateringer i $\widetilde{\mathcal{O}}(\log^2 n)$ amortiseret tid og kan finde en bro i den komponent der indeholder en given knude, eller en bro der adskiller to givne knuder, i $\mathcal{O}(\log n / \log \log n)$ tid i værste fald. Køretiden afviger således kun med $\log \log n$ faktorer fra den bedste deterministiske fuldt-dynamiske datastruktur der kan afgøre om to givne knuder er sammenhængende.

Den hidtil bedste datastruktur for bro-søgning, var en $\widetilde{\mathcal{O}}(\log^3 n)$-tids datastruktur fra Thorup [STOC2000], som var en bit-trick-baseret forbedring af $\mathcal{O}(\log^4 n)$-tids datastrukturen fra Holm et al. [STOC98,JACM2001].

Vores tilgang er baseret på en anden, rent kombinatorisk, forbedring af datastrukturen fra Holm et al., som i sig selv giver en ny $\widetilde{\mathcal{O}}(\log^3 n)$-tids datastruktur. I kombination med Thorups bit-tricks giver det så den påståede $\widetilde{\mathcal{O}}(\log^2 n)$ amortiserede tid.

Omtrent det samme nye kombinatoriske trick kan anvendes på datastrukturen for 2-sammenhæng fra [STOC98,JACM2001], hvilket forbedrer dennes amortiserede opdateringstid til $\widetilde{\mathcal{O}}(\log^3 n)$.

Vi forbedrer også pladsforbruget. Til det bruger vi et generelt trick, som kan anvendes på både de nye og de gamle algoritmer, og som bringer pladsforbruget ned til lineært hvor det tidligere bedste var $\mathcal{O}(m + n \log n \log \log n)$.

Vores resultat forbedrer køretiden for at afgøre om en graf har en unik perfekt matching.

Ensretning af veje Givet en graf, findes der en orientering af kanterne så den resulterende graf er stærkt sammenhængende? Robbins' theorem [Robbins, Am. Math. Monthly, 1939] siger, at en sådan orientering findes hvis og kun hvis grafen er 2-kant sammenhængende. En naturlig generalisering af problemet er: Antag, at kanterne i grafen er fordelt på veje. Kan disse veje så orienteres konsistent, så den resulterende graf er stærkt sammenhængende?

Vi viser, at 2-kant sammenhæng igen er en tilstrækkelig betingelse, og vi giver en lineær-tids algoritme for at finde en sådan orientering.

Det generaliserede Robbins' Theorem [Boesch, Am. Math. Monthly, 1980] for blandede multigrafer siger, at de ikke-orienterede kanter i en blandet multigraf kan orienteres så den resulterende graf er stærkt sammenhængende, hvis og kun hvis den blandede graf er stærkt sammenhængende og den underliggende ikke-orienterede graf er 2-kant sammenhængende.

Vi betragter den naturlige generalisering, hvor de ikke-orienterede kanter i en blandet graf er fordelt på veje. Det viser sig, at i dette tilfælde er betingelsen fra det generaliserede Robbins' Theorem ikke tilstrækkelig. I stedet viser vi, at så længe ethvert cut enten indeholder mindst 2 ikke-orienterede kanter, eller indeholder orienterede kanter der krydser i begge retninger, så findes en orientering af vejene så den resulterende graf er stærkt sammenhængende. Ydermere, hvis betingelsen er opfyldt, kan vi starte med at orientere en vilkårlig vej i vilkårlig retning. Med brug af dette resultat, får man en meget simpel polynomiel-tids algoritme for at finde en stærk orientering af vejene hvis den findes, både i det ikke-orienterede og det blandede tilfælde.

Acknowledgements

I would like to thank my supervisors Mikkel Thorup and Stephen Alstrup, first for getting me interested in graph problems and data structures all those many years ago, and second for being so supportive when I decided to return to academia after my long break.

That decision might never have been made had it not been for the intervention of my long time friend and old collaborator Kristian de Lichtenberg. The masters thesis I co-authored with Kristian back in 1998, and some of the discussions surrounding it, are direct predecessors to some of the work in this thesis. And had he not invited me to that talk by Bob Tarjan on August 23. 2013 at DIKU (just over 5 years ago), it is doubtful that I would be where I am today.

Also a huge thanks to my most consistent and excellent collaborator Eva Rotenberg, who I first met at that aforementioned talk. Kristian may have been the one who got me there, but Eva arranged the weekly meetings that made me realize that I still have a place in academia, and not just as a hobby. And then it turned out we make an excellent team, as evidenced by our 10 papers together so far (9 of them made during my PhD).

I would also like to thank Uri Zwick for being an excellent host during my 6 week stay in Tel Aviv. Too bad none of the things we discussed then bore fruit. The same goes for Virginia Vassilevska Williams and MIT. It was only a brief visit, but I had a great time while I was there. My visit to Pino Italiano at Tor Vergata and Jakub Łącki at Sapienza, both in Rome was not only fun, but also very productive.

A big thanks also to all my co-authors during my PhD (in alphabetic order): Anders Aamand, Mikkel Abrahamsen, Stephen Alstrup, Aaron Bernstein, Niklas Hjuler, Guiseppe F. Italiano, Adam Karczmarz, Mathias Bæk Tejs Knudsen, Jakub Łącki, Eva Rotenberg, Piotr Sankowski, Morten Stöckel, Mikkel Thorup, and Christian Wulff-Nilsen. It has been an absolute pleasure working with all of you, and I am proud of what we have achieved.

Thanks to Alan Roytman for proofreading the almost-final version of this thesis.

Finally, a big thanks to my extended family, for their patience and support through the years, for being there when I needed it, but also for giving me space when I needed that.

Preface

This thesis is written as "a synopsis with manuscripts of papers or already published papers attached" in accordance with the formal requirements laid out in Section 5.1 of the "General rules and guidelines for the PhD programme"[1] at the Faculty of Science, University of Copenhagen. All results listed here were finalized during my enrollment as a Ph.D. student from September 2015 to August 2018. They consist of 9 papers published in peer reviewed conferences [Aam+18; Abr+17a; Abr+17c; BHR18; Hol+18; Hol+17; HRT15; HRT18] or journals [HR17], and 2 submitted manuscripts [Abr+17b; HR18]. Of these, 2 of the published conference papers [BHR18; Hol+18] have won "Best Paper" awards.

For brevity, this thesis will discuss only 4 of these results [Aam+18; BHR18; HRT15; HRT18]. For completeness, I include the full list here.

[Aam+18] Anders Aamand, Niklas Hjuler, Jacob Holm, and Eva Rotenberg. "One-Way Trail Orientations". In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. July 2018, 6:1–6:13. DOI: 10.4230/LIPIcs.ICALP.2018.6.

[Abr+17a] Mikkel Abrahamsen, Stephen Alstrup, Jacob Holm, Mathias Bæk Tejs Knudsen, and Morten Stöckel. "Near-Optimal Induced Universal Graphs for Bounded Degree Graphs". In: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. **Note that my name is incorrectly listed as "Stephen Holm" in the proceedings.** 2017. ISBN: 978-3-95977-041-5. DOI: 10.4230/LIPIcs.ICALP.2017.128.

[Abr+17b] Mikkel Abrahamsen, Stephen Alstrup, Jacob Holm, Mathias Bæk Tejs Knudsen, and Morten Stöckel. "Near-Optimal Induced Universal Graphs for Bounded Degree Graphs". In: *CoRR* abs/1607.04911 (2017). **Split in two for publication. One part published as [Abr+17a], other part submitted to Discrete Applied Mathematics**. arXiv: 1607.04911.

[Abr+17c] Mikkel Abrahamsen, Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. "Best Laid Plans of Lions and Men". In: *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*. Ed. by Boris Aronov and Matthew J. Katz. Vol. 77. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 6:1–6:16. DOI: 10.4230/LIPIcs.SoCG.2017.6.

[1] See https://www.science.ku.dk/english/research/phd/student/filer/regelsaet/SCIENCE_regels_t_2015_FINAL.pdf.

[BHR18] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. "Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Ed. by Artur Czumaj. **SODA 2018 best paper**. SIAM, 2018, pp. 947–959. DOI: 10.1137/1.9781611975031.61.

[Hol+18] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Eva Rotenberg. "Decremental SPQR-trees for Planar Graphs". In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Ed. by Yossi Azar, Hannah Bast, and Grzegorz Herman. Vol. 112. Leibniz International Proceedings in Informatics (LIPIcs). **ESA 2018 best paper**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 46:1–46:16. ISBN: 978-3-95977-081-1. DOI: 10.4230/LIPIcs.ESA.2018.46.

[Hol+17] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. "Contracting a Planar Graph Efficiently". In: *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 50:1–50:15. DOI: 10.4230/LIPIcs.ESA.2017.50.

[HR17] Jacob Holm and Eva Rotenberg. "Dynamic Planar Embeddings of Dynamic Graphs". In: *Theory Comput. Syst.* 61.4 (2017). **Announced at STACS'15**, pp. 1054–1083. DOI: 10.1007/s00224-017-9768-7.

[HR18] Jacob Holm and Eva Rotenberg. "Good r-divisions Imply Optimal Amortised Decremental Biconnectivity". In: *CoRR* abs/1808.02568 (Aug. 2018). **Submitted to SODA 2019**. arXiv: 1808.02568.

[HRT15] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Planar Reachability in Linear Space and Constant Time". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. Ed. by Venkatesan Guruswami. IEEE Computer Society, 2015, pp. 370–389. DOI: 10.1109/FOCS.2015.30.

[HRT18] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Ed. by Artur Czumaj. SIAM, Jan. 2018, pp. 35–52. DOI: 10.1137/1.9781611975031.3.

The papers I have not included in this thesis fall roughly into 4 categories. I will discuss them briefly here.

Induced universal graphs and adjacency labelling schemes [Abr+17a; Abr+17b] These two papers are really part of the same project, which aims to find the smallest *induced universal graph* for the family of graphs with n vertices and degree at most D. The first paper shows new bounds for a wide range of values of D. The second specifically treats the case $D = 2$.

Pursuit and evasion games in the plane [Abr+17c] This paper contains two main results about the classical "man and lion" game. The first answers a question dating back to J.E. Littlewood (1885–1977) by showing that two lions are not always enough to catch a man in a bounded region with obstacles. The second result is that a fast man can escape arbitrarily many slightly slower lions in an unbounded region without obstacles. In the arXiv version [Abr+17d] we even extend this to show that the man can survive against any countably infinite set of lions.

Unfortunately, after publication it has come to our attention that the first of these results had essentially already been shown by Bhadauria et al. [Bha+12]

Decremental algorithms for planar/separable graphs [Hol+18; Hol+17; HR18] In [Hol+17] we show how to maintain a planar graph under edge contractions in linear total time, while supporting adjacency queries in worst case constant time, as well as providing explicit neighbor lists. We then apply this data structure to get optimal decremental algorithms for bridge-finding, 2-edge connectivity, and maximal 3-edge connected components on planar graphs, as well as an optimal algorithm for determining if a planar graph has a unique perfect matching. We also improve the best decremental algorithms for 2-vertex- and 3-edge connectivity.

In [Hol+18] we continue this work by showing a decremental $\mathcal{O}(\log^2 n)$ algorithm for maintaining the SPQR-tree and the 3-vertex connected components of a planar graph.

In [HR18] we show that sometimes we can use a pair of suitable r-divisions to turn a fully-dynamic algorithm with polylogarithmic update time into a decremental algorithm with constant update time. In this particular case, we show how to do this for connectivity and 2-vertex connectivity. Thus giving optimal algorithms for these problems on e.g. minor-free graphs.

Planarity testing [HR17] This paper shows how to maintain a fully-dynamic plane graph under certain changes to the embedding in $\mathcal{O}(\log^2 n)$ worst case time per operation. The structure can not only tell whether a given edge insertion is valid in the current embedding. If a single change to the embedding is sufficient to allow the insertion of an edge, then the structure can also find this change. This result may not seem that impressive by itself, but it is an essential part of my "master plan" to one day do fully dynamic planarity testing in polylogarithmic time.

Contents

Part I
Synopsis

This part comprises an overview of the results presented in the thesis.

1 Introduction

One of the most important abstractions used in computer science is that of a *graph*. Intuitively, a graph is simply a collection of "things" (usually called *nodes* or *vertices*), where some pairs of "things" are related (we say there is an *edge* or an *arc* between them) and others are not. Simple examples of graphs include e.g.:

geographical road maps Here vertices are intersections (and other points of interest), and edges are the roads connecting them.

facebook friendships Here vertices are people (facebook accounts), and edges correspond to facebook "friends".

chemical molecules Here vertices might represent individual atoms, and edges the covalent bonds between them.

Because graphs are ubiquitous, many real-world problems involve or can be reduced to a graph problem. Conversely, solving a single graph problem has the potential to solve many seemingly different real-world problems.

But what do we mean by *solving* a graph problem? Since this is a theoretical work, we will stop short of actually writing a computer program, and call a problem solved when we have found an *algorithm* (a recipe that a computer can follow) and an associated *data structure* (a representation of the problem in computer memory) for the problem.

Since the graphs we need to work with are often large (millions or even billions of vertices), we need our solutions to be *efficient* in terms of both time and space used. In particular, we need the time (number of steps used in the algorithm) and space (amount of memory used by the data structure) to grow "slowly" (if at all) with the problem size, because otherwise even a moderately large graph will be too large to handle even for the best computers.

As the title implies, the goal of this thesis is to describe efficient algorithms and data structures for certain graph problems.

1.1 Outline

In Section 2 we will briefly define the terms and notation used in the rest of this synopsis. Then in each of Sections 3, 4, 5, and 6 we describe a graph problem and its solution in one of the papers included in the appendix. Finally, in Section 7 we add some concluding remarks.

Note that with the exception of a few remarks in Subsection 4.4 and all of Subsection 6.4, there is nothing in these chapters that is not also in the corresponding papers in the appendices.

The appendices contain the following papers, each of which is an extended version of one of my peer-reviewed conference papers:

Appendix A Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Planar Reachability in Linear Space and Constant Time". In: *ArXiv e-prints* (Nov. 2014). **Extended version of [HRT15]**. arXiv: 1411.5867 [cs.DS]

Appendix B Aaron Bernstein, Jacob Holm, and Eva Rotenberg. "Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements". In: *ArXiv e-prints* (July 2017). **Extended version of [BHR18], submitted to JACM**. arXiv: 1707.06063 [cs.DS]

Appendix C Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time". In: *ArXiv e-prints* (July 2017). **Extended version of [HRT18]**. arXiv: 1707.06311 [cs.DS]

Appendix D Anders Aamand, Niklas Hjuler, Jacob Holm, and Eva Rotenberg. "One-Way Trail Orientations". In: *ArXiv e-prints* (Aug. 2017). **Extended version of [Aam+18]**. arXiv: 1708.07389 [cs.DS]

The papers are included exactly as they appear on the arXiv preprint server at the time of writing, except: 1) They have been scaled to fit the pages of this thesis, and 2) Each page has an added header with (among other things) the running page number in this thesis.

In particular, no attempt has been made to unify the notation and terminology for inclusion in this thesis. Thus, some minor inconsistencies are to be expected.

2 Preliminaries

2.1 Machine models

Before we can even start to talk about being efficient in *time* and *space*, we need to pin down exactly what we mean by those words. Since we are doing theory, we can't just implement it on a computer, ask it to run a few examples, and measure the time/memory used. Not because the algorithms can't be implemented (they can). Not even because such measurements wouldn't be useful for evaluating practical uses (they would). The simple reason is that we want to know how our algorithms perform *asymptotically*, so we can predict how they scale with the problem size.

Instead, we will be using an abstract model of a computer. For all algorithms and data structures mentioned in this thesis, unless otherwise noted, we will be using the model known as "Word RAM with word size $w \in \Theta(\log n)$ and standard AC^0 operations". Here is what that means:

Word RAM means that our memory is partitioned into cells called *words*, and that we can access (read or write) any word in memory in constant time given its address.

word size $w \in \Theta(\log n)$ means that the words of our machine each hold enough bits that the size of our problem, denoted n, can be stored in at most some

constant number of them, but not enough bits to do "unreasonable" things, like representing the whole problem in a single word. (We'll get back to that Θ notation later).

standard AC^0 operations Standard operations means that what our machine supports are the operations we are used to from a programming language such as C, e.g. addition, subtraction, bitwise and/or/xor, bit shifts, etc.

AC^0 means that we only allow operations that can be computed by a circuit of constant "depth", unlimited fan-in, and size polynomial in w. Essentially that is the same as saying that the hardware needed to implement the operation can be made to run in constant time independent of the word size, and that it does not take up "too much" space on the chip. In particular, this means we don't allow multiplication or division of arbitrary numbers as elementary operations.

In this model, whenever we talk about the *space* used by a data structure, we mean the number of w-bit words of memory used. And the *time* is really the number of elementary operations used.

In contrast, for most lower bounds mentioned, we will be assuming the "cell-probe" model. This is similar to the "Word RAM with word size $\Theta(\log n)$" model, except we allow any computation whatsoever for free, and redefine *time* as the number of memory cells accessed.

2.2 Asymptotic notation: $\mathcal{O}$, Ω, Θ, and $\widetilde{\mathcal{O}}$

Now that we have a definition of time and space, we are almost ready to start a discussion of what it means to be *efficient*. However, we want to make sure that our discussion happens at the *right* level of abstraction, and counting the exact number of instructions executed by an algorithm is both cumbersome, and not really transferable between similar machine models (which may e.g. have different but equally good instruction sets). So rather than counting the exact number of instructions $f(n)$ executed for some problem of size n and finding that e.g. $f(n) \leq 4n^3 + 6n^2 \log n + 42 \log n$, we will use asymptotic notation[2] and just write that $f(n) \in \mathcal{O}(n^3)$ (or informally that $f(n)$ is $\mathcal{O}(n^3)$, or even that $f(n) = \mathcal{O}(n^3)$). Similarly, instead of writing e.g. that $f(n) \geq 5n^2/\log n + n \log n$, we will say that $f(n) \in \Omega(n^2/\log n)$. Finally, if both $f(n) \in \mathcal{O}(g(n))$ and $f(n) \in \Omega(g(n))$ for some function g, then we write $f(n) \in \Theta(g(n))$. In each of these cases, we ignore constant factors and keep only the *dominating term*, meaning the term that determines the growth rate of the function in question as the parameter(s) grow large. Sometimes, we even go a little bit further and drop not just constant factors, but also logarithmic factors. We use the notation $f(n) \in \widetilde{\mathcal{O}}(g(n))$ to mean that $f(n)$ is $\mathcal{O}(g(n))$ except for logarithmic factors of $g(n)$. For example, this means that both $\log^2 n / \log\log n \in \widetilde{\mathcal{O}}(\log^2 n)$ and $\log^2 n \cdot \log^2 \log n \in \widetilde{\mathcal{O}}(\log^2 n)$.

[2] Sometimes called Landau notation.

$$
\begin{aligned}
f(n) \in \mathcal{O}(g(n)) &\iff \exists k > 0\, \exists N\, \forall n \geq N : |f(n)| \leq k \cdot g(n) \\
f(n) \in \Omega(g(n)) &\iff \exists k > 0\, \exists N\, \forall n \geq N : |f(n)| \geq k \cdot g(n) \\
f(n) \in \Theta(g(n)) &\iff f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n)) \\
f(n) \in o(g(n)) &\iff \forall k > 0\, \exists N\, \forall n \geq N : |f(n)| < k \cdot g(n) \\
f(n) \in \omega(g(n)) &\iff \forall k > 0\, \exists N\, \forall n \geq N : |f(n)| > k \cdot g(n) \\
f(n) \in \widetilde{\mathcal{O}}(g(n)) &\iff \exists k \in \mathbb{R} : f(n) \in O(g(n) \log^k \max\{1, g(n)\})
\end{aligned}
$$

Figure 1: For single-variable functions $f(n)$ and $g(n) \geq 0$, here is a formal definition of the asymptotic notation used in this paper. The correct definition for functions of multiple variables is much more subtle (See e.g. [How08]).

The point is that, with the proper definition of the notation (See Figure 1), we can do calculations without having to care about irrelevant details. E.g. if $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))$ and $f_1(n) \cdot f_2(n) \in \mathcal{O}(g_1(n) \cdot g_2(n))$, and similar rules apply to Ω, Θ, and $\widetilde{\mathcal{O}}$.

As the notation implies, $\mathcal{O}(g(n))$ is really a set of functions, and it is useful to know how these sets for different functions are related. Some representative examples used in this thesis are shown in Figure 2.

To summarize, when we describe how efficient our algorithms are, we usually find "nice" functions $t(n)$ and $s(n)$ such that the total number of w-bit words of memory space used for the data structure is $\mathcal{O}(s(n))$, and the number of standard AC^0 instructions used is $\mathcal{O}(t(n))$.

Similarly, for lower bounds we typically describe functions such that the number of words used is $\Omega(s(n))$ and the number of memory accesses is $\Omega(t(n))$.

2.3 Graphs

With the technicalities of the abstract machine model and how to measure time and space out of the way, we are finally ready to start discussing the main subject of this thesis, namely graphs.

Formally, a graph is a pair (V, E), consisting of two sets:

V is the called the set of *vertices*, and we usually let $n = |V|$.

E is is called the set of *edges*, and we usually let $m = |E|$.

Each edge has an associated ordered or unordered pair of (not necessarily distinct) vertices, called its end vertices. An edge is a *self-loop* if its end vertices are the same. It is *directed* if its associated pair of end vertices is ordered, and it is *undirected* otherwise. For a directed edge $e \in E$ with end vertices (u, v), we call u the *tail*,

$$
\begin{aligned}
\mathcal{O}(\log n/\log\log n) &\subset \mathcal{O}(\log n) \\
&\subset \mathcal{O}(\log n \cdot \log^2 \log n) \\
&\subset \mathcal{O}(\log n \cdot \log^3 \log n) && \subset \widetilde{\mathcal{O}}(\log n) \\
&\subset \mathcal{O}(\log^2 n/\log\log n) \\
&\subset \mathcal{O}(\log^2 n) \\
&\subset \mathcal{O}(\log^2 n \cdot \log^2 \log n) && \subset \widetilde{\mathcal{O}}(\log^2 n) \\
&\subset \mathcal{O}(\log^3 n \cdot \log\log n) && \subset \widetilde{\mathcal{O}}(\log^3 n) \\
&\subset \mathcal{O}(\log^4 n) && \subset \widetilde{\mathcal{O}}(\log^4 n) \\
&\subset \mathcal{O}(n^{o(1)}) && \subset \widetilde{\mathcal{O}}(n^{\varepsilon}) && \subset \widetilde{\mathcal{O}}(n^{\frac{1}{2}-\varepsilon}) \\
&\subset \mathcal{O}\left(\sqrt{\frac{n}{\log n}}\log\log n\right) \\
&\subset \mathcal{O}(\sqrt{n}) && \subset \widetilde{\mathcal{O}}(\sqrt{n}) \\
&\subset \mathcal{O}(n^{2/3}) && \subset \widetilde{\mathcal{O}}(n^{2/3}) \\
&\subset \mathcal{O}(n) \\
&\subset \mathcal{O}(n\alpha(n)) \\
&\subset \mathcal{O}(n \log n) \\
&\subset \mathcal{O}(n \log^2 n) && \subset \widetilde{\mathcal{O}}(n) \\
&\subset \mathcal{O}(n^{1+\Omega(1)})
\end{aligned}
$$

Figure 2: The complexity classes used in this paper, in order from most to least efficient. Note that we use $\subset$ and $\subseteq$ analogously to $<$ and $\leq$, so in particular $A \subset B$ means $A \subseteq B$ and $A \neq B$. Some authors, especially in the mathematical community, prefer to use $\subsetneq$ and $\subset$ instead. ε here is any constant $0 < \varepsilon < \frac{1}{4}$. $\alpha(n)$ here is the extremely slowly growing inverse of *Ackermanns function*.

and v the *head* of e. When drawing a directed edge, we usually draw it as an arrow, pointing from the tail to the head. A directed edge is sometimes called an *arc*.

Similarly, each vertex v has a set of *incident* edges, which is the set of edges that have v as one of its associated end vertices. The *degree* of v, denoted $d(v)$, is the number of times v occurs as an end vertex (counting self-loops twice). The *in-degree* of a vertex v, denoted $d_{\text{IN}}(v)$, is the number of directed edges whose head is v. Similarly, the *out-degree*, denoted $d_{\text{OUT}}(v)$, is the number of directed edges whose tail is v.

A graph is said to be a *subgraph* of another graph, if the first graph consists of a subset of the vertices and a subset of the edges of the second. An *induced* subgraph is a graph defined by a subset of vertices and all edges with both end vertices in that subset.

A graph is called *loopless* if none of its edges are self-loops. For the rest of this thesis, we will assume that all graphs are loopless. A graph is called *simple* if no two edges are associated with the same (ordered or unordered) pair of end vertices, and otherwise it is called a *multigraph*. A graph is *undirected* (resp) *directed* if all its edges are. A directed graph is also called a *digraph*. A graph with both directed and undirected edges is called *mixed*.

A graph is *planar*, if it can be drawn in the plane such that each vertex is a distinct point, each edge is a simple curve connecting the points corresponding to its end vertices, and no two of these edge curves intersect except at their end points. Such a drawing is called an *embedding* of the graph (in the plane). Any such embedding defines a clockwise cyclic order of the edges incident to each vertex. An assignment of such a clockwise cyclic order for each vertex is called a *topological embedding*. A planar graph, together with a topological embedding that corresponds to some embedding of the graph in the plane is called a *plane* graph. A useful fact about planar graphs is that a simple, undirected planar graph with $n \geq 3$ vertices has at most $m \leq 3n - 6$ edges, and a simple directed planar graph with at least $n \geq 3$ vertices has at most $m \leq 6n - 12$ edges. Thus a simple planar graph has at most $\mathcal{O}(n)$ edges. This is in contrast with the general case for simple graphs, which may have as many as $m = \binom{n}{2} = \frac{n(n-1)}{2} = \Omega(n^2)$ edges, or for multigraphs where there is no upper bound.

A graph is *bipartite* if we can partition its vertices into two sets $V = C \cup S$, $C \cap S = \emptyset$, such that every edge has one end vertex in each set.

A *matching* in a graph is a subset of the edges, such that no two edges in the matching have an end vertex in common. A matching is *maximal* if every edge not in the matching shares an end vertex with some edge in the matching. A matching is *maximum* if it has the maximum size over all possible matchings.

Undirected graphs

A *walk* (of length k) in an undirected graph is pair consisting of a sequence of vertices $v_0, \ldots, v_k \in V$, and a sequence of edges $e_1, \ldots, e_k \in E$, such that for each $i = 1, \ldots, k$ the end vertices of e_i are $\{v_{i-1}, v_i\}$. A walk is called a *trail* if all

its edges are distinct, and a *(simple) path* if all its vertices are distinct. Note that the edges of a walk uniquely determine the vertices, so we will sometimes think of a walk/trail/path as just a sequence of edges. Similarly, in a simple graph, the sequence of vertices on a walk uniquely determine the edges, so for simple graphs we sometimes think of a walk/trail/path as just a sequence of vertices.

A walk or trail is *closed* if its first and last vertex are the same. A *(simple) cycle* is a closed trail with as many distinct vertices as edges.

A pair of vertices u, v are connected if there exists a path with u and v as first and last vertices respectively. A graph is *connected* if every pair of vertices u, v in it are connected. A *connected component* in a graph is a maximal connected subgraph.

An edge is a *bridge* (also known as a 1-*cut*) if deleting it from the graph increases the number of connected components. Similarly, for $k \geq 1$, a set of k edges is a k-*cut* if deleting all edges in the set increases the number of connected components. A pair of vertices u, v are k-*edge connected* if they are connected in the graph obtained by removing any $k - 1$ edges. By Menger's Theorem [Men27] this is equivalent to saying that there are k edge-disjoint paths connecting them. In particular, if u and v are connected but not 2-edge connected, any path between u and v contains a bridge. In fact, every path from u to v contains exactly the same bridges, in the same order. A graph is k-*edge connected* if every pair of vertices in it are k-edge connected. A k-edge connected component is a maximal subset of the vertices that are pairwise k-edge connected. Every vertex is in exactly one k-edge connected component. For $k \leq 2$ each k-edge connected component induces a connected subgraph. For $k > 2$ this is not the case in general.

k-edge connectivity is a natural generalization of connectivity. In particular, 1-edge connectivity is exactly the same as connectivity. Another natural generalization is called k-vertex connectivity, or just k-connectivity, but that is beyond the scope of this thesis.

Directed and mixed graphs

A *directed walk* in a directed or mixed graph is defined similarly as a walk in an undirected graph, but with the additional requirement that for $i \in 1, \ldots, k$ if e_i is directed then v_{i-1} must be the tail and v_i be the head of e_i. Essentially, we are not allowed to follow an edge in the wrong direction. A directed walk is called a *directed trail* if all its edges are distinct, and a *(simple) directed path* or a *(simple) dipath* if all its vertices are distinct.

A directed walk or trail is *closed* if its first and last vertex are the same. A *(simple) directed cycle* is a closed directed trail with as many distinct vertices as edges.

A vertex v is *reachable* from a vertex u if there exists a dipath with u as first and v as last vertex. Two vertices are *strongly connected* if each is reachable from the other. A graph is *strongly connected* if every pair of vertices in it are strongly connected. The *strongly connected components* of a graph are the maximal strongly

connected subgraphs. Note that every vertex is in exactly one strongly connected component.

A *strong bridge* in a directed or mixed graph is an edge whose removal increases the number of strongly connected components.

A *strong orientation* of an undirected or mixed graph is a digraph with the same vertices and edges, but where an order has been imposed on the end vertices of each edge to make it directed, in such a way that the whole graph is strongly connected.

A digraph is *acyclic* if every strongly connected component consists of a single vertex. A vertex is called a *source* if it has in-degree 0, and a *sink* or a *target* if it has out-degree 0. Any acyclic digraph has at least one source and at least one sink. An acyclic digraph with exactly one source and one sink is sometimes called an st-*graph*.

2.4 Static vs. Dynamic graph problems

Now that we know how to talk about graphs, we need just a bit more terminology regarding the different types of problems we tend to work on.

Static

We call a graph problem *static* if we are given the whole graph at once, and then work on that graph until we have solved the problem or it is no longer interesting. This is the case for the problem we consider in two of the included papers (See Sections 3 and 6).

The alternative to a static problem is a *dynamic* problem, where part of the problem is handling the fact that the graph changes over time. We further subdivide the dynamic problems into 3 distinct categories.

Decremental

If we are given a (possibly very complicated) initial graph, and every update operation decreases (some measure of) the complexity, then we say we have a *decremental* problem. An example could be a data structure where the updates consist of deleting or contracting edges (See e.g. [Hol+18; Hol+17; HR18]). Here every update decreases the total number of edges, so after at most m updates, no more updates are possible. None of the included papers directly address this type of problem.

Incremental

If instead (some measure of) the complexity of the graph increases with each update, then we say we have an *incremental* problem. An obvious example is when the update consists of adding edges and/or vertices. For this kind of problem we can typically assume we start with an empty edge set, and sometimes even an empty vertex set (although the graph $(\emptyset, \emptyset)$ is not a well-defined concept). The problem we consider in Section 4 is of this type.

Fully dynamic

Finally, if the update operations on the graph can both decrease and increase the complexity of the graph arbitrarily, we say we have a *fully dynamic problem*. A common example is that our graph has a fixed set of vertices, and starts with an empty set of edges, but updates may insert or delete edges. The problem we consider in Section 5 is of this type.

Note that the assumption about a fixed set of vertices is not really necessary. There is a more-or-less standard *doubling technique* that can be used to remove the restriction in most cases. It works by maintaining 3 copies of the structure in parallel, each supporting a different number of vertices. The middle one of these structures is the *active* one, that we use to answer queries. Each update to the graph causes a constant number of updates in the *small* and *large* structures, and these are chosen in such a way that by the time the current graph has too few or too many vertices, one of the other structures is just right and ready to be activated.

3 Planar reachability (Synopsis of Appendix A)

Appendix A Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Planar Reachability in Linear Space and Constant Time". In: *ArXiv e-prints* (Nov. 2014). **Extended version of [HRT15]**. arXiv: `1411.5867 [cs.DS]`

3.1 Problem

Given a directed graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, the *reachability problem* is to build a data structure that can answer queries of the form:

REACHABLE(u, v)**:** given $u, v \in V$, is there a directed path from u to v?

This paper considers the restriction of this problem to (simple) planar digraphs. Note that we can trivially convert any (not necessarily simple) planar digraph into a simple planar digraph (with $\mathcal{O}(n)$ edges) in $\mathcal{O}(m + n)$ time and space without affecting reachability, by doing a radix sort (with radix n) of the edges in $\mathcal{O}(m + n)$ time, and removing duplicate edges.

3.2 Known results

Thorup and Zwick proved in [TZ05] that there are graph classes where any representation of reachability needs $\Omega(m)$ bits, and Pătraşcu proved in [P a11] that there are sparse directed graphs with $\mathcal{O}(n)$ edges such that any representation that can answer reachability correctly in constant time requires $n^{1+\Omega(1)}$ bits. Thus, for general graphs, constant query time requires at least $\Omega(m + n^{1+\Omega(1)})$ bits.

For planar digraphs, this bound does not hold. Thorup showed in [Tho04] that for planar graphs it was possible to get constant query time using $\mathcal{O}(n \log n)$ words of size $w = \Omega(\log n)$ (so $\mathcal{O}(n \log^2 n)$ bits).

Tamassia and Tollis [TT93] showed that for a very special kind of planar digraph, here called a planar st-graph[3], we can get constant query time using only $\mathcal{O}(n)$ w-bit words. A planar st-graph is an acyclic planar graph with two distinguished vertices, called the *source* s and *sink* t, such that every vertex is reachable from the source and every vertex can reach the sink.

3.3 Our result

We show that if G is simple and planar, we can build a data structure in $\mathcal{O}(n)$ time and space, such that each reachability query can be answered in worst case constant time. This is optimal (up to constant factors), because even if the graph is just a single directed path, every possible order of the vertices must use a distinct representation. Since there are $n!$ possible orders, this requires the representation of some graph to use at least $\log_2(n!) = \Omega(n \log n)$ bits, which is $\Omega(n)$ space in total.

[3] In their paper, *planar st-graph* means something different. What we call a planar st-graph, they call a *spherical* st-graph.

3.4 Techniques

We define a new type of graph decomposition, called a *good st-decomposition*, that exists for acyclic plane digraphs with a single source (see Appendix A, Definition 3.3). Essentially, an st-decomposition is a rooted tree where each node corresponds to a subgraph of G that is a so-called *truncated st-graph*, and these subgraphs partition the vertices in such a way that all dipaths go "down" in the tree and never "across" or "up". We then show that we can always, in linear time, find a *good* such partition, meaning that the tree has height $\mathcal{O}(\log n)$, and that for any subtree rooted at some node x in the decomposition, the set of edges E_x that enter that subtree from above all have their "tails" on some set of at most 4 dipaths, together called the *frame* F_x.

These features of the st-decomposition combined lets us reduce the problem of answering **REACHABLE**(u, v) on single-source acyclic plane digraphs to a question of finding, in the subgraph C_x containing u, the at most 4 "last" vertices $v_1, \ldots, v_4$ that are on some frame that can reach v, and then asking if u can reach either of them.

We then prove some deep properties of good st-decompositions (the "Crossing Lemmas" 3.28 and 3.46) that essentially allow us to represent the reachability relationships between each frame vertex and each vertex that needs to be able to find it using 9 different overlapping forests whose nodes are vertices in V, and to find each of $v_1, \ldots, v_4$ using at most a constant number of level ancestor queries in these forests. Using well-known data structures for level-ancestor (e.g. [AH00]), we can therefore find $v_1, \ldots, v_4$ in worst case constant time, and using the result by Tamassia and Tollis [TT93] we can answer each **REACHABLE**(u, v_i) in worst case constant time. Thus, for acyclic plane digraphs with a single source this gives us the required solution.

We then extend this data structure to handle a slightly larger class of graphs, which we call acyclic planar *In-Out graphs*. These are graphs which may have multiple sources, but where there is still one special source s that can reach all sinks. The main idea here is to color the vertices into red and green, such that the green vertices are exactly those that are reachable from s, and then flipping the direction of every edge with a red end vertex. This makes the graph single-source, and for every u, v where u is green and/or v is red, we can use this structure to answer **REACHABLE**(u, v) in worst case constant time. We then show that we can augment this structure to handle the final case, where u is red and v is green, still in worst case constant time per query.

Finally, we use the known reduction from [Tho04] to handle general acyclic planar digraphs, and then contracting each strongly connected component using [Tar72] lets us handle general planar digraphs.

3.5 Future work

While there are lower bounds proving that these bounds are unobtainable for general graphs, it seems likely that similar results can be achieved for e.g. graphs of bounded genus or more generally for any class of graphs defined by a set of excluded minors.

Some of the previous results (e.g. [Tho04]) had the form of *labelling schemes*, where each vertex u is assigned a (unique) b-bit label $\ell(u)$, and the answer must be computed only from $\ell(u)$ and $\ell(v)$. Labelling schemes are in some sense the ultimate distributed data structures, so finding such a labelling scheme with $b = \mathcal{O}(\log n)$ that could still answer queries in worst case constant time would be extremely interesting. Alternatively, proving that such a labelling scheme does not exist would give a nontrivial separation between the capabilities of labelling schemes and more general data structures.

4 Online Bipartite Matching (Synopsis of Appendix B)

Appendix B Aaron Bernstein, Jacob Holm, and Eva Rotenberg. "Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements". In: *ArXiv e-prints* (July 2017). **Extended version of [BHR18], submitted to JACM**. arXiv: 1707.06063 [cs.DS]

4.1 Problem

Given a bipartite graph $G = (S \cup C, E)$ where S is a fixed set of *servers*, and C is an (initially empty) set of clients, maintain a maximum cardinality matching M under the following operation:

ADDCLIENT(c, S_c)**:** given a new client c and a subset $S_c \subseteq S$ of servers, add c to C and $\{(c, s) \mid s \in S_c\}$ to E and update the matching.

For this particular problem, we are less interested in time or space usage, and more interested in analyzing the number of *replacements* during a sequence of n **ADDCLIENT** operations, where each replacement is defined as some client getting matched to a different server.

4.2 Known results

There is a very natural greedy algorithm for the problem, which for each **ADDCLIENT** just updates all clients along a so-called *shortest augmenting path*. We call any such algorithm a SAP-algorithm, or just SAP.

The problem was introduced by Grove et al. [Gro+95], who showed that $\Omega(n \log n)$ replacements may be needed, even if each client has at most 2 neighbors, and that SAP is optimal in this special case. Chaudhuri et al. [Cha+09] showed that as long as the clients are added in *random* order, the *expected* number of replacements when using SAP is $\mathcal{O}(n \log n)$, and that for this *random-arrival* version of the problem this is optimal. They posed as an open question whether SAP is optimal for the original *adversarial-arrival* version of the problem.

The problem has been investigated for a number of special classes for graphs, e.g. graphs of client-degree 2 [Gro+95], and forests [Bos+18; Bos+17; Cha+09], in each case showing matching upper and lower bounds of $\Theta(n \log n)$.

However, for general bipartite graphs, no analysis of SAP showing better than $O(n^2)$ replacements was known, and the best non-SAP algorithm (from [Bos+14]) still used $\mathcal{O}(n\sqrt{n})$ replacements.

4.3 Our Result

We show that any SAP-algorithm uses at most $\mathcal{O}(n \log^2 n)$ replacements. Furthermore, this result still holds if we allow an adversary to arbitrarily change the matching between **ADDCLIENT** operations, as long as we only count the replacements made by the SAP-algorithm.

In addition, we show that a SAP-algorithm can actually be implemented in linear space and $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$ total time, where $m = |E|$ and $n = |C|$ in the final graph. This is only a factor of $\mathcal{O}(\sqrt{\log n})$ worse than the best offline algorithm by Hopcroft and Karp [HK73].

Our analysis extends to give improved bounds for a number of other, related, problems.

4.4 Techniques

The key idea in our new analysis is to assign each server s a *server necessity*, which is a number between 0 and 1, denoted[4] $\alpha(s)$, that is independent of the current matching (See Definition 11 and 13).

While this is not obvious from the definition, we show that the server necessities are unique (Lemma 14), that they only increase (Lemma 21), and that each **ADDCLIENT**(c, S_c) only increases necessities that are already at least as large as $\min_{s \in S_c} \alpha(s)$ (Lemma 22).

We then use the server necessities to show a crucial "Expansion Lemma" (Lemma 29), which can be rephrased as: Using SAP, each **ADDCLIENT**(c, S_c) does at most $\frac{1}{\varepsilon} \ln n$ replacements, where $\varepsilon = 1 - \min_{s \in S_c} \alpha(s)$.

By combining these results, we show that for any $h > 0$, any SAP-algorithm processes at most $4n \ln(n)/h$ paths of length $> h$, and the result immediately follows[5] because in the sum

$$\sum_{i=0}^{n-1} \begin{cases} n & \text{if } i = 0 \\ 4n \ln(n)/i & \text{otherwise} \end{cases} = \mathcal{O}(n \log^2 n)$$

each path of length h is counted by exactly h terms of the sum.

For the implementation, we show that the problem of maintaining the necessary shortest augmenting paths can be transformed into a problem of maintaining shortest dipaths up to length $h = \sqrt{n}\sqrt{\log n}$ in a certain orientation of the graph (and brute forcing the at most $4 \ln(n)/h$ searches for longer paths needed in $\mathcal{O}(mn \log(n)/h)$ total time). Using known techniques (See Lemma 32) we can maintain this graph in $\mathcal{O}(mh + n \log^2 n)$ total time. Thus, the total time for the algorithm becomes $\mathcal{O}(mn \log(n)/h + mh + n \log^2 n)$ which for our choice of h is $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$.

4.5 Future Work

We do not know how tight the $4n \ln(n)/h$ bound is. If it could be improved to $\mathcal{O}(n/h)$ that would immediately prove the optimality of SAP, and improve the total running time of our online matching algorithm to match the $\mathcal{O}(m\sqrt{n})$ time of the best offline algorithm.

[4] Not to be confused with the inverse of Ackermanns function, also commonly written as $\alpha(n)$.

[5] The paper uses a slightly more involved argument for the last step. Thanks to Bartlomiej Bosek and Anna Zych for pointing out this simpler version.

5 Dynamic Bridge-Finding (Synopsis of Appendix C)

Appendix C Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time". In: *ArXiv e-prints* (July 2017). **Extended version of [HRT18]**. arXiv: `1707.06311 [cs.DS]`

5.1 Problem

Given an undirected multigraph $G = (V, E)$ where V is fixed and E is initially empty, maintain a data structure for G supporting the following operations.

INSERT-EDGE(u, v)**:** Insert a new edge e between vertices u and v, and return e.

DELETE-EDGE(e)**:** Delete the edge e.

CONNECTED(u, v)**:** Are vertices u and v connected?

2**-EDGE-CONNECTED**(u, v)**:** Are vertices u and v 2-edge-connected?

FIND-BRIDGE(u)**:** Find and return a bridge in the connected component of vertex u, or **nil** if this component is 2-edge-connected.

FIND-BRIDGE(u, v)**:** Among the bridges separating vertices u and v, return the one closest to u, or **nil** if u and v are 2-edge-connected or not connected.

FIND-SIZE(u)**:** Return the number of vertices in the connected component containing vertex u.

FIND-2**-SIZE**(u)**:** Return the number of vertices in the 2-edge-connected component containing vertex u.

5.2 Known results

In a static graph, there are simple linear-time algorithms for finding all bridges (See e.g. [Tar74]). Such an algorithm can be used to build a data structure that can answer all the queries in worst case constant time. This can be used to construct a *trivial* solution to the dynamic problem, by simply recomputing everything as needed, either immediately after each update, or before each query that is preceded by an update. This trivial solution runs in worst case $\mathcal{O}(m + n)$ time per operation.

The first non-trivial data structure for fully dynamic 2-edge-connectivity was given by Galil et al. [GI92], and had worst case $\mathcal{O}(m^{2/3})$ time per operation. This was improved by Frederickson [Fre97], to worst case $\mathcal{O}(\sqrt{m})$ update time and worst case $\mathcal{O}(\log n)$ query time. Eppstein et al. [Epp+97] then improved the worst case update time to $\mathcal{O}(\sqrt{n})$ using their *sparsification* technique.

The first 2-edge connectivity structure with polylogarithmic update time was given by Henzinger et al. [HK95]. It was a Las Vegas style randomized algorithm using $\mathcal{O}(\log^4 n)$ expected amortized time per update and worst case $\mathcal{O}(\log n / \log \log n)$ per query.

The first *deterministic* data structure with polylogarithmic update time for 2-edge connectivity was given by Holm et al. [HLT01], and used amortized $\mathcal{O}(\log^4 n)$ time per update and (with trivial modifications) $\mathcal{O}(\log n)$ time per query. This was later improved[6], by Thorup [Tho00], who used approximate counting to reduce the amortized update time to $\mathcal{O}(\log^3 n \cdot \log\log n)$.

None of the above results directly support the **FIND-BRIDGE** or **FIND-2-SIZE** queries, although some of them (e.g. [HLT01; Tho00]) can be extended to do so.

The best lower bound we have for fully dynamic 2-edge connectivity is based on the $\mathcal{O}(\log n)$ lower bound by Pătraşcu et al. [PD06] for fully dynamic connectivity via a simple reduction.

For comparison, the current best update times for connectivity data structures are: Deterministic $\mathcal{O}(\sqrt{\frac{n}{\log n}}\log\log n)$ worst case by Kejlberg-Rasmussen et al. [Kej+16], Las Vegas randomized $\mathcal{O}(n^{o(1)})$ with high probability by Nanongkai et al. [NSW17], Monte Carlo randomized $\mathcal{O}(\log^4 n)$ worst case with one-sided error by Kapron et al. [KKM13] and Gibb et al. [Gib+15], Deterministic $\mathcal{O}(\log^2 n/\log\log n)$ amortized by Wulff-Nilsen [Wul13], and Las Vegas randomized $\mathcal{O}(\log n \cdot \log^2 \log n)$ expected by Huang et al. [Hua+17].

More generally, for k-edge connectivity not much is known beyond the deterministic data structures by [Epp+97] which for fixed $k = 2, 3, 4$ or $k > 4$ handles updates in worst case $\mathcal{O}(\sqrt{n})$, $\mathcal{O}(n^{2/3})$, $\mathcal{O}(n\alpha(n))$, and $\mathcal{O}(n \log n)$ time respectively.

5.3 Our result

We present a deterministic data structure using linear space, that supports **INSERT-EDGE** and **DELETE-EDGE** operations in amortized $\mathcal{O}(\log^2 n \cdot \log^2 \log n)$ time, and **CONNECTED**, **2-EDGE-CONNECTED**, **FIND-BRIDGE** and **FIND-SIZE** in $\mathcal{O}(\log n/\log\log n)$ worst case time, and **FIND-2-SIZE** in $\mathcal{O}(\log n \cdot \log^2 \log n)$ worst case time.

In other words, ignoring factors of $\mathcal{O}(\log\log n)$, bridge-finding is now as fast as connectivity in the deterministic amortized setting.

5.4 Techniques

We carefully dissect the algorithm from [HLT01], and make it explicit how it reduces the bridge-finding problem to a certain problem of maintaining the sizes of certain subtrees of a dynamic tree. The algorithm uses *top trees* [Als+05] (See Figure 3) to keep track of these sizes, and for each *cluster* maintains an $\mathcal{O}(\log n) \times \mathcal{O}(\log n)$ matrix of subtree sizes.

The main new idea comes from recognizing that all but a constant number of the differences between adjacent columns in this matrix are unchanged between a cluster

[6] This result is relatively unknown. It is mentioned briefly in a paper whose main result is a Las Vegas style randomized algorithm for connectivity with $\mathcal{O}(\log n \cdot \log^3 \log n)$ expected amortized update time.

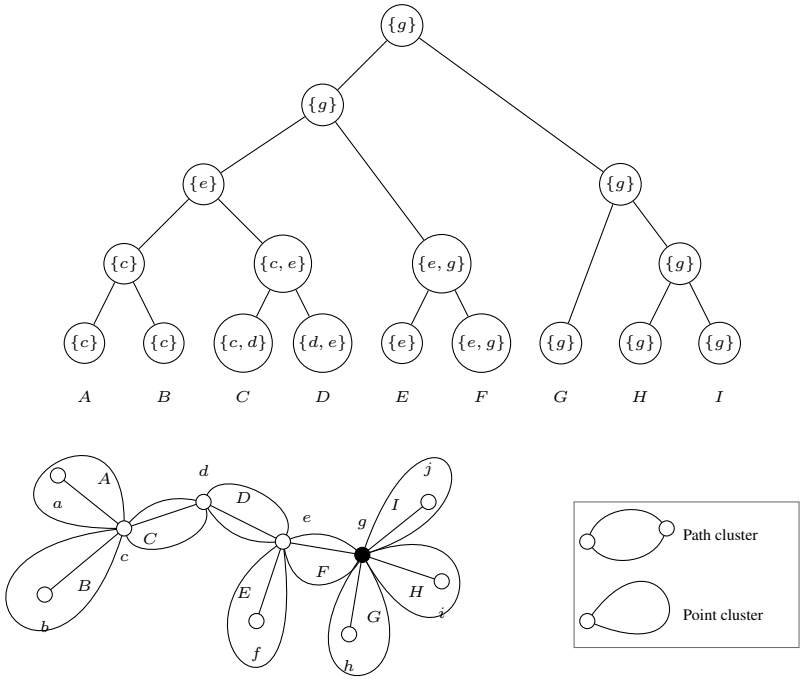


Figure 3: A top tree and its underlying tree. A *cluster* is a connected subtree with at most two *boundary vertices*. Each node in the top tree corresponds to a cluster with the listed boundary vertices. Vertex g is chosen as *external boundary vertex*.

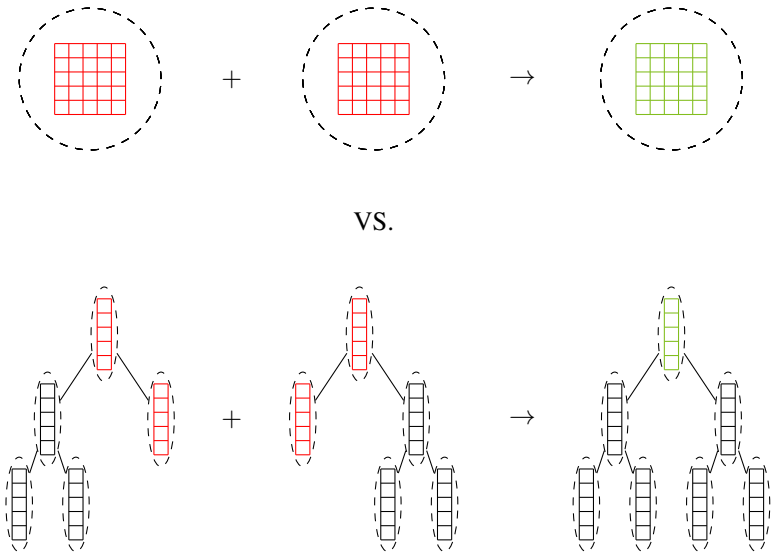


Figure 4: In each MERGE, the old algorithm had to recompute the whole matrix. The new one inherits most of the vectors from the previous nodes.

and its parent. Maintaining the column difference vectors in a suitable balanced tree then lets us compute any entry of the original matrix in $\mathcal{O}(\log \log n)$ time as a simple prefix sum when needed, and each MERGE requires only $\mathcal{O}(\log \log n)$ new difference vectors to be computed, essentially as a sum of two existing vectors (See Figure 4). This is enough to give a factor $\Theta(\log n / \log \log n)$ improvement in amortized update time over [HLT01], and is of independent interest because it achieves the same $\mathcal{O}(\log^3 n \cdot \log \log n)$ update time as [Tho00] without the use of bit-tricks.

The actual "bit-tricks" used in [Tho00] are based on the fact that we don't always[7] need the exact subtree sizes, which require $\mathcal{O}(\log n)$ bits each, but can make do with an $(1 + \frac{O(1)}{\log n})$-approximate lower bound, which can be stored as a $\mathcal{O}(\log \log n)$ bit floating point value per subtree size. That means we can pack $\Omega(\log n / \log \log n)$ of these approximate subtree sizes in a single word of size $w = \Omega(\log n)$ and can e.g. add them in parallel in constant time. This gives an immediate speedup by a factor of $\Omega(\log n / \log \log n)$ for most operations, but in one case we need to compute the *diagonal* of the matrix, which would still naively take $\Theta(\log n \cdot \log \log n)$ time. We fix this by letting the prefix sum tree also maintain a *filtered* prefix sum, where all entries on the wrong side of the diagonal are counted as zero. With this, every MERGE or SPLIT in the top tree takes worst case $\mathcal{O}(\log^2 \log n)$ time, and each **INSERT-EDGE** or **DELETE-EDGE** causes amortized $\mathcal{O}(\log^2 n)$ MERGEs and SPLITs, giving the claimed $\mathcal{O}(\log^2 n \cdot \log^2 \log n)$ amortized time per update.

[7] Specifically, we only need the exact sizes to answer **FIND-SIZE** and **FIND-2-SIZE** queries, and for this it is sufficient with exact sizes for cover levels -1 and 0 respectively.

The *natural* query time using this structure is $\mathcal{O}(\log n \cdot \log^2 \log n)$, which is also the time we end up with for the **FIND-2-SIZE** query[8]. The remaining queries can be sped up by maintaining a separate $\mathcal{O}(\log^\varepsilon n)$-nary top tree on the side (for some small $\varepsilon > 0$). We show for the first time how to maintain such a wide top tree of height $\mathcal{O}(\log n/(\varepsilon \log \log n))$. Using bit-tricks to work on information from all children in parallel, and avoiding the use of EXPOSE, we can use this to answer **CONNECTED**, **2-EDGE-CONNECTED**, **FIND-BRIDGE**, and **FIND-SIZE** queries in worst case $\mathcal{O}(\log n/ \log \log n)$ time, without affecting the amortized update time.

The structure as described uses $\mathcal{O}(m + n \log^2 \log n)$ space. To reduce this, we show a somewhat general trick with top trees, consisting of only storing the full data for clusters that are larger than some value $q(n)$. In our case $q(n) = \Omega(\log^2 \log n)$. There are at most $O(n/q(n))$ of these clusters, so the total space becomes linear, and we show that $q(n)$ is small enough that we can compute the information as needed without increasing the total running time.

5.5 Future work

There are gaps everywhere in our understanding of fully dynamic k-edge-connectivity for all k. Even for $k = 1$, the fine-grained complexity of the problem is not well understood. E.g. is randomization necessary to get $\mathcal{O}(\log^{2-\epsilon} n)$ amortized update time for connectivity? Or to get $\mathcal{O}(n^{o(1)})$ (expected) worst case time for connectivity? Is it possible to find an algorithm that (expected) behaves well both worst case and amortized, e.g. has $\mathcal{O}(n^{1-\epsilon})$ worst case per operation, but still $\mathcal{O}(\mathrm{poly}(\log n))$ amortized?

For $k > 2$, no algorithm for k-edge connectivity with polylogarithmic time per operation is known. Would it be possible to achieve e.g. $\mathcal{O}(\mathrm{poly}(\log^k n))$ for fixed k?

[8] It seems likely that this can be improved to $\mathcal{O}(\log n)$ by implementing **FIND-2-SIZE** without using EXPOSE, similarly to what we do for the other queries.

6 One-Way Trail Orientations (Synopsis of Appendix D)

Appendix D Anders Aamand, Niklas Hjuler, Jacob Holm, and Eva Rotenberg. "One-Way Trail Orientations". In: *ArXiv e-prints* (Aug. 2017). **Extended version of [Aam+18]**. arXiv: 1708.07389 [cs.DS]

6.1 Problem

TRAIL-ORIENTABLE$(G, \mathcal{P})$: Given an undirected or mixed graph $G = (V, E)$, and a partition $\mathcal{P}$ of the undirected edges of E into *trails*, does there exist an orientation of the undirected edges of E, such that each trail is oriented consistently, and the whole graph is strongly connected?

TRAIL-ORIENTATION$(G, \mathcal{P})$: Given an undirected or mixed graph $G = (V, E)$, and a partition $\mathcal{P}$ of the undirected edges of E into *trails*, either return a strong orientation of G such that each trail is oriented consistently, or **nil** if no such strong orientation exists.

6.2 Known results

When each trail in $\mathcal{P}$ consists of a single edge, the problem is well understood. For an undirected graph G, Robbins Theorem [Rob39] states that **TRAIL-ORIENTABLE$(G, \mathcal{P})$** is true if and only if G is connected and bridgeless. For mixed graphs, the Generalized Robbins Theorem from [BT80] similarly states that **TRAIL-ORIENTABLE$(G, \mathcal{P})$** is true if and only if G is strongly connected and the graph obtained by making all the edges undirected is bridgeless. In either case, simple DFS-based algorithms by Tarjan et al. [HT73] and Chung et al. [CGT] can compute **TRAIL-ORIENTATION$(G, \mathcal{P})$** in linear time.

The more general version, where each trail in $\mathcal{P}$ may consist of more than one edge, was posed as an open problem by Tarjan at the Erice summer school on "Graph Theory, Algorithms and Applications" in May 2017.

6.3 Our result

For undirected $G = (V, E)$ and any partition $\mathcal{P}$ of E into trails, we show that **TRAIL-ORIENTABLE$(G, \mathcal{P})$** if and only if G is connected and bridgeless, and we give a linear-time algorithm for computing **TRAIL-ORIENTATION$(G, \mathcal{P})$**.

For $G = (V, E)$ strongly connected and mixed we show that

1. if G has an undirected strong bridge e and $T \in \mathcal{P}$ is the trail containing e, then **TRAIL-ORIENTABLE$(G, \mathcal{P})$** if and only if there is an orientation of T such that the resulting graph G' is strongly connected, and **TRAIL-ORIENTABLE$(G', \mathcal{P} \setminus \{T\})$**.

2. if G has no undirected strong bridges, then for any partition $\mathcal{P}$ and any trail $T \in \mathcal{P}$, any orientation of T can be extended to a strong orientation of G.

An immediate corollary to this is that the obvious naive greedy algorithm (See Algorithm 1) works. In the paper we just mention that it runs in polynomial time.

6.4 Subsequent improvement, made for this thesis

In fact, we can make Algorithm 1 run in $\mathcal{O}(pm)$ time where $p = |\mathcal{P}| \leq m$ is the number of trails in the partition. For this, we can use e.g. the algorithm by Italiano et al. [ILS12] to find all strong bridges in line 5, and e.g. Tarjan's algorithm from [Tar72] to find the strongly connected components in line 7. This makes each of the at most p iterations of the algorithm run in $\mathcal{O}(m)$ time. We can improve this to $\mathcal{O}(\min\{p, n\}m)$ time by observing that:

1. Each time we reach line 8, the number of connected components is increased in the graph $G_u = (V, E_u)$ where E_u is the set of undirected edges in G. Thus, line 6 is reached at most $\min\{p, n-1\}$ times.

2. If we ever reach line 13, we know that *any* partition of the undirected edges into trails admit a strong orientation. In particular, we can merge any pair of trails that share a common end vertex and still preserve this property. Doing so until there are no such trail pairs left reduces the total number of remaining trails to at most $\min\{p, n/2\}$. It is possible to do all these merges in $\mathcal{O}(m)$ time.

With these modifications, the main loop of Algorithm 1 runs at most $\mathcal{O}(\min\{p, n\})$ iterations, and each iteration takes only $\mathcal{O}(m)$ time, thus the total time becomes $\mathcal{O}(\min\{p, n\}m)$ as claimed.

6.5 Techniques

For the undirected version of our new theorem, we consider an edge that is at the end of its trail. Either this edge can be deleted without creating a bridge, in which case it follows by induction that the remaining graph has a strong trail-orientation and that this orientation extends to a trail-orientation of G; or the edge is part of a 2-edge cut, and we can construct two smaller graphs which, again by induction, have a trail-orientation that can be extended to a trail-orientation of G.

The linear-time algorithm takes this idea a bit further. We first transform the graph to have maximum degree 3. Then we show that if we remove a maximal set of edges that are at the ends of their trails, such that the graph remains connected and bridgeless, then the resulting graph has a linear number of 2-edge cuts. We can then construct a new graph for each 3-edge connected component, and show that given a trail-orientation for each of these new graphs, we can construct a trail-orientation for G. Since there are a linear number of 2-edge cuts, at most a constant fraction of the 3-edge connected components have size greater than some small constant. Thus, as long as we can find the edges to delete, construct the new graphs for the 3-edge connected components, and combine the results, all in linear time, then the

total running time is also linear. Finally we show that existing techniques can be combined to do exactly that.

For the mixed version, we assume G has no undirected strong bridges and consider any trail $T \in \mathcal{P}$ and some chosen orientation of T. The proof is by induction on $|\mathcal{P}|$. If the graph G' resulting from this orientation still has no undirected strong bridges, then by induction $(G', \mathcal{P} \setminus \{T\})$ has a strong trail orientation, which is clearly also the required orientation of G. Otherwise G' has an undirected strong bridge e, and we show that given $(G, \mathcal{P})$ and e we can construct instances $(G_1, \mathcal{P}_1)$ and $(G_2, \mathcal{P}_2)$, with $|\mathcal{P}_1|, |\mathcal{P}_2| < |\mathcal{P}|$, such that G_1 and G_2 have no undirected strong bridges and each chosen orientation of T enforces at most one orientation in each. Since these graphs are smaller, it follows by induction that they have the required strong orientation, and by construction these orientations give the required strong orientation of $(G, \mathcal{P})$.

6.6 Future Work

The problem seems quite well understood for undirected graphs now, but for mixed graphs there are still some open questions, both graph theoretically and algorithmically. On the graph theory side, it would be interesting to find some simpler criteria for when a mixed graph with undirected strong bridges has a strong orientation. Algorithmically, it would be interesting to beat the $\mathcal{O}(\min\{p, n\}m)$ time algorithm presented in this thesis.

An obvious idea would be to try to extend the linear-time algorithm for undirected graphs. It turns out there exists a similar transformation to cubic graphs that works for mixed graphs. The main problem with this idea is to find a large subset S of directed edges and trail-ends such that $G - S$ is strongly connected and is easily partitioned into subproblems whose total size is at most a constant fraction of the original size.

7 Concluding remarks

In this thesis we have presented several results on efficient graph algorithms and data structures. While these seem important in and of themselves, the "spin-offs" might be equally interesting. By that I mean things like the st-decomposition from Appendix A, the server necessities from Appendix B, the wide or fat-bottomed top trees from Appendix C, or the "delete as much as possible while staying 2-edge connected, then consider the 3-edge connected components" idea from Appendix D.

All of these seem like interesting tools, and I can't wait to see what they will be used for in the future.

Bibliography

[Aam+17] Anders Aamand, Niklas Hjuler, Jacob Holm, and Eva Rotenberg. "One-Way Trail Orientations". In: *ArXiv e-prints* (Aug. 2017). **Extended version of [Aam+18]**. arXiv: 1708.07389 [cs.DS].

[Aam+18] Anders Aamand, Niklas Hjuler, Jacob Holm, and Eva Rotenberg. "One-Way Trail Orientations". In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. July 2018, 6:1–6:13. DOI: 10.4230/LIPIcs.ICALP.2018.6.

[Abr+17a] Mikkel Abrahamsen, Stephen Alstrup, Jacob Holm, Mathias Bæk Tejs Knudsen, and Morten Stöckel. "Near-Optimal Induced Universal Graphs for Bounded Degree Graphs". In: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. **Note that my name is incorrectly listed as "Stephen Holm" in the proceedings.** 2017. ISBN: 978-3-95977-041-5. DOI: 10.4230/LIPIcs.ICALP.2017.128.

[Abr+17b] Mikkel Abrahamsen, Stephen Alstrup, Jacob Holm, Mathias Bæk Tejs Knudsen, and Morten Stöckel. "Near-Optimal Induced Universal Graphs for Bounded Degree Graphs". In: *CoRR* abs/1607.04911 (2017). **Split in two for publication. One part published as [Abr+17a], other part submitted to Discrete Applied Mathematics**. arXiv: 1607.04911.

[Abr+17c] Mikkel Abrahamsen, Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. "Best Laid Plans of Lions and Men". In: *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*. Ed. by Boris Aronov and Matthew J. Katz. Vol. 77. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 6:1–6:16. DOI: 10.4230/LIPIcs.SoCG.2017.6.

[Abr+17d] Mikkel Abrahamsen, Jacob Holm, Eva Rotenberg, and Christian Wulff-Nilsen. "Best Laid Plans of Lions and Men". In: *CoRR* abs/1703.03687 (2017). arXiv: 1703.03687.

[AH00] Stephen Alstrup and Jacob Holm. "Improved Algorithms for Finding Level Ancestors in Dynamic Trees". In: *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*. Ed. by Ugo Montanari, José D. P. Rolim, and Emo Welzl. Vol. 1853. Lecture Notes in Computer Science. Springer, 2000, pp. 73–84. ISBN: 3-540-67715-1. DOI: 10.1007/3-540-45022-X_8.

[Als+05] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Maintaining information in fully dynamic trees with top trees". In: *ACM Trans. Algorithms* 1.2 (2005), pp. 243–264. DOI: 10.1145/1103963.1103966.

[BHR17] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. "Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements". In: *ArXiv e-prints* (July 2017). **Extended version of [BHR18], submitted to JACM**. arXiv: 1707.06063 [cs.DS].

[BHR18] Aaron Bernstein, Jacob Holm, and Eva Rotenberg. "Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Ed. by Artur Czumaj. **SODA 2018 best paper**. SIAM, 2018, pp. 947–959. DOI: 10.1137/1.9781611975031.61.

[Bha+12] Deepak Bhadauria, Kyle Klein, Volkan Isler, and Subhash Suri. "Capturing an evader in polygonal environments with obstacles: The full visibility case". In: *I. J. Robotics Res.* 31.10 (2012), pp. 1176–1189. DOI: 10.1177/0278364912452894.

[BT80] Frank Boesch and Ralph Tindell. "Robbins's Theorem for Mixed Multigraphs". In: *The American Mathematical Monthly* 87.9 (1980), pp. 716–719. DOI: 10.1080/00029890.1980.11995131.

[Bos+14] Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych. "Online Bipartite Matching in Offline Time". In: *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*. IEEE Computer Society, 2014, pp. 384–393. ISBN: 978-1-4799-6517-5. DOI: 10.1109/FOCS.2014.48.

[Bos+18] Bartlomiej Bosek, Dariusz Leniowski, Piotr Sankowski, and Anna Zych-Pawlewicz. "Shortest Augmenting Paths for Online Matchings on Trees". In: *Theory Comput. Syst.* 62.2 (2018), pp. 337–348. DOI: 10.1007/s00224-017-9838-x.

[Bos+17] Bartlomiej Bosek, Dariusz Leniowski, Anna Zych, and Piotr Sankowski. "The Shortest Augmenting Paths for Online Matchings on Trees". In: *CoRR* abs/1704.02093 (2017). arXiv: 1704.02093. URL: http://arxiv.org/abs/1704.02093.

[Cha+09] Kamalika Chaudhuri, Constantinos Daskalakis, Robert D. Kleinberg, and Henry Lin. "Online Bipartite Perfect Matching With Augmentations". In: *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro,*

Brazil. IEEE, 2009, pp. 1044–1052. ISBN: 978-1-4244-3513-5. DOI: 10.1109/INFCOM.2009.5062016.

[CGT] Fan R. K. Chung, Michael Randolph Garey, and Robert Endre Tarjan. "Strongly connected orientations of mixed multigraphs". In: *Networks* 15.4 (), pp. 477–484. DOI: 10.1002/net.3230150409.

[Epp+97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. "Sparsification - a technique for speeding up dynamic graph algorithms". In: *J. ACM* 44.5 (1997), pp. 669–696. DOI: 10.1145/265910.265914.

[Fre97] Greg N. Frederickson. "Ambivalent Data Structures for Dynamic 2-Edge-Connectivity and k Smallest Spanning Trees". In: *SIAM J. Comput.* 26.2 (1997), pp. 484–538. DOI: 10.1137/S0097539792226825.

[GI92] Zvi Galil and Giuseppe F. Italiano. "Fully Dynamic Algorithms for 2-Edge Connectivity". In: *SIAM J. Comput.* 21.6 (1992), pp. 1047–1069. DOI: 10.1137/0221062.

[Gib+15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. "Dynamic graph connectivity with improved worst case update time and sublinear space". In: *CoRR* abs/1509.06464 (2015). arXiv: 1509.06464. URL: http://arxiv.org/abs/1509.06464.

[Gro+95] Edward F. Grove, Ming-Yang Kao, P. Krishnan, and Jeffrey Scott Vitter. "Online Perfect Matching and Mobile Computing". In: *Algorithms and Data Structures, 4th International Workshop, WADS '95, Kingston, Ontario, Canada, August 16-18, 1995, Proceedings*. Ed. by Selim G. Akl, Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro. Vol. 955. Lecture Notes in Computer Science. Springer, 1995, pp. 194–205. ISBN: 3-540-60220-8. DOI: 10.1007/3-540-60220-8_62.

[HK95] Monika Rauch Henzinger and Valerie King. "Randomized dynamic graph algorithms with polylogarithmic time per operation". In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*. Ed. by Frank Thomson Leighton and Allan Borodin. ACM, 1995, pp. 519–527. ISBN: 0-89791-718-9. DOI: 10.1145/225058.225269.

[Hol+18] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Eva Rotenberg. "Decremental SPQR-trees for Planar Graphs". In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Ed. by Yossi Azar, Hannah Bast, and Grzegorz Herman. Vol. 112. Leibniz International Proceedings in Informatics (LIPIcs). **ESA 2018 best paper**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 46:1–46:16. ISBN: 978-3-95977-081-1. DOI: 10.4230/LIPIcs.ESA.2018.46.

[Hol+17] Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki, Eva Rotenberg, and Piotr Sankowski. "Contracting a Planar Graph Efficiently". In: *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 50:1–50:15. DOI: 10.4230/LIPIcs.ESA.2017.50.

[HLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *J. ACM* 48.4 (2001), pp. 723–760. DOI: 10.1145/502090.502095.

[HR17] Jacob Holm and Eva Rotenberg. "Dynamic Planar Embeddings of Dynamic Graphs". In: *Theory Comput. Syst.* 61.4 (2017). **Announced at STACS'15**, pp. 1054–1083. DOI: 10.1007/s00224-017-9768-7.

[HR18] Jacob Holm and Eva Rotenberg. "Good r-divisions Imply Optimal Amortised Decremental Biconnectivity". In: *CoRR* abs/1808.02568 (Aug. 2018). **Submitted to SODA 2019**. arXiv: 1808.02568.

[HRT14] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Planar Reachability in Linear Space and Constant Time". In: *ArXiv e-prints* (Nov. 2014). **Extended version of [HRT15]**. arXiv: 1411.5867 [cs.DS].

[HRT15] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Planar Reachability in Linear Space and Constant Time". In: *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*. Ed. by Venkatesan Guruswami. IEEE Computer Society, 2015, pp. 370–389. DOI: 10.1109/FOCS.2015.30.

[HRT17] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time". In: *ArXiv e-prints* (July 2017). **Extended version of [HRT18]**. arXiv: 1707.06311 [cs.DS].

[HRT18] Jacob Holm, Eva Rotenberg, and Mikkel Thorup. "Dynamic Bridge-Finding in $\widetilde{\mathcal{O}}(\log^2 n)$ Amortized Time". In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. Ed. by Artur Czumaj. SIAM, Jan. 2018, pp. 35–52. DOI: 10.1137/1.9781611975031.3.

[HK73] John Edward Hopcroft and Richard Manning Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM J. Comput.* 2.4 (1973), pp. 225–231. DOI: 10.1137/0202019.

[HT73] John Hopcroft and Robert Endre Tarjan. “Algorithm 447: Efficient Algorithms for Graph Manipulation”. In: *Commun. ACM* 16.6 (June 1973), pp. 372–378. ISSN: 0001-0782. DOI: 10.1145/362248.362272.

[How08] Rodney R. Howell. *On Asymptotic Notation with Multiple Variables*. Tech. rep. 2007-4. Kansas State University, Jan. 2008. URL: http://people.cs.ksu.edu/~rhowell/asymptotic.pdf.

[Hua+17] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. “Fully Dynamic Connectivity in $O(\log n(\log\log n)^2)$ Amortized Expected Time”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. Ed. by Philip N. Klein. SIAM, 2017, pp. 510–520. ISBN: 978-1-61197-478-2. DOI: 10.1137/1.9781611974782.32.

[ILS12] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. “Finding strong bridges and strong articulation points in linear time”. In: *Theor. Comput. Sci.* 447 (2012), pp. 74–84. DOI: 10.1016/j.tcs.2011.11.011.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. “Dynamic graph connectivity in polylogarithmic worst case time”. In: *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. Ed. by Sanjeev Khanna. SIAM, 2013, pp. 1131–1142. ISBN: 978-1-61197-251-1. DOI: 10.1137/1.9781611973105.81.

[Kej+16] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. “Faster Worst Case Deterministic Dynamic Connectivity”. In: *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*. Ed. by Piotr Sankowski and Christos D. Zaroliagis. Vol. 57. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 53:1–53:15. ISBN: 978-3-95977-015-6. DOI: 10.4230/LIPIcs.ESA.2016.53.

[Kha13] Sanjeev Khanna, ed. *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. SIAM, 2013. ISBN: 978-1-61197-251-1. DOI: 10.1137/1.9781611973105.

[Men27] Karl Menger. “Zur allgemeinen Kurventheorie”. In: *Fundamenta Mathematicae* 10 (1927).

[NSW17] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. “Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time”. In: *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, Octo-*

ber 15-17, 2017. Ed. by Chris Umans. IEEE Computer Society, 2017, pp. 950–961. ISBN: 978-1-5386-3464-6. DOI: 10.1109/FOCS.2017.92.

[P a11] Mihai Pătraşcu. "Unifying the Landscape of Cell-Probe Lower Bounds". In: *SIAM J. Comput.* 40.3 (2011), pp. 827–847. DOI: 10.1137/09075336X.

[PD06] Mihai Pătraşcu and Erik D. Demaine. "Logarithmic Lower Bounds in the Cell-Probe Model". In: *SIAM J. Comput.* 35.4 (2006), pp. 932–963. DOI: 10.1137/S0097539705447256.

[Rob39] Herbert Ellis Robbins. "A Theorem on Graphs, with an Application to a Problem of Traffic Control". In: *The American Mathematical Monthly* 46.5 (1939), pp. 281–283. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2303897.

[TT93] Roberto Tamassia and Ioannis G. Tollis. "Dynamic Reachability in Planar Digraphs with One Source and One Sink". In: *Theor. Comput. Sci.* 119.2 (1993), pp. 331–343. DOI: 10.1016/0304-3975(93)90164-O.

[Tar72] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[Tar74] Robert Endre Tarjan. "A Note on Finding the Bridges of a Graph". In: *Inf. Process. Lett.* 2.6 (1974), pp. 160–161. DOI: 10.1016/0020-0190(74)90003-9.

[Tho00] Mikkel Thorup. "Near-optimal fully-dynamic graph connectivity". In: *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*. Ed. by F. Frances Yao and Eugene M. Luks. ACM, 2000, pp. 343–350. ISBN: 1-58113-184-4. DOI: 10.1145/335305.335345.

[Tho04] Mikkel Thorup. "Compact oracles for reachability and approximate distances in planar digraphs". In: *J. ACM* 51.6 (2004), pp. 993–1024. DOI: 10.1145/1039488.1039493.

[TZ05] Mikkel Thorup and Uri Zwick. "Approximate distance oracles". In: *J. ACM* 52.1 (2005), pp. 1–24. DOI: 10.1145/1044731.1044732.

[Wul13] Christian Wulff-Nilsen. "Faster Deterministic Fully-Dynamic Graph Connectivity". In: *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*. Ed. by Sanjeev Khanna. SIAM, 2013, pp. 1757–1769. ISBN: 978-1-61197-251-1. DOI: 10.1137/1.9781611973105.126.

Part II
Appendix

Planar Reachability in Linear Space and Constant Time

Jacob Holm and Eva Rotenberg and Mikkel Thorup*†

University of Copenhagen (DIKU),
jaho@di.ku.dk, roden@di.ku.dk, mthorup@di.ku.dk

April 7, 2015

Abstract

We show how to represent a planar digraph in linear space so that reachability queries can be answered in constant time. The data structure can be constructed in linear time. This representation of reachability is thus optimal in both time and space, and has optimal construction time. The previous best solution used $O(n \log n)$ space for constant query time [Thorup FOCS'01].

*Research partly supported by Thorup's Advanced Grant from the Danish Council for Independent Research under the Sapere Aude research career programme.

†Research partly supported by the FNU project AlgoDisc - Discrete Mathematics, Algorithms, and Data Structures.

1 Introduction

Representing reachability of a directed graph is a fundamental challenge. We want to represent a digraph $G = (V, E)$, $n = |V|$, $m = |E|$, so that we for any vertices u and w can tell if u reaches v, that is, if there is a dipath from u to v. There are two extreme solutions: one is to just store the graph, as is, using $O(m)$ words of space and answering reachability queries from scratch, e.g., using breadth-first-search, in $O(m)$ time. The other is to store a reachability matrix using n^2 bits and then answer reachability queries in constant time. Thorup and Zwick [20] proved that there are graphs classes such that any representation of reachability needs $\Omega(m)$ bits. Also, Pătraşcu [16] has proved that there are directed graphs with $O(n)$ edges where constant time reachability queries require $n^{1+\Omega(1)}$ space. Thus, for constant time reachability queries to a general digraph, all we know is that the worst-case space is somewhere between $\Omega(m + n^{1+\Omega(1)})$ and n^2 bits.

The situation is in stark contrast to the situation for undirected/symmetric graphs where we can trivially represent reachability queries on $O(n)$ space and constant time, simply by enumerating the connected components, and storing with each vertex the number of the component it belongs to. Then u reaches v if and only if the have the same component number.

In this paper we focus on the planar case, which feels particularly relevant when you live on a sphere. For planar digraphs it is already known that we can do much better than for general digraphs. Back in 2001, Thorup [19] presented a reachability oracle for planar digraphs using $O(n \lg n)$ space for constant query time, or linear space for $O(\log n)$ query time. In this paper, we present the first improvement; namely an $O(n)$ space reachability oracle that can answer reachability queries in constant time. Note that this bound is asymptotically optimal; even to distinguish between the subclass of directed paths of length n, we need $\Omega(n \log n)$ bits. Our oracle is constructed in linear time.

Computational model The computational model for all upper bounds is the word RAM, modelling what we can program in a standard programming language such as C [12]. A word is a unit of space big enough to fit any vertex identifier, so a word has $w \geq \lg n$ bits, and word operations take constant time. Here $\lg = \log_2$. In our upper bounds, we limit ourselves to the *practical RAM* model [14], which is a restriction of the word RAM to the standard operations on words available in C that are AC0. This includes indexing arrays as needed just to store a reachability matrix with constant time access, but excludes e.g. multiplication and division. Thus, unless otherwise specified, we measure *space* as the number of words used and *time* as the number of word operations performed.

The $\Omega(m + n^{1+\Omega(1)})$ space lower bound from [16] for general graphs is in the cell-probe model subsuming the word RAM with an arbitrary instruction set.

Other related work Before [19], the best reachability oracles for general planar digraphs were distance oracles, telling not just if u reaches w, but if so, also the length of the shortest dipath from u to w [3–5]. For such planar distance oracles, the best current time-space trade-off is $\tilde{O}(n/\sqrt{s})$ time for any $s \in [n, n^2]$ [15].

The construction of [19] also yields approximate distance oracles for planar digraphs. With edge weights from $[N]$, $N \leq 2^w$, distance queries where answered within a factor $(1 + \epsilon)$ in $O(\log\log(Nn) + 1/\varepsilon)$ time using $O(n(\log n)(\log(Nn))/\varepsilon)$ space. These bounds have not been improved.

For the simpler case of undirected graphs, where reachability is trivial, [13, 19] provides a more efficient $(1 + \varepsilon)$-approximate distance queries for planar graphs in $O(1/\varepsilon)$ time and $O(n(\log n)/\varepsilon)$ space. In [10] it was shown that the space can be improved to linear if the query time is increased to $O((\log n)^2/\varepsilon^2)$. In [11] it was shown how to represent planar graphs with bounded weights using $O(n \log^2((\log n)/\varepsilon) \log^*(n) \log\log(1/\varepsilon))$ space and answering $(1 + \varepsilon)$ approximate distance queries in $O((1/\varepsilon)\log(1/\varepsilon)\log\log(1/\varepsilon)\log^*(n) + \log\log\log n))$ time. Using $\tilde{O}$ to suppress factors of $O(\log\log n)$ and $O(\log(1/\varepsilon))$, these bounds reduce

to $\bar{O}(n)$ space and $\bar{O}(1/\varepsilon)$ time. This improvement is similar in spirit to our improvement for reachability in planar digraphs. However, the techniques are entirely different.

There has also been work on special classes of planar digraphs. In particular, for a planar s-t-graph, where all vertices are on dipaths between s and t, Tamassia and Tollis [17] have shown that we can represent reachability in linear space, answering reachability queries in constant time. Also, [4,6,7] presents improved bounds for planar exact distance oracles when all the vertices are on the boundary of a small set of faces.

Techniques We will develop our linear space constant query time reachability oracles by considering more and more complex classes of planar digraphs. We make reductions from $i+1$ to i in the following:

1. Acyclic planar s-t-graph; $\exists(s,t)$, such that all vertices are reachable from s and may reach t. [17]
2. Acyclic planar single-source graph; $\exists s$, such that all vertices are reachable from s. See Section 3.
3. Acyclic planar In-Out graph; $\exists s$ such that all vertices with out-degree 0 are reachable from s. See Section 4
4. Any acyclic planar graph. The reduction to acyclic planar In-Out graphs from general acyclic planar graphs is known. [19]
5. Any planar graph. The reduction to acyclic planar graphs is well-known. Using the depth first search algorithm by Tarjan [18], we can contract each strongly connected component to get an acyclic planar graph. Vertices in the same strongly connected component can always reach each other, and vertices in distinct strongly connected components can reach each other if the corresponding vertices in the contracted graph can.

The most technically involved step is the reduction from single-source graph to s-t-graph. As in [19], we use separators to form a tree over a partitioning of the vertices of the graph. However, in [19], the *alternation number*; the number of directed segments in the frame that separates a child from its parent (see Section 2), needs only be a constant number. In contrast, it is a crucial part of our construction that the alternation number, which must be even, is at most 4. Also, in our data structure, paths cannot go upward in the rooted tree, whereas there is no such restriction in [19]. These two features let us use a level ancestor -like algorithm to quickly calculate the best ≤ 4 vertices in a given tree-node that can reach a given vertex v. Each component is an s-t-graph, and v can be reached by some u in the ancestral component if and only if u can reach at least one of these best ≤ 4 vertices.

2 Preliminaries

For a vertex v at depth d in a rooted forest T and an integer $0 \leq i \leq d$, the i'th *level ancestor* of v in T is the ancestor to v in T at depth i. For two nodes x, y in a rooted tree, let $x \preceq y$ denote that x is an ancestor to y, and $x \prec y$ that x is a proper ancestor to y.

We say a graph is *plane*, if it is embedded in the plane, and denote by π_v the permutation of edges around v. Given a plane graph, (G,π), we may introduce *corners* to describe the incidence of a vertex to a face. A vertex of degree n has n corners, where if $\pi_v((v,u)) = (v,w)$, and the face f is incident to (v,u) and (v,w), then there is a corner of f incident to v between (v,u) and (v,w). We denote by $V[X]$ and $E[X]$ the vertices and edges, of some (not necessarily induced) subgraph X. Given a subgraph H of a planar embedded graph G, the faces of H define *superfaces* of those of G, and the faces of G are *subfaces* of those of H. Similarly for corners. Note that the faces of H correspond to the connected components of $G^* \setminus H$. The super-corners incident to v correspond to a set of consecutive corners in the ordering around v.

In a directed graph, we may consider the boundary of a face in some subgraph, H. A corner of a face f of H is a *target* for f if it lies between ingoing edges (u, v) and (w, v), and *source* if it lies between outgoing edges (v, u) and (v, w). We say the face boundary has *alternation number* $2a$ if it has a source and a target corners. When a face boundary has alternation number $2a$, we say it consists of $2a$ *disegments* (*di*rected segments), associated with the directed paths from source to target. We associate with each disegment the total ordering stemming from reachability of vertices on the path via the path, and by convention we set $\mathrm{succ}(t, S) = \perp$ for a target vertex t on the disegment. Given a set of edges $S \subset E$, we denote by $\mathrm{init}(S)$ the set of inital vertices, $\mathrm{init}(S) = \{u | (u, v) \in S\}$. Given a connected planar graph with a spanning tree T, the edges $T^* := E \setminus T$ form a spanning tree for the dual graph. We call the pair (T, T^*) a tree-cotree decomposition of the graph, referring to T and T^* as tree and cotree.

When u can reach v we write $u \rightsquigarrow v$. An s-t-graph is a graph with special vertices s, t such that $s \rightsquigarrow v$ and $v \rightsquigarrow t$ for all vertices v. We say a graph is a *truncated s-t-graph* if it is possible to add vertices s, t to obtain an s-t-graph, without violating the embedding. In an s-t-graph, all faces has alternation number 2.

3 Acyclic planar single-source digraph

Given a global source vertex s for the planar digraph, we wish to make a data structure for reachability queries. We do this by reduction to the s-t-case. A tree-like structure with truncated s-t-graphs as nodes is obtained by recursively choosing a face f wisely, and then letting vertices that can reach vertices on f belong to this node, and partitioning all other vertices among the descendants of this node. As we shall see in Section 3.1, this can be done in such a way that we obtain logarithmic height and such that the border between a node and its ancestors is a cycle of alternation number at most 4. We call this the *frame* of the node.

We always choose the truncated s-t-graph maximally, such that once a path crosses a frame, it does not exit the frame again. Thus, for u to reach v, u has to lie in a component which is ancestral to that of v, and since the alternation number of any frame between those two component is at most 4, the path could always be chosen to use one of the at most 4 different "best" vertices for reaching v on that frame. Thus, the idea is to do something inspired by level ancestry to find those "best" vertices in u's component. We handle the case of frames with alternation number 2 in Section 3.3. Frames with alternation number 4 are similar but more involved, and the details are found in Section 3.4.

Definition 3.1. Given a graph $G = (V, E)$, a subgraph $G' = (V', E')$ is *backward closed* if $\forall (u, v) \in E : v \in V' \implies (u, v) \in E'$.

Definition 3.2. The *backward closure* of a face f, denoted $\mathrm{bc}(f)$ is the unique smallest backward closed graph that contains all the vertices incident to f.

Definition 3.3. Let $G = (V, E)$ be an acyclic single-source plane digraph, and let $G^* = (V^*, E^*)$ be its dual. An *s-t-decomposition* of G is a rooted tree where each node x is associated with a face $f_x \in V^*$ and subgraphs $G^*_x \subseteq G^*$ and $C_x \subseteq S_x \subseteq G$ such that:

- f_x is unique ($f_x \neq f_y$ for $x \neq y$).
- S_x is $\mathrm{bc}(f_x)$ if x is the root, and $\mathrm{bc}(f_x) \cup S_y$ if x is a child of y.
- C_x is $\mathrm{bc}(f_x)$ if x is the root, and $\mathrm{bc}(f_x) \setminus S_y$ if x is a child of y.

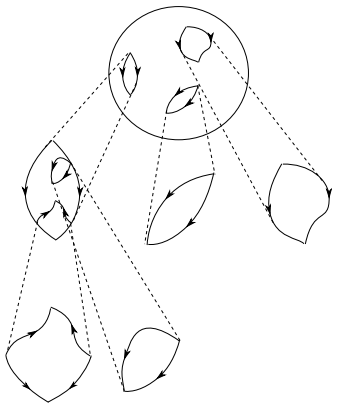


Figure 1: A tree of truncated s-t-graphs, each child contained in a face-cycle of its parent.

- G^*_x is the subgraph of G^* induced by $\{f_z |\ z \text{ is a descendent of } x\}$. Furthermore, if x is a child of y we require that G^*_x is the connected component of $G^* \setminus E^*[S_y]$ containing f_x.

If x is a child of y, x has a *parent frame* $F_x \subseteq S_y$ and a set of *down-edges* $E_x \subseteq E$ such that:

- F_x is the face cycle in S_y that corresponds to G^*_x.
- E_x is the set of edges (w, w') such that $w \in V[F_x]$ and $w' \in V[C_z]$ for some descendant z of x.

An s-t-decomposition is *good* if the tree has height $\mathcal{O}(\log n)$ and each frame has alternation number 2 or 4.

The name s-t-decomposition is chosen based on the following

Lemma 3.4. Each vertex of G is in exactly one C_x, and each C_x is a truncated s-t-graph.

Proof. If x is the root, $C_x = \mathrm{bc}(f_x)$ and this is clearly a truncted s-t-graph. Otherwise let y be the parent of x. Then $S_x = \mathrm{bc}(f_x) \cup S_y$, is backward-closed and therefore contains s. Contracting S_y in that graph to a single vertex s' gives a single-source graph S_x/S_y with s' as the source. Adding a dummy target t' in f_x results in an s-t-graph $(S_x/S_y) \cup \{t'\}$. Thus, S_x/S_y is a truncated s-t-graph, and since $C_x = \mathrm{bc}(f_x) \setminus S_y = S_x \setminus S_y = (S_x/S_y) \setminus \{s'\}$ so is C_x.

Let v be a vertex, let I be the set of all nodes in the s-t-decomposition whose associated faces $\{f_x\}_{x \in I}$ are reachable from v, and let $N = \mathrm{lca}(I)$. We now show that v lies in C_N and only in C_N. To see that $v \in C_N$, note that $v \in S_x$ for all $x \in I$, but then $v \in \bigcap_{x \in I} S_x = S_N$. But $v \notin S_a$ for any ancestor a of N by definition of lca, and thus, $v \notin S_y$ for the parent y of N, entailing $v \in S_N \setminus S_y = C_N$. We have now seen that $v \in C_N$ and that $v \notin C_x$ when $x \prec N$ or $N \prec x$. To see that $v \notin C_x$ for any unrelated $x \neq N$, note the following: if x has no descendants in I, then $v \notin V[S_x]$ since all vertices reachable from v lie on some face. Thus, $v \notin C_x \subseteq S_x$.

□

Theorem 3.5. *Any acyclic single-source plane digraph has a good s-t-decomposition.*

We defer the proof to section 3.1. The reason for studying s-t-decompositions in the context of reachability is the following

Lemma 3.6. If $u \rightsquigarrow v$ where $u \in C_x$ and $v \in C_y$ then either $x = y$ or x has a child z that is ancestor to y such that any $u \rightsquigarrow v$ path contains a vertex in F_z.

Proof. Note that whenever $w \rightsquigarrow w'$ with $w' \in C_a$, w must belong to an ancestor of a, since $w \in bc(f_a)$. Thus, x is an ancestor of y, which means that either $x = y$ or x has a child z that is an ancestor to y. But then either w lies on F_x, or F_x is a cycle separating w from w'. In either case, a path from w to w' must contain a vertex on F_x. □

Since (by theorem 3.5) we can assume the alternation number is at most 4, this reduces the reachability question to the problem of finding the at most 4 "last" vertices on $F_z \cap C_x$ that can reach v and then checking in C_x if u can reach either of them. In section 3.3 we will show how to do this efficiently when F_z is *a 2-frame*, that is, has alternation number 2, and in section 3.4 we will extend this to the case when F_z is *a 4-frame*, that is, has alternation number 4.

Theorem 3.7. *There exists a practical RAM data structure that for any planar digraph with n vertices uses $O(n)$ words of $O(\log n)$ bits and can answer reachability queries in constant time. The data structure can be built in linear time.*

Proof. First, build a good s-t-decomposition of G. Such a decomposition exists (Lemma 3.5) and can be built in linear time (Lemma 3.15). Adding DFS pre- and postorder numbers to each node in the tree lets us discover the ancestry relationship between any two vertices in constant time. Then, calculate the structures described in Section 3.3 (in particular $d_2[]$) and Section 3.4 ($c[]$ and $d[]$).

To answer $\mathrm{reachable}(u, v)$, there are the following cases. Let $x = c[u]$ and $y = c[v]$.

1. If $x \not\preceq y$, then u cannot reach v.
2. If $x = y$, then the answer is given by the s-t-graph labelling of C_x from [17].
3. If $x \prec y$ and $d_2[u] == d_2[v]$ there are no 2-frames separating u and v, but since $x \prec y$ there are 4-frames. Let $i = d[u]$, then by 3.51 we can in constant time compute $l_i^0(v)$, $r_i^0(v)$, $l_i^1(v)$, and $r_i^1(v)$. If u can reach any of them, them u can reach v, otherwise no.
4. Otherwise $x \prec y$ and $d_2[u] < d_2[v]$ and there is a 2-frame separating u and v. Let $i = d_2[u]$, then by 3.33 we can in constant time compute $l_i(v)$ and $r_i(v)$. If u can reach any of them, then u can reach v, otherwise no.

Note that the recursive calls in step 3 only leads to questions of type < 3, and similarly the recursive calls in step 4 only leads to questions of type < 4. Thus any query uses case 3 at most twice and case $1 + 2$ at most 8 times. Thus we use only constant time per query. □

A consequence of our construction which might be of independent interest is the following:

Theorem 3.8. *If a planar digraph G admits an s-t-decompostion of height h where all frames have alternation number 2 and 4, there exists an $O(h \log n)$ bit labelling scheme for reachability with evaluation time $O(h)$*

Especially, if a class of planar digraphs have such an s-t-decompositions of constant height, they have an $O(\log n)$ bit labelling scheme for reachability.

3.1 Constructing an s-t-decomposition

The s-t-decomposition recursively chooses a face f and consequently a subgraph $H = bc(f)$ of the graph G induced by all vertices that can reach a vertex on f. Since G was embedded in the plane, the subgraph H is embedded in the plane, and all vertices of $G \setminus H$ lie in a unique face of H. We may choose a tree-cotree composition wisely, such that for each face of H, the restriction of T^* to the subfaces of that face is again a dual spanning tree (Lemma 3.10).

We also have to choose H carefully to ensure logarithmic height, and a limited alternation number on the frames. To ensure at most logarithmic height, we show two cases: 2-frame-nodes have only small children, while for 4-frame-nodes, we only need to ensure that their 4-frame children themselves are small.

Lemma 3.9. Let $G = (V, E)$ be a plane graph, let $G^* = (V^*, E^*)$ be its dual, let (T, T^*) be a tree/cotree decomposition of G, and let S be a subgraph of G such that $S \cap T$ is connected. Then the faces of S correspond to connected components of $T^* \setminus E^*[S]$.

Proof. Let S^* be the dual of S, then $S^* = G^*/(G^* \setminus E^*[S])$ and the claim is equivalent to saying that the components of $G^* \setminus E^*[S]$ correspond to the components of $T^* \setminus E^*[S]$. Consider a pair of faces $f_1, f_2 \in V^*$. Clearly, if they are in separate components of $G^* \setminus E^*[S]$, they are also in separate components in $T^* \setminus E^*[S]$. On the other hand, suppose f_1 and f_2 are in different components in $T^* \setminus E^*[S]$. Then there exists an edge $e^* \in E^*[S] \cap T^*$ separating them. The corresponding edge $e \in E[S]$ induces a cycle in T, which is also part of S since $S \cap T$ is connected. The dual to that cycle is an edge cut in G^* that separates f_1 from f_2. □

Lemma 3.10. Let T be a spanning tree where all edges point away from the source s of G, then for any node x in an st-decomposition of G, the subgraph T_x^* of T^* induced by $V^*[G_x^*]$ is a connected subtree of T^*.

Proof. If x is the root, this trivially holds. If x has a parent y, G_x^* corresponds to a face in S_y. Now $S_y \cap T$ is connected since S_y is the union of backward-closed graphs, and the result follows from Lemma 3.9. □

Lemma 3.11. Let x be a node in an st-decomposition whose parent frame F_x has alternation number 2, and let A^* be the set of faces in T_x^* incident to the target corner of F_x. Then for any child y of x:

$$A^* \subseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 4.$$
$$A^* \not\subseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 2.$$

Proof. Let t_x be the target corner of F_x and let A^* be the set of faces in ${T_x}^*$ incident to t_x. For any child y if x, F_y consists of a (possibly empty) segment of F_x and two directed paths that meet at a new target corner t_y. Each target corner of F_y must therefore be at either t_x or t_y. Now if $A^* \subseteq V^*[T_y^*]$, then both t_x and t_y are target corners of F_y, otherwise only t_y is. Either way the result follows. □

Lemma 3.12. Let x be a node in an st-decomposition whose parent frame F_x has alternation number 4, and let A^{0^*} and A^{1^*} be the sets of faces in T_x^* incident to the target corners of F_x. Then for any child y of x:

$$A^{0^*} \not\subseteq V^*[T_y^*] \vee A^{1^*} \not\subseteq V^*[T_y^*] \implies F_y \text{ has alternation number at most } 4.$$
$$A^{0^*} \not\subseteq V^*[T_y^*] \wedge A^{1^*} \not\subseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 2.$$

Proof. Let t_x^0 and t_x^1 be the two target corners of F_x and for $i \in \{0,1\}$ let A^{i^*} be the set of faces in ${T_x}^*$ incident to t_x^i. For any child y of x, F_y consists of a (possibly empty) segment of F_x and two directed paths that meet at a new target corner t_y. Each target corner of F_y must therefore be at either t_y, t_x^0, or t_x^1. Now if $A^{i^*} \not\subseteq V^*[T_y^*]$ for some $i \in \{0,1\}$, then t_x^i is not a target corner of F_y. So the number of target corners in F_y is at least 1, and at most 3 minus the number of such i, and the result follows. □

proof of theorem 3.5. Let s be the source of G and let (T, T^*) be a tree/cotree decomposition of G such that all edges in T point away from s. The st-decomposition can be constructed recursively as follows. Start with the root. In each step we have a node x and by Lemma 3.10 the subgraph T_x^* induced in T^* by $V^*[G_x^*]$ is a tree. The goal is to select a face f_x such that for each child y:

- The alternation number of F_y is at most 4, and
- For each child z of y (and thus grandchild of x), $|T_z^*| \leq \frac{1}{2}|T_x^*|$.

If we can do this for all x, we are done. There are 3 cases:

x is the root Let f_x be the median of $T_x^* = T^*$. Then for each child y, $|T_y^*| \leq \frac{1}{2}|T_x^*|$, and, since $S_x = \mathrm{bc}(f_x)$ is a truncated s-t-graph with a single source, f_y has alternation number 2.

F_x has alternation number 2 Let f_x be the median of T_x^*. Then for each child y, $|T_y^*| \leq \frac{1}{2}|T_x^*|$, and, by Lemma 3.11, f_y has alternation number at most 4.

F_x **has alternation number** 4 Let t_0 and t_1 be the local targets of F_x and let $f_0, f_1 \in V^*[T_x^*]$ be (not necessarily distinct) faces incident to t_0 and t_1 respectively. Now choose f_x as the projection of the median m of T_x^* on the path $f_0, \ldots, f_1$ in T_x^*. By Lemma 3.12 this means that for any child y of x, the alternation number of the parent frame F_y is at most 4.
- If $f_x = m$ then $|T_y^*| \leq \frac{1}{2}|T_x^*|$.
- If $f_x \neq m$ and T_y^* is not the component of m in $T_x^* \setminus E^*[\mathrm{bc}(f_x)]$, then $|T_y^*| \leq \frac{1}{2}|T_x^*|$.
- If $f_x \neq m$, and T_y^* is the component of m in $T_x^* \setminus E^*[\mathrm{bc}(f_x)]$, then T_y^* contains neither f_0 nor f_1, so by Lemma 3.12 the parent frame F_y has alternation number at most 2 and we have just shown this means any child z of y has $|T_z^*| \leq \frac{1}{2}|T_y^*| \leq \frac{1}{2}|T_x^*|$. □

Note that this construction can be implemented in linear time by using ideas similar to [2].

3.2 Constructing a good s-t-decomposition in linear time

In the construction of an s-t-decomposition, a face is chosen, some edges are deleted, and new connected components of the dual graph arise. We then recurse on the new connected components of the dual graph. By Lemma 3.10 we can choose a tree/cotree-decomposition such that each component that arises is spanned by a subtree of the cotree.

To obtain linear construction time, we use a variation of the decremental tree connectivity algorithm from [2] to keep track of the subtrees of the cotree, and associate some information with each subtree. In particular, when T_x^* is a component at some point, we can in constant time find the node x.

For each node x we keep the set of target vertices on F_x (or $\emptyset$ if x is the root), and a face in T_x^* incident to each target in the set.

Build a top tree (see [1]) of height $O(\log n)$ over T^*, and let v_{n-i}^* be the i'th face that stops being boundary during the construction. Using this enumeration, the boundary faces of a cluster will be visited before boundary faces of their descendants. We use this ordering to find the splitting faces of the s-t-decomposition.

For each v_i^*, we can use the connectivity structure to find the relevant node x to split. We then need to choose the target face f_x defining the split. If x is the root or F_x is a 2-frame, we just set $f_x = v_i^*$. If F_x is a 4-frame, the information in x contains a pair of faces f_1, f_2 and we use a static nearest common ancestor data structure from Harel and Tarjan [8] to find the projection $f_x = \pi(v_i^*)$ of v_i^* on $f_1, \cdots, f_2$. Note that the projection of v_i^* is always contained in the same connected component as f_1, f_2, and thus, the data structure for the whole tree suffices to answer this query for the particular subtree.

Once f_x has been selected, we traverse the graph backwards from the vertices of f_x until we have found all the edges with destination in C_x. This search takes $|C_x|$ time. We delete these edges from the forest as we go along. Once we are done, we take all targets in C_x and select an incident face for each component it is incident to. This again takes $|C_x|$ time. If $f_x \neq v_i^*$ we try with v_i^* again, otherwise we move on to v_{i+1}^*.

Lemma 3.13. The s-t-decomposition constructed via the approach sketched above has no frame of alternation number > 4.

Proof. Components with 2-frames always have children with 2- and 4-frames. For components with 4-frames, this follows directly from Lemma 3.11, since we chose a splitting face on the cotree path between faces near the two targets. □

Lemma 3.14. The s-t-decomposition constructed via the approach sketched above has height $O(\log n)$.

Proof. Since the top-tree has height $O(\log n)$, choosing the boundary face v_i^* as a splitting face every time would result in a tree of the same height; $O(\log n)$. However, for each 4-frame, we might choose a face

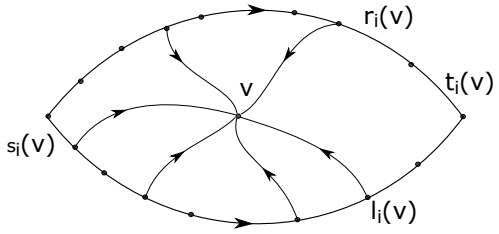


Figure 2: The best two vertices that can reach v on level i.

$f_x \neq v_i^*$ which is the projection of v_i^* on $f_1 \dots f_2$. As noted in Lemma 3.11, when this happens, v_i^* will lie in a child which has a 2-frame. But then, v_i^* will be the splitting face for that child. We thus increase the height by no more than a factor 2, and the s-t-decomposition has height $2O(\log n) = O(\log n)$. □

Lemma 3.15. Let $G = (V, E)$ be a plane single-source graph with source s, then we can construct a good s-t-decomposition of G in linear time.

Proof. Since the top-tree can be constructed in linear time, and since the decremental connectivity for trees takes linear time, and since the static nearest common ancestor data stucture is constructed in linear time and answers queries in constant time, the construction takes linear time. By Lemma 3.13 and 3.14, the resulting s-t-decomposition is good. □

3.3 2-frames

Definition 3.16. Let $\mathcal{T}$ be an st-decomposition of $G = (V, E)$. Then we can define a 2-*frame-decomposition* $\mathcal{T}_2$ by contracting each edge in $\mathcal{T}$ that corresponds to a 4-frame. For each node x in $\mathcal{T}_2$ that is contracted from a set of nodes $Y \subseteq \mathcal{T}$ define $C_x := \bigcup_{y \in Y} C_y$ and if x is not the root, define $F_x := F_{\text{lca}(Y)}$ and $E_x := E_{\text{lca}(Y)}$. Then F_x is a 2-frame, and we can define s_x to be the source corner, and t_x to be the target corner on F_x.

Definition 3.17. Let $(\mathcal{L}, \mathcal{R})$ be the partition of $\cup_{x \in \mathcal{T}_2} E_x$ defined as follows: For each $(u, v) \in \cup_{x \in \mathcal{T}_2} E_x$ let y be the node (if it exists) closest to the root of $\mathcal{T}_2$ such that $(u, v) \in E_y$ but u is not the target vertex of F_y. If y exists and (u, v) is incident to a corner on the clockwise disegment of F_y between s_y and t_y assign (u, v) to $\mathcal{R}$, otherwise assign (u, v) to $\mathcal{L}$.

Definition 3.18. Let $\mathcal{T}_2$ be an 2-frame-decomposition of $G = (V, E)$. For any vertex $v \in V$ define:

$c_2[v]$:= The node x in $\mathcal{T}_2$ such that $v \in V[C_x]$
$d_2[v]$:= The depth of $c_2[v]$ in $\mathcal{T}_2$

Definition 3.19. For any $0 \le i < d_2[v]$, let x be the ancestor of $c_2[v]$ at depth $i+1$ and define:

$$
\begin{aligned}
E_i(v) &:= E_x \\
L_i(v) &:= E_x \cap \mathcal{L} \\
R_i(v) &:= E_x \cap \mathcal{R} \\
\widehat{L}_i(v) &:= \left\{(w,w') \in L_i(v) \middle|\ w' \rightsquigarrow v\right\} \\
\widehat{R}_i(v) &:= \left\{(w,w') \in R_i(v) \middle|\ w' \rightsquigarrow v\right\} \\
\widehat{F}_i(v) &:= \widehat{L}_i(v) \cup \widehat{R}_i(v) \\
l_i(v) &:= \begin{cases} \bot & \text{if } \widehat{L}_i(v) = \emptyset \\ \text{the last vertex in } \operatorname{init}(\widehat{L}_i(v)) \text{ on the counterclockwise dipath of } F_x & \text{otherwise} \end{cases} \\
r_i(v) &:= \begin{cases} \bot & \text{if } \widehat{R}_i(v) = \emptyset \\ \text{the last vertex in } \operatorname{init}(\widehat{R}_i(v)) \text{ on the clockwise dipath of } F_x & \text{otherwise} \end{cases} \\
s_i(v) &:= \text{The vertex associated with } s_x \\
t_i(v) &:= \text{The vertex associated with } t_x
\end{aligned}
$$

Additionally, let $L_i(v)$ and $\widehat{L}_i(v)$ be totally ordered by the position of the starting vertices on the counterclockwise disegment of F_x and the clockwise order around each starting vertex. Similarly let $R_i(v)$ and $\widehat{R}_i(v)$ be totally ordered by the position of the starting vertices on the clockwise disegment of F_x and the counterclockwise order around each starting vertex.

The goal in this section is a data structure for efficiently computing $l_i(v)$ and $r_i(v)$ for $0 \le i < d_2[v]$.

Lemma 3.20. For any vertex $v \in V$ and $0 \le i < d_2[v]$: $\widehat{F}_i(v) \neq \emptyset$

Proof. Let x be the ancestor of $c_2[v]$ at depth $i+1$. Since G is a single-source graph, there is a path from s to v. This path must contain a vertex in $V[F_x]$. But then the edge following the last such vertex on the path must be in $\widehat{L}_i(v) \cup \widehat{R}_i(v)$ which is therefore nonempty. □

Lemma 3.21. For any $u, v \in V$ and $0 \le i < d_2[u]$: If $u \rightsquigarrow v$ then $\widehat{L}_i(u) \subseteq \widehat{L}_i(v)$ and $\widehat{R}_i(u) \subseteq \widehat{R}_i(v)$.

Proof. Since $u \rightsquigarrow v$, $c_2[u]$ is ancestor to $c_2[v]$ and so $L_i(u) = L_i(v)$ and hence $\widehat{L}_i(u) \subseteq \widehat{L}_i(v)$. Similarly, $R_i(u) = R_i(v)$ and $\widehat{R}_i(u) \subseteq \widehat{R}_i(v)$. □

Lemma 3.22. Given any vertex $v \in V$, $0 \le i < d_2[v]$, and $(w,w') \in E_i(v)$. Then:

$$
\begin{aligned}
(w,w') \in \widehat{L}_i(v) \quad &\implies \quad (w,w') \in \widehat{L}_{i'}(v) \text{ for all } i', d_2[w] \le i' < \min\left\{d_2[w'], d_2[v]\right\} \\
(w,w') \in \widehat{R}_i(v) \quad &\implies \quad (w,w') \in \widehat{R}_{i'}(v) \text{ for all } i', d_2[w] \le i' < \min\left\{d_2[w'], d_2[v]\right\}
\end{aligned}
$$

Proof. Let $j = d_2[w]$ and $k = \min\{d_2[w'], d_2[v]\}$. Clearly $(w,w') \in E_{i'}$ for all $j \le i' < k$. Suppose $(w,w') \in \widehat{L}_i(v) \subseteq L_i(v)$, then since $j \le i < k$ the definition give us $(w,w') \in L_{i'}(v)$ for all $j \le i' < k$. And since $w' \rightsquigarrow v$ this implies $(w,w') \in \widehat{L}_{i'}(v)$ for all $j \le i' < k$ and the result follows. The case for R is symmetric. □

Definition 3.23. For any vertex $v \in V$ let

$$p_l[v] := \begin{cases} \perp & \text{if } d_2[v] = 0 \\ l_{d_2[v]-1}(v) & \text{otherwise} \end{cases}$$
$$p_r[v] := \begin{cases} \perp & \text{if } d_2[v] = 0 \\ r_{d_2[v]-1}(v) & \text{otherwise} \end{cases}$$

and let T_l and T_r denote the rooted forests over V whose parent pointers are p_l and p_r respectively.

Definition 3.24. For any $v \in V \cup \{\perp\}$, and $i \geq 0$ let

$$l'_i(v) := \begin{cases} v & \text{if } v = \perp \vee d_2[v] \leq i \\ l'_i(p_l[v]) & \text{otherwise} \end{cases}$$
$$r'_i(v) := \begin{cases} v & \text{if } v = \perp \vee d_2[v] \leq i \\ r'_i(p_r[v]) & \text{otherwise} \end{cases}$$

Lemma 3.25. Let $v \in V$, and $i \geq 0$ be given, then

$$\begin{array}{lllll} i = d_2[v] - 1 & \implies & l'_i(v) = l_i(v) & \wedge & r'_i(v) = r_i(v) \\ i \leq d_2[v] - 1 & \implies & l'_i(v) \in \text{init}(\widehat{L}_i(v)) \cup \{\perp\} & \wedge & r'_i(v) \in \text{init}(\widehat{R}_i(v)) \cup \{\perp\} \\ i > d_2[v] - 1 & \implies & l'_i(v) = v & \wedge & r'_i(v) = v \end{array}$$

Proof. We will show this for l' only, as r' is completely symmetrical. If $i > d_2[v] - 1$ then $d_2[v] \leq i$ and we get $l'_i(v) = v$ directly from the definition of l'. Similarly if $i = d_2[v] - 1$ then $l'_i(v) = l'_i(p_l[v]) = l'_i(l_{d_2[v]-1}(v)) = l'_i(l_i(v)) = l_i(v) \in \text{init}(\widehat{L}_i(v)) \cup \{\perp\}$. Finally suppose $i < d_2[v] - 1$. If $l'_i(v) = \perp$ we are done, so suppose that is not the case. Let u be the child of $l'_i(v)$ in T_l that is ancestor to v. Then $l'_i(v) = l'_i(u) = p_l[u] = l_{d_2[u]-1}(u)$. By definition of $l_{d_2[u]-1}(u)$ there exists an edge $(w, w') \in \widehat{L}_{d_2[u]-1}$ where $w = l_{d_2[u]-1}(u)$ and $d_2[w] \leq i < d_2[w'] \leq d_2[u]$ and by setting $(v, i, (w, w')) = (u, d_2[u] - 1, (w, w'))$ in lemma 3.22 we get $(w, w') \in \widehat{L}_i(u)$, and therefore $l'_i(v) \in \text{init}(\widehat{L}_i(u))$. But since $u \rightsquigarrow v$ we have $\widehat{L}_i(u) \subseteq \widehat{L}_i(v)$ by Lemma 3.21 and we are done. □

Lemma 3.26. Let $v \in V$ and $0 \leq i \leq j$ then

$$l'_i(l'_j(v)) = l'_i(v) \qquad \wedge \qquad r'_i(r'_j(v)) = r'_i(v)$$

Proof. $l'_j(v)$ is on the path from v to $l'_i(v)$ in T_l, so this follows trivially from the recursion. The case for r' is symmetric. □

Lemma 3.27. Let $v \in V$, and $0 \leq i < d_2[v] - 1$, then

$$l_i(v) = \perp \implies l'_i(l_{i+1}(v)) = \perp \qquad \wedge \qquad r_i(v) = \perp \implies r'_i(r_{i+1}(v)) = \perp$$

Proof. If $l_i(v) = \perp$ then $\widehat{L}_i(v) = \emptyset$, so either $l_{i+1}(v) = \perp$ implying $l'_i(l_{i+1}(v)) = \perp$ by the definition of l', or $l_{i+1}(v) \notin \text{init}(\widehat{L}_i(v))$ so $d_2[l_{i+1}(v)] = i + 1$ and by Lemma 3.25 and Lemma 3.21 $l'_i(l_{i+1}(v)) \in \text{init}(\widehat{L}_i(l_{i+1}(v))) \cup \{\perp\} \subseteq \text{init}(\widehat{L}_i(v)) \cup \{\perp\} = \{\perp\}$ so again $l'_i(l_{i+1}(v)) = \perp$. The case for r is symmetric. □

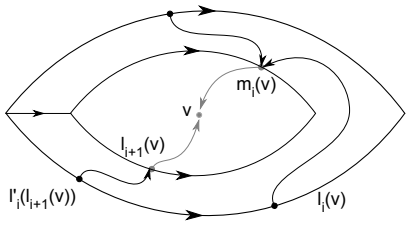


Figure 3: The best path from $L_i(v)$ goes via $r_{i+1}(v)$.

Lemma 3.28 (Crossing lemma)**.** Let $v \in V$, and $0 \leq i < d_2[v] - 1$.

$$\begin{aligned} l_i(v) \neq l'_i(l_{i+1}(v)) \implies\ & l_i(v) = l'_i(m) \wedge r_i(v) = r'_i(m) \wedge d_2[m] = i+1 \\ & \text{where } m = r_{i+1}(v) \neq \perp \\ r_i(v) \neq r'_i(r_{i+1}(v)) \implies\ & l_i(v) = l'_i(m) \wedge r_i(v) = r'_i(m) \wedge d_2[m] = i+1 \\ & \text{where } m = l_{i+1}(v) \neq \perp \end{aligned}$$

Proof. Suppose $l_i(v) \neq l'_i(l_{i+1}(v))$ (the case $r_i(v) \neq r'_i(r_{i+1}(v))$ is symmetrical). Then $l_i(v) \neq \perp$ by lemma 3.27. Thus there is a last edge $(w, w') \in \widehat{L}_i(v)$ with $w = l_i(v)$ and $d_2[w] \leq i < d_2[w']$ and a path $P = w' \rightsquigarrow v$.

Now $(w, w') \notin E_{i+1}(v)$ since otherwise by Definition 3.19 $(w, w') \in L_{i+1}(v)$ and since $w' \rightsquigarrow v$ even $(w, w') \in \widehat{L}_{i+1}(v)$ implying $l_i(v) = l_{i+1}(v)$ and thus $l_i(v) = l'_i(l_{i+1}(v))$ by lemma 3.25, contradicting our assumption.

Since $(w, w') \notin E_{i+1}(v)$, the path P must cross $\widehat{F}_{i+1}(v)$. Let (u, u') be the last edge in $P \cap \widehat{F}_{i+1}(v)$. Then $w' \rightsquigarrow u$ so $d_2[u] \geq i+1$ and $(u, u') \notin L_{i+1}(v)$ since otherwise $d_2[l_{i+1}(v)] = i+1$ and hence by Lemma 3.25 $l_i(v) = l'_i(l_{i+1}(v))$, again contradicting our assumption. Since $\widehat{F}_{i+1}(v) \neq \emptyset$, we therefore have $(u, u') \in \widehat{R}_{i+1}(v)$.

But then we can choose P so it goes through (m, m') where $m = r_{i+1}(v) \neq \perp$. Now $i+1 \leq d_2[w'] \leq d_2[r_{i+1}(v)] \leq i+1$ so $d_2[m] = i+1$.

Let e be the last edge in $\widehat{R}_i(v)$ then any path $r_i(v) \rightsquigarrow v$ that starts with e crosses $P \cup \widehat{R}_{i+1}(v)$, implying that there exists such a path that contains (m, m') and thus $r_i(v) = r_i(m)$. Since $d_2[m] = i+1$, then $l_i(v) = l'_i(m)$ and $r_i(v) = r'_i(m)$ follows from lemma 3.25. □

Definition 3.29. Let $v \in V$ and $0 \leq i < d_2[v]$.

$$m_i(v) := \begin{cases} v & \text{if } i+1 = d_2[v] \\ l_{i+1}(v) & \text{if } i+1 < d_2[v] \wedge r_i(v) \neq r'_i(r_{i+1}(v)) \\ r_{i+1}(v) & \text{if } i+1 < d_2[v] \wedge l_i(v) \neq l'_i(l_{i+1}(v)) \\ m_{i+1}(v) & \text{otherwise} \end{cases}$$

Corollary 3.30. Let $v \in V$ and $0 \leq i < d_2[v] - 1$. If $l_i(v) \neq l'_i(l_{i+1}(v))$ or $r_i(v) \neq r'_i(r_{i+1}(v))$ then

$$l_i(v) = l'_i(m_i(v)) \qquad \wedge \qquad r_i(v) = r'_i(m_i(v)) \qquad \wedge \qquad d_2[m_i(v)] = i+1$$

Proof. This is just a reformulation of lemma 3.28 in terms of $m_i(v)$. □

Lemma 3.31. For any vertex $v \in V$ and $0 \leq i < d_2[v]$

$$l_i(v) = l'_i(m_i(v)) \qquad \wedge \qquad r_i(v) = r'_i(m_i(v))$$

Proof. The proof is by induction on j, the number of times the "otherwise" case is used before reaching one of the other cases when expanding the recursive definition of $m_i(v)$.

For $j = 0$, either $i + 1 = d_2[v]$ and the result follows from Lemma 3.25, or $i + 1 < d_2[v]$ and $l_i(v) \neq l'_i(l_{i+1}(v))$ or $r_i(v) \neq r'_i(r_{i+1}(v))$. In either case we have by Corollary 3.30, that $l_i(v) = l'_i(m_i(v))$ and $r_i(v) = r'_i(m_i(v))$.

For $j > 0$ we have $i + 1 < d_2[v]$ and $l_i(v) = l'_i(l_{i+1}(v))$ and $r_i(v) = r'_i(r_{i+1}(v))$ and $m_i(v) = m_{i+1}(v)$. By induction we can assume that $l_{i+1}(v) = l'_{i+1}(m_{i+1}(v))$ and $r_{i+1}(v) = r'_{i+1}(m_{i+1}(v))$. Then by Lemma 3.26, $l'_i(l_{i+1}(v)) = l'_i(l'_{i+1}(m_{i+1}(v))) = l'_i(m_{i+1}(v)) = l'_i(m_i(v))$, showing that $l_i(v) = l'_i(m_i(v))$ as desired. The case for r is symmetric. □

Definition 3.32. For any vertex $v \in V$, let

$$M[v] := \{i |\ 0 < i < d_2[v] \wedge m_{i-1}(v) \neq m_i(v)\}$$
$$p_m[v] := \begin{cases} \perp & \text{if } M[v] = \emptyset \\ m_{\max M[v]-1}(v) & \text{otherwise} \end{cases}$$

And define T_m as the rooted forest over V whose parent pointers are p_m.

Theorem 3.33. *There exists a practical RAM data structure that for any good st-decomposition of a graph with n vertices uses $\mathcal{O}(n)$ words of $\mathcal{O}(\log n)$ bits and can answer $l_i(v)$ and $r_i(v)$ queries in constant time.*

Proof. For any vertex $v \in V$, let

$$D_l[v] := \{i |\ v \text{ has a proper ancestor } w \text{ in } T_l \text{ with } d_2[w] = i\}$$
$$D_r[v] := \{i |\ v \text{ has a proper ancestor } w \text{ in } T_r \text{ with } d_2[w] = i\}$$

Now, store levelancestor structures for each of T_l, T_r, and T_m, together with $d_2[v]$, $D_l[v]$, $D_r[v]$, and $M[v]$ for each vertex. Since the height of the st-decomposition is $\mathcal{O}(\log n)$ each of $D_l[v]$, $D_r[v]$, and $M[v]$ can be represented in a single $\mathcal{O}(\log n)$-bit word.

This representation allows us to find $d_2[m_i(v)] = \text{succ}(M[v] \cup \{d_2[v]\}, i)$ in constant time, as well as computing the depth in T_m of $m_i(v)$. Then using the levelancestor structure for T_m we can compute $m_i(v)$ in constant time.

Similarly, this representation of the $D_l[v]$ set lets us compute the depth in T_l of $l'_i(v)$ in constant time, and with the levelancestor structure that lets us compute $l'_i(v)$ in constant time. A symmetric argument shows that we can compute $r'_i(v)$ in constant time.

Finally, lemma 3.31 says we can compute $l_i(v)$ and $r_i(v)$ in constant time given constant-time functions for l', r', and m. □

3.4 4-frames

Definition 3.34. Let x be a node in an s-t-decomposition such that F_x is a 4-frame, and let y be its parent. Let s^0_x and s^1_x be the source corners on F_x and let t^0_x and t^1_x be the target corners on F_x, numbered such that their clockwise cyclic order on F_x is $s^0_x, t^0_x, s^1_x, t^1_x$, and such that if F_y is a 4-frame there is an $\alpha \in \{0, 1\}$ so $t^\alpha_x = t^\alpha_y$.

Definition 3.35. Let $\mathcal{E}_4$ be the set of edges (u, v) such if x is the node in the s-t-decomposition that contains v, then $(u, v) \in E_x$ and F_x is a 4-frame. Let $(\mathcal{L}^0, \mathcal{R}^0, \mathcal{L}^1, \mathcal{R}^1)$ be the partition of $\mathcal{E}_4$ defined as follows: For each $(u, v) \in \mathcal{E}_4$ let x be the node such that $v \in C_x$, and let y be the node (if it exists) closest to the root of $\mathcal{T}$ such that

- For any z that is ancestor to x and descendent to y, F_z is a 4-frame.
- $(u, v) \in E_y$.
- u is not a target vertex of F_y.

If y exists, then (u, v) is incident to a corner c on F_y. If there is an $\alpha \in \{0, 1\}$ such that c is on the clockwise disegment of F_y between s_y^α and t_y^α we assign (u, v) to $\mathcal{R}^\alpha$. Otherwise there must be an $\alpha \in \{0, 1\}$ such that c is on the counterclockwise disegment of F_y between $s_y^{1-\alpha}$ and t_y^α, and we assign (u, v) to $\mathcal{L}^\alpha$. If no such y exists, (u, v) must be incident to t_x^α for some $\alpha \in \{0, 1\}$ and we (arbitrarily) assign (u, v) to $\mathcal{L}^\alpha$.

Definition 3.36. Let $\mathcal{T}$ be an st-decomposition of $G = (V, E)$. For any vertex $v \in V$ define:

$$
\begin{aligned}
c[v] &:= \text{The node } x \text{ in } \mathcal{T} \text{ such that } v \in V[C_x] \\
d[v] &:= \text{The depth of } c[v] \text{ in } \mathcal{T} \\
J_2[v] &:= \{\text{depth}(x) |\ x \text{ is a non-root ancestor to } c[v] \text{ in } \mathcal{T} \text{ and } F_x \text{ is a 2-frame}\} \\
j_2[v] &:= \max(J_2[v]))
\end{aligned}
$$

The number $j_2[v]$ is especially useful for 4-frame nodes. On the path from the root to the component of v in the s-t-decomposition tree, there will be a last component whose frame is a 2-frame. We call the depth of the next component on the path $j_2[v]$. If $c[v]$ has a 4-frame, then for the rest of the path, that is, depth i with $j_2[v] \leq i < d[v]$, we will have 4-frames nested in 4-frames, which gives a lot of useful structure.

Definition 3.37. For any $j_2[v] \leq i < d[v]$ and $\alpha \in \{0, 1\}$, let x be the ancestor of $c[v]$ at depth $i + 1$ and define:

$$
\begin{aligned}
E_i(v) &:= E_x \\
L_i^\alpha(v) &:= E_x \cap \mathcal{L}^\alpha \\
R_i^\alpha(v) &:= E_x \cap \mathcal{R}^\alpha \\
\widehat{L}_i^\alpha(v) &:= \left\{(w, w') \in L_i^\alpha(v) \middle|\ w' \rightsquigarrow v\right\} \\
\widehat{R}_i^\alpha(v) &:= \left\{(w, w') \in R_i^\alpha(v) \middle|\ w' \rightsquigarrow v\right\} \\
\widehat{F}_i(v) &:= \widehat{L}_i^0(v) \cup \widehat{R}_i^0(v) \cup \widehat{L}_i^1(v) \cup \widehat{R}_i^1(v) \\
l_i^\alpha(v) &:= \begin{cases} \bot & \text{if } \widehat{L}_i^\alpha(v) = \emptyset \\ \text{the last vertex in } \text{init}(\widehat{L}_i^\alpha(v)) \text{ on the counterclockwise dipath of } F_x & \text{otherwise} \end{cases} \\
r_i^\alpha(v) &:= \begin{cases} \bot & \text{if } \widehat{R}_i^\alpha(v) = \emptyset \\ \text{the last vertex in } \text{init}(\widehat{R}_i^\alpha(v)) \text{ on the clockwise dipath of } F_x & \text{otherwise} \end{cases} \\
s_i^\alpha(v) &:= \text{The vertex associated with } s_x^\alpha \\
t_i^\alpha(v) &:= \text{The vertex associated with } t_x^\alpha
\end{aligned}
$$

Additionally, let $L_i^\alpha(v)$ and $\widehat{L}_i^\alpha(v)$ be totally ordered by the position of the starting vertices on the counterclockwise disegment of F_x and the clockwise order around each starting vertex. Similarly let $R_i^\alpha(v)$ and $\widehat{R}_i^\alpha(v)$ be totally ordered by the position of the starting vertices on the clockwise disegment of F_x and the counterclockwise order around each starting vertex.

We know from Section 3.3 that we can find the relevant vertices on each 2-frame surrounding v. The goal in this section is a data structure for efficiently computing $l_i^\alpha(v)$ and $r_i^\alpha(v)$ for $j_2[v] \leq i < d[v]$.

Lemma 3.38. For any vertex $v \in V$ and $j_2[v] \leq i < d[v]$: $\widehat{F}_i(v) \neq \emptyset$

Proof. Let x be the ancestor of $c[v]$ at depth $i+1$. Since G is a single-source graph, there is a path from s to v. This path must contain a vertex in $V[F_x]$. But then the edge following the last such vertex on the path must be in $\widehat{L}_i^0(v) \cup \widehat{R}_i^0(v) \cup \widehat{L}_i^1(v) \cup \widehat{R}_i^1(v)$ which is therefore nonempty. □

Lemma 3.39. For any $u, v \in V$, $j_2[v] \leq i < d[u]$, and $\alpha \in \{0,1\}$: If $u \rightsquigarrow v$ then $\widehat{L}_i^\alpha(u) \subseteq \widehat{L}_i^\alpha(v)$ and $\widehat{R}_i^\alpha(u) \subseteq \widehat{R}^a lpha_i(v)$.

Proof. Since $u \rightsquigarrow v$, $c[u]$ is ancestor to $c[v]$ and so $L_i^\alpha(u) = L_i^\alpha(v)$ and hence $\widehat{L}_i^\alpha(u) \subseteq \widehat{L}_i^\alpha(v)$. Similarly, $R_i^\alpha(u) = R_i^\alpha(v)$ and $\widehat{R}_i^\alpha(u) \subseteq \widehat{R}_i^\alpha(v)$. □

Lemma 3.40. Given any vertex $v \in V$, $j_2[v] \leq i < d[v]$, $\alpha \in \{0,1\}$, and $(w, w') \in E_i(v)$. Then:

$$(w,w') \in \widehat{L}_i^\alpha(v) \implies (w,w') \in \widehat{L}_{i'}^\alpha(v) \text{ for all } i', \max\{d[w], j_2[v]\} \leq i' < \min\{d[w'], d[v]\}$$
$$(w,w') \in \widehat{R}_i^\alpha(v) \implies (w,w') \in \widehat{R}_{i'}^\alpha(v) \text{ for all } i', \max\{d[w], j_2[v]\} \leq i' < \min\{d[w'], d[v]\}$$

Proof. Let $j = \max\{d[w], j_2[v]\}$ and $k = \min\{d[w'], d[v]\}$. Clearly $(w,w') \in E_{i'}$ for all $j \leq i' < k$. Suppose $(w,w') \in \widehat{L}_i^\alpha(v) \subseteq L_i^\alpha(v)$, then since $j \leq i < k$ the definition give us $(w,w') \in L_{i'}^\alpha(v)$ for all $j \leq i' < k$. And since $w' \rightsquigarrow v$ this implies $(w,w') \in \widehat{L}_{i'}^\alpha(v)$ for all $j \leq i' < k$ and the result follows. The case for R is symmetric. □

Definition 3.41. For any vertex $v \in V$ and $\alpha \in \{0,1\}$ let

$$p_l^\alpha[v] := \begin{cases} \bot & \text{if } d[v] = 0 \vee F_{d[v]-1}(v) \text{ is a 2-frame} \\ l^\alpha_{d[v]-1}(v) & \text{otherwise} \end{cases}$$
$$p_r^\alpha[v] := \begin{cases} \bot & \text{if } d[v] = 0 \vee F_{d[v]-1}(v) \text{ is a 2-frame} \\ r^\alpha_{d[v]-1}(v) & \text{otherwise} \end{cases}$$

and let T_l^α and T_r^α denote the rooted forests over V whose parent pointers are p_l^α and p_r^α respectively.

Definition 3.42. For any $v \in V \cup \{\bot\}$, $\alpha \in \{0,1\}$, and $i \geq j_2[v]$ let

$$l_i'^\alpha(v) := \begin{cases} v & \text{if } v = \bot \vee d[v] \leq i \\ l_i'^\alpha(p_l^\alpha[v]) & \text{otherwise} \end{cases}$$
$$r_i'^\alpha(v) := \begin{cases} v & \text{if } v = \bot \vee d[v] \leq i \\ r_i'^\alpha(p_r^\alpha[v]) & \text{otherwise} \end{cases}$$

Lemma 3.43. Let $v \in V$, $\alpha \in \{0,1\}$, and $i \geq j_2[v]$ be given, then

$$i = d[v]-1 \implies l_i'^\alpha(v) = l_i^\alpha(v) \quad \wedge \quad r_i'^\alpha(v) = r_i^\alpha(v)$$
$$i \leq d[v]-1 \implies l_i'^\alpha(v) \in \text{init}(\widehat{L}_i^\alpha(v)) \cup \{\bot\} \quad \wedge \quad r_i'^\alpha(v) \in \text{init}(\widehat{R}_i^\alpha(v)) \cup \{\bot\}$$
$$i > d[v]-1 \implies l_i'^\alpha(v) = v \quad \wedge \quad r_i'^\alpha(v) = v$$

Proof. We will show this for l' only, as r' is completely symmetrical. If $i > d[v] - 1$ then $d[v] \leq i$ and we get $l'^{\alpha}_i(v) = v$ directly from the definition of l'. Similarly if $i = d[v] - 1$ then $l'^{\alpha}_i(v) = l'^{\alpha}_i(p^{\alpha}_l[v]) = l'^{\alpha}_i(l^{\alpha}_{d[v]-1}(v)) = l'^{\alpha}_i(l^{\alpha}_i(v)) = l^{\alpha}_i(v) \in \text{init}(\widehat{L}^{\alpha}_i(v)) \cup \{\perp\}$. Finally suppose $i < d[v] - 1$. If $l'^{\alpha}_i(v) = \perp$ we are done, so suppose that is not the case. Let u be the child of $l'^{\alpha}_i(v)$ in T_l that is ancestor to v. Then $l'^{\alpha}_i(v) = l'^{\alpha}_i(u) = p^{\alpha}_l[u] = l^{\alpha}_{d[u]-1}(u)$. By definition of $l^{\alpha}_{d[u]-1}(u)$ there exists an edge $(w, w') \in \widehat{L}^{\alpha}_{d[u]-1}$ where $w = l^{\alpha}_{d[u]-1}(u)$ and $d[w] \leq i < d[w'] \leq d[u]$ and by setting $(v, i, (w, w')) = (u, d[u] - 1, (w, w'))$ in lemma 3.40 we get $(w, w') \in \widehat{L}^{\alpha}_i(u)$, and therefore $l'^{\alpha}_i(v) \in \text{init}(\widehat{L}^{\alpha}_i(u))$. But since $u \rightsquigarrow v$ we have $\widehat{L}^{\alpha}_i(u) \subseteq \widehat{L}^{\alpha}_i(v)$ by Lemma 3.39 and we are done. □

Lemma 3.44. Let $v \in V$, $\alpha \in \{0, 1\}$, and $j_2[v] \leq i \leq j$ then

$$l'^{\alpha}_i(l'^{\alpha}_j(v)) = l'^{\alpha}_i(v) \qquad \wedge \qquad r'^{\alpha}_i(r'^{\alpha}_j(v)) = r'^{\alpha}_i(v)$$

Proof. $l'^{\alpha}_j(v)$ is on the path from v to $l'^{\alpha}_i(v)$ in T_l, so this follows trivially from the recursion. The case for r' is symmetric. □

Lemma 3.45. Let $v \in V$, $\alpha \in \{0, 1\}$, and $j_2[v] \leq i < d[v] - 1$, then

$$l^{\alpha}_i(v) = \perp \implies l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \perp \qquad \wedge \qquad r^{\alpha}_i(v) = \perp \implies r'^{\alpha}_i(r^{\alpha}_{i+1}(v)) = \perp$$

Proof. If $l^{\alpha}_i(v) = \perp$ then $\widehat{L}^{\alpha}_i(v) = \emptyset$, so either $l^{\alpha}_{i+1}(v) = \perp$ implying $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \perp$ by the definition of l', or $l^{\alpha}_{i+1}(v) \notin \text{init}(\widehat{L}^{\alpha}_i(v))$ so $d[l^{\alpha}_{i+1}(v)] = i + 1$ and by Lemma 3.43 $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) \in \text{init}(\widehat{L}^{\alpha}_i(l^{\alpha}_{i+1}(v))) \cup \{\perp\} \subseteq \text{init}(\widehat{L}^{\alpha}_i(v)) \cup \{\perp\} = \{\perp\}$ so again $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \perp$. The case for r is symmetric. □

Lemma 3.46 (Crossing lemma)**.** Let $v \in V$, $\alpha \in \{0, 1\}$, and $j_2[v] \leq i < d[v] - 1$.

$$\begin{aligned} l^{\alpha}_i(v) \neq l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) \implies\ & l^{\alpha}_i(v) = l'^{\alpha}_i(m) \wedge r^{\alpha}_i(v) = r'^{\alpha}_i(m) \wedge d[m] = i + 1 \\ & \text{where } m = r^{\alpha}_{i+1}(v) \neq \perp \\ r^{\alpha}_i(v) \neq r'^{\alpha}_i(r^{\alpha}_{i+1}(v)) \implies\ & l^{\alpha}_i(v) = l'^{\alpha}_i(m) \wedge r^{\alpha}_i(v) = r'^{\alpha}_i(m) \wedge d[m] = i + 1 \\ & \text{where } m = l^{\alpha}_{i+1}(v) \neq \perp \end{aligned}$$

Proof. Suppose $l^{\alpha}_i(v) \neq l'^{\alpha}_i(l^{\alpha}_{i+1}(v))$ (the case $r^{\alpha}_i(v) \neq r'^{\alpha}_i(r^{\alpha}_{i+1}(v))$ is symmetrical). Then $l^{\alpha}_i(v) \neq \perp$ by lemma 3.45. Thus there is a last edge $(w, w') \in \widehat{L}^{\alpha}_i(v)$ with $w = l^{\alpha}_i(v)$ and $d[w] \leq i < d[w']$ and a path $P = w' \rightsquigarrow v$.

Now $(w, w') \notin E_{i+1}(v)$ since otherwise by Definition 3.37 $(w, w') \in L^{\alpha}_{i+1}(v)$ and since $w' \rightsquigarrow v$ even $(w, w') \in \widehat{L}^{\alpha}_{i+1}(v)$ implying $l^{\alpha}_i(v) = l^{\alpha}_{i+1}(v)$ and thus $l^{\alpha}_i(v) = l'^{\alpha}_i(l^{\alpha}_{i+1}(v))$ by lemma 3.43, contradicting our assumption.

Since $(w, w') \notin E_{i+1}(v)$, the path P must cross $\widehat{F}_{i+1}(v)$. Let (u, u') be the last edge in $P \cap \widehat{F}_{i+1}(v)$. Then $w' \rightsquigarrow u$ so $d[u] \geq i + 1$ and $(u, u') \notin L^{\alpha}_{i+1}(v)$ since otherwise $d[l^{\alpha}_{i+1}(v)] = i + 1$ and hence by Lemma 3.43 $l^{\alpha}_i(v) = l'^{\alpha}_i(l^{\alpha}_{i+1}(v))$, again contradicting our assumption.

Also, $t^{\alpha}_i(v) \neq t^{\alpha}_{i+1}(v)$ because $t^{\alpha}_i(v) = t^{\alpha}_{i+1}(v)$ would imply $(w, w') \in L^{\alpha}_{i+1}(v) \cup \{\perp\}$ which we have just shown is not the case.

Since $t^{\alpha}_i(v) \neq t^{\alpha}_{i+1}(v)$, then by definition $t^{1-\alpha}_i(v) = t^{1-\alpha}_{i+1}(v)$ and hence $L^{1-\alpha}_{i+1}(v) \subseteq L^{1-\alpha}_i(v)$ and $R^{1-\alpha}_{i+1}(v) \subseteq R^{1-\alpha}_i(v)$, implying $d[w''] \leq i$ for all $w'' \in L^{1-\alpha}_{i+1}(v) \cup R^{1-\alpha}_{i+1}(v)$. Thus, $(u, u') \notin L^{1-\alpha}_{i+1}(v) \cup R^{1-\alpha}_{i+1}(v)$ since $d[u] > i$, and we can conclude that $(u, u') \in \widehat{R}^{\alpha}_{i+1}(v)$.

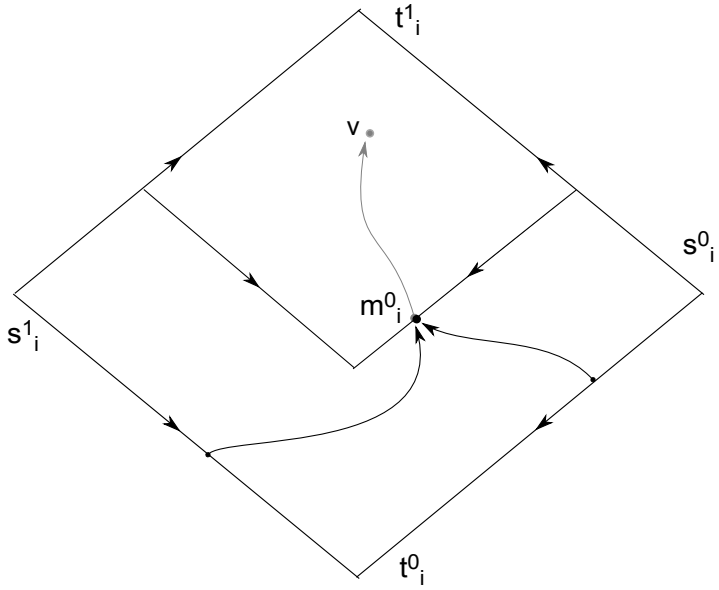


Figure 4: Sometimes the best path from $L_i^0(v)$ to v must go through $R_{i+1}^0(v)$.

But then we can choose P so it goes through (m, m') where $m = r_{i+1}^\alpha(v) \neq \perp$. Now $i+1 \leq d[w'] \leq d[r_{i+1}^\alpha(v)] \leq i+1$ so $d[m] = i+1$.

Let e be the last edge in $\widehat{R}_i^\alpha(v)$ then any path $r_i^\alpha(v) \rightsquigarrow v$ that starts with e crosses $P \cup \widehat{R}_{i+1}^\alpha(v)$, implying that there exists such a path that contains (m, m') and thus $r_i^\alpha(v) = r_i^\alpha(m)$. Since $d[m] = i+1$, then $l_i^\alpha(v) = l_i'^\alpha(m)$ and $r_i^\alpha(v) = r_i'^\alpha(m)$ follows from lemma 3.43. □

Definition 3.47. Let $v \in V$, $\alpha \in \{0,1\}$, and $0 \leq i < d[v]$.

$$m_i^\alpha(v) := \begin{cases} v & \text{if } i+1 = d[v] \\ l_{i+1}^\alpha(v) & \text{if } i+1 < d[v] \wedge r_i^\alpha(v) \neq r_i'^\alpha(r_{i+1}^\alpha(v)) \\ r_{i+1}^\alpha(v) & \text{if } i+1 < d[v] \wedge l_i^\alpha(v) \neq l_i'^\alpha(l_{i+1}^\alpha(v)) \\ m_{i+1}^\alpha(v) & \text{otherwise} \end{cases}$$

Corollary 3.48. Let $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leq i < d[v] - 1$. If $l_i^\alpha(v) \neq l_i'^\alpha(l_{i+1}^\alpha(v))$ or $r_i^\alpha(v) \neq r_i'^\alpha(r_{i+1}^\alpha(v))$ then

$$l_i^\alpha(v) = l_i'^\alpha(m_i^\alpha(v)) \quad \wedge \quad r_i^\alpha(v) = r_i'^\alpha(m_i^\alpha(v)) \quad \wedge \quad d[m_i^\alpha(v)] = i+1$$

Proof. This is just a reformulation of lemma 3.46 in terms of $m_i^\alpha(v)$. □

Lemma 3.49. For any vertex $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leq i < d[v]$

$$l_i^\alpha(v) = l_i'^\alpha(m_i^\alpha(v)) \quad \wedge \quad r_i^\alpha(v) = r_i'^\alpha(m_i^\alpha(v))$$

Proof. The proof is by induction on j, the number of times the "otherwise" case is used before reaching one of the other cases when expanding the recursive definition of $m_i(v)$.

For $j = 0$, either $i + 1 = d[v]$ and the result follows from Lemma 3.43, or $i + 1 < d[v]$ and $l_i(v) \neq l'_i(l_{i+1}(v))$ or $r_i(v) \neq r'_i(r_{i+1}(v))$. In either case we have by Corollary 3.48, that $l^\alpha_i(v) = l'^\alpha_i(m^\alpha_i(v))$ and $r^\alpha_i(v) = r'^\alpha_i(m^\alpha_i(v))$.

For $j > 0$ we have $i + 1 < d[v]$ and $l_i(v) = l'_i(l_{i+1}(v))$ and $r_i(v) = r'_i(r_{i+1}(v))$ and $m_i(v) = m_{i+1}(v)$. By induction we can assume that $l^\alpha_{i+1}(v) = l'^\alpha_{i+1}(m^\alpha_{i+1}(v))$ and $r^\alpha_{i+1}(v) = r'^\alpha_{i+1}(m^\alpha_{i+1}(v))$. Then by Lemma 3.44, $l'^\alpha_i(l^\alpha_{i+1}(v)) = l'^\alpha_i(l'^\alpha_{i+1}(m^\alpha_{i+1}(v))) = l'^\alpha_i(m^\alpha_{i+1}(v)) = l'^\alpha_i(m^\alpha_i(v))$, showing that $l^\alpha_i(v) = l'^\alpha_i(m^\alpha_i(v))$ as desired. The case for r is symmetric. □

Definition 3.50. For any vertex $v \in V$, and $\alpha \in \{0, 1\}$ let

$$M^\alpha[v] := \left\{ i \middle| \, j_2[v] < i < d[v] \wedge m^\alpha_{i-1}(v) \neq m^\alpha_i(v) \right\}$$

$$p^\alpha_m[v] := \begin{cases} \perp & \text{if } M^\alpha[v] = \emptyset \\ m^\alpha_{\max M^\alpha[v]-1}(v) & \text{otherwise} \end{cases}$$

And define T^α_m as the rooted forest over V whose parent pointers are p^α_m.

Theorem 3.51. *There exists a practical RAM data structure that for any good st-decomposition of a graph with n vertices uses $\mathcal{O}(n)$ words of $\mathcal{O}(\log n)$ bits and can answer $l^\alpha_i(v)$ and $r^\alpha_i(v)$ queries in constant time.*

Proof. For any vertex $v \in V$, and $\alpha \in \{0, 1\}$ let

$$D^\alpha_l[v] := \{i| \, v \text{ has a proper ancestor } w \text{ in } T^\alpha_l \text{ with } d[w] = i\}$$
$$D^\alpha_r[v] := \{i| \, v \text{ has a proper ancestor } w \text{ in } T^\alpha_r \text{ with } d[w] = i\}$$

Now, store levelancestor structures for each of T^α_l, T^α_r, and T^α_m, together with $d[v]$, $j_2[v]$, $J_2[v]$, $D^\alpha_l[v]$, $D^\alpha_r[v]$, and $M^\alpha[v]$ for each vertex. Since the height of the st-decomposition is $\mathcal{O}(\log n)$ each of $J_2[v]$, $D^\alpha_l[v]$, $D^\alpha_r[v]$, and $M^\alpha[v]$ can be represented in a single $\mathcal{O}(\log n)$-bit word.

This representation allows us to find $d[m^\alpha_i(v)] = \text{succ}(M^\alpha[v] \cup \{d[v]\}, i)$ in constant time, as well as computing the depth in T^α_m of $m^\alpha_i(v)$. Then using the levelancestor structure for T^α_m we can compute $m^\alpha_i(v)$ in constant time.

Similarly, this representation of the $D^\alpha_l[v]$ set lets us compute the depth in T^α_l of $l'^\alpha_i(v)$ in constant time, and with the levelancestor structure that lets us compute $l'^\alpha_i(v)$ in constant time. A symmetric argument shows that we can compute $r'^\alpha_i(v)$ in constant time.

Finally, lemma 3.49 says we can compute $l^\alpha_i(v)$ and $r^\alpha_i(v)$ in constant time given constant-time functions for l', r', and m. □

4 Acyclic planar In- out- graphs

For an in-out-graph G we have a source, s, that can reach all vertices of outdegree 0. Given such a source, s, we may assign all vertices a colour: A vertex is green if it can be reached from s, and red otherwise. We may also colour the directed edges: (u, v) has the same colour as its endpoints, or is a blue edge in the special case where u is red and v is green. Our idea is to keep the colouring and flip all non-green edges, thus obtaining a single source graph H with source s. (Any vertex was either green and thus already reachable from s, or could reach some target t, and is reachable from s in H via the first green vertex on its path to t.)

Consider the single source reachability data structure for the red-green graph, H. This alone does not suffice to determine reachability in G, but it does when endowed with a few extra words per vertex:

M1 A red vertex u must remember the additional information of the best green vertices $BestGreen(u)$ on its own parent frame it can reach. There are at most 4 such vertices, one for each disegment.

M2 Information about paths from a red to a green vertex in the same component. See Section 4.1.

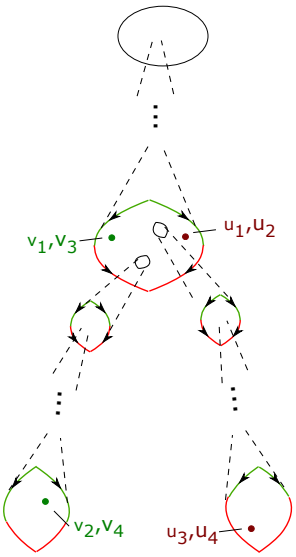
M3 Information about paths from a red vertex in some component C to a green vertex in an ancestor component of C. See Section 4.2.

Given a green vertex v, we know for each ancestral frame segment the best vertex that can reach v. For a red vertex u, given a segment p on an ancestral frame to u, we have information about the best vertex on p that may reach u in H via "ingoing" edges, that is, an edge from the corresponding $\widehat{F}_i(u)$. If that best vertex is red, then it is the best vertex on p that u can reach, again, from the "inside".

We may now case reachability based on the colour of nodes:

- For green u and red v, $\text{reach}_G(u, v)$ = No.
- For green vertices u, v, $\text{reach}_G(u, v) = \text{reach}_H(u, v)$
- For red vertices u, v, $\text{reach}_G(u, v) = \text{reach}_H(v, u)$
- When u is red and v is green, to determine $\text{reach}_G(u, v)$ we need more work. It will depend on where in the hierarchy of components, u and v reside.

When u is red and v is green, there are the following cases.



1. $c[u] = c[v]$. There may be a path from u to v:
   - Via a green vertex w in the parent frame of u. For each candidate $w \in BestGreen(u)$, try $\text{reach}_H(w, v)$. (See M1).
   - Staying within the frame, that is, $\text{reach}_{c[u]}(u, v)$. To handle this case we need to store more information, see Section 4.1.
2. $c[u] \prec c[v]$. There may be a path from u to v:
   - Via a green vertex w in the parent frame of u, $\text{reach}_H(w, v)$. (See M1).
   - Via a green vertex w, where $c[w] = c[u]$, then $\text{reach}_G(u, w)$ is in case 1 above. v knows the at most 4 such ws from the single source structure.
3. $c[u] \succ c[v]$. There may be a path from u to v:
   - Via a red edge (w', w) in G with $c[w] \preceq c[v] \prec c[w'] \preceq c[u]$. That is, in the single-source structure for H, u can find its best vertex w for each disegment of the parent frame of $c[v]$. For a path via that disegment to exist, w must be red, and $\text{reach}_G(w, v)$, which is in case 1 or 2 above, must return true.
   - Via a blue edge (w', w) with $c[w] \preceq c[v] \prec c[w'] \preceq c[u]$. We handle this case in Section 4.2.
4. $c[u], c[v] \succ N$, where $N = \text{lca}(c[u], c[v])$. A path from u to v must go:
   - Via w, $c[w] \preceq N$, then $\text{reach}_G(u, w)$ is in case 3 above. v computes at most 4 such ws from the single source structure, and note that all the vertices that v computes must be green.

4.1 Intracomponental blue edges

Consider the set of "blue" edges (a, b) from G where both the red vertex a and green b reside in some given component in the s-t-decomposition of H.

Lemma 4.1. We may assign to each vertex ≤ 2 numbers, such that if red u remembers $i, j \in \mathbb{N}$ and green v remembers $l, r \in \mathbb{N}$, then u can reach v if and only if $i \leq l \leq j$ or $i \leq r \leq j$ or $\min\{l, r\} \leq j < i$ or $j < i \leq \max\{l, r\}$.

Proof. The key observation is that we may enumerate all blue edges $b_0 = (u_0, v_0), \ldots b_i = (u_m, v_m)$ such that any red vertex can reach a segment of their endpoints, $v_i, \ldots, v_j$. Namely, the blue edges form a minimal cut in the planar graph which separates the red from the green vertices, and this cut induces a cyclic order. In this order, each red vertex may reach a segment of blue edges, and each green vertex may reach a segment of blue edge endpoints. Thus, the blue edge endpoints reachable from a given red vertex (through any path) is a union of overlapping segments, which is again a segment.

Now each red vertex remembers the indices of the first v_i and last v_j blue edge endpoint it may reach. For a green vertex v, the s-t-subgraph with v as target has a delimiting face consisting of two paths, P and Q. v remembers the indices l, r of the latest blue edge endpoints $v_l \in P$ and $v_r \in Q$, if they exist. Clearly, if l or r is within range, u may reach v. Contrarily, if u may reach v, it must do so via some vertex v' on $P \cup Q$. But then v' must be able to reach v_l or v_r, and thus, l or r is within range. □

4.2 Intercomponental blue edges

For any red vertex u, if a blue edge (u', v) reachable from u is separated u by a frame, then one of the best red vertices on that frame can reach u'. So let each red vertex remember the best ≤ 4 blue edges it can reach on its own frame. Then we can define 4 bitmasks $\{B^\beta(u)\}_{0 \leq \beta \leq 3}$ such that for any i finding the highest 1-bit $\leq i$ in each, gives at most 4 levels such that the best red vertices reachable from u on those levels together know the best blue edges for u.

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005.

[2] S. Alstrup, J. P. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64:161–164, 1997.

[3] S. Arikati, D.Z. Chen, L.P. Chew, G. Das, M. Smid, and C.D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *ESA '96*, pages 514–528, 1996.

[4] D.Z. Chen and J. Xu. Shortest path queries in planar graphs. In *STOC '00*, pages 469–478, 2000.

[5] H. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *WG '96*, pages 151–165, 1996.

[6] H. Djidjev, G. Panziou, and C. Zaroliagis. Computing shortest paths and distances in planar graphs. In *ICALP '91*, pages 327–339, 1991.

[7] H. Djidjev, G. Panziou, and C. Zaroliagis. Fast algorithms for maintaining shortest paths in outerplanar and planar digraphs. In *FCT '95*, pages 191–200, 1995.

[8] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[9] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Inf. Process. Lett.*, 3(3):75–77, 1975.

[10] K. Kawarabayashi, P.N. Klein, and C. Sommer. Linear-space approximate distance oracles for planar, bounded-genus, and minor-free graphs. In *ALP '11*, pages 135–146, 2011.

[11] K. Kawarabayashi, C. Sommer, and M. Thorup. More compact oracles for approximate distances in undirected planar graphs. In *SODA '13*, pages 550–563, 2013.

[12] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

[13] P. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *SODA '02*, pages 820–827, 2002.

[14] P. B. Miltersen. Lower bounds for static dictionaries on rams with bit operations but no multiplication. In *ICALP '96*, pages 442–453. 1996.

[15] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *SODA '12*, pages 209–222, 2012.

[16] M. Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. Announced at FOCS'08. See also arXiv:1010.3783.

[17] R. Tamassia and I.G. Tollis. Dynamic reachability in planar digraphs with one source and one sink. *Theor. Comput. Sci.*, 119(2):331–343, 1993.

[18] R. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1972.

[19] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[20] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):183–192, 2005. Announced at STOC'01.

Online Bipartite Matching with Amortized $\mathcal{O}(\log^2 n)$ Replacements

Aaron Bernstein[1], Jacob Holm[*2], and Eva Rotenberg[†3]

[1]Technical University of Berlin, bernstei@gmail.com
[2]University of Copenhagen (DIKU), jaho@di.ku.dk
[3]Technical University of Denmark, erot@dtu.dk

May 7, 2018

Abstract

In the online bipartite matching problem with replacements, all the vertices on one side of the bipartition are given, and the vertices on the other side arrive one by one with all their incident edges. The goal is to maintain a maximum matching while minimizing the number of changes (replacements) to the matching. We show that the greedy algorithm that always takes the shortest augmenting path from the newly inserted vertex (denoted the SAP protocol) uses at most amortized $\mathcal{O}(\log^2 n)$ replacements per insertion, where n is the total number of vertices inserted. This is the first analysis to achieve a polylogarithmic number of replacements for *any* replacement strategy, almost matching the $\Omega(\log n)$ lower bound. The previous best strategy known achieved amortized $\mathcal{O}(\sqrt{n})$ replacements [Bosek, Leniowski, Sankowski, Zych, FOCS 2014]. For the SAP protocol in particular, nothing better than then trivial $\mathcal{O}(n)$ bound was known except in special cases. Our analysis immediately implies the same upper bound of $\mathcal{O}(\log^2 n)$ reassignments for the capacitated assignment problem, where each vertex on the static side of the bipartition is initialized with the capacity to serve a number of vertices.

We also analyze the problem of minimizing the maximum server load. We show that if the final graph has maximum server load L, then the SAP protocol makes amortized $\mathcal{O}(\min\{L\log^2 n, \sqrt{n}\log n\})$ reassignments. We also show that this is close to tight because $\Omega(\min\{L, \sqrt{n}\})$ reassignments can be necessary.

[*]This research is supported by Mikkel Thorup's Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

[†]This research was partly conducted during the third author's time as a PhD student at University of Copenhagen.

1 Introduction

In the online bipartite matching problem, the vertices on one side are given in advance (we call these the servers S), while the vertices on the other side (the clients C) arrive one at a time with all their incident edges. In the standard online model the arriving client can only be matched immediately upon arrival, and the matching cannot be changed later. Because of this irreversibility, the final matching might not be maximum; no algorithm can guarantee better than a $(1 - 1/e)$-approximation [22]. But in many settings the irreversibility assumption is too strict: rematching a client is expensive but not impossible. This motivates the online bipartite matching problem with replacements, where the goal is to at all times match as many clients as possible, while minimizing the number of changes to the matching. Applications include hashing, job scheduling, web hosting, streaming content delivery, and data storage; see [8] for more details.

In several of the applications above, a server can serve multiple clients, which raises the question of online bipartite *assignment* with reassignments. There are two ways of modeling this:

Capacitated assignments. Each server s comes with the capacity to serve some number of clients $u(s)$, where each $u(s)$ is given in advance. Clients should be assigned to a server, and at no times should the capacity of a server be exceeded. There exists an easy reduction showing that this problem is equivalent to online matching with replacements [2]. A more formal description is given in Section 6.1.

Minimize max load. There is no limit on the number of clients a server can serve, but we want to (at all times) distribute the clients as "fairly" as possible, while still serving all the clients. Defining the load on a server as the number of clients assigned to it, the task is to, at all times, minimize the maximum server load — with as few reassignments as possible. A more formal description is given in Section 6.2

While the primary goal is to minimize the number of replacements, special emphasis has been placed on analyzing the *SAP* protocol in particular, which always augments down a shortest augmenting path from the newly arrived client to a free server (breaking ties arbitrarily). This is the most natural replacement strategy, and shortest augmenting paths are already of great interest in graph algorithms: they occur in for example in Dinitz' and Edmonds and Karp's algorithm for maximum flow [9, 10], and in Hopcroft and Karp's algorithm for maximum matching in bipartite graphs [19].

Throughout the rest of the paper, we let n be the number of clients in the final graph, and we consider the *total* number of replacements during the entire sequence of insertions; this is exactly n times the amortized number of replacements. The reason for studying the vertex-arrival model (where each client arrives with all its incident edges) instead of the (perhaps more natural) edge-arrival model is the existence of a trivial lower bound of $\Omega(n^2)$ total replacements in this model: Start with a single edge, and maintaining at all times that the current graph is a path, add edges to alternating sides of the path. Every pair of insertions cause the entire path to be augmented, leading to a total of $\sum_{i=1}^{n/2} i \in \Omega(n^2)$ replacements.

1.1 Previous work

The problem of online bipartite matchings with replacements was introduced in 1995 by Grove, Kao, Krishnan, and Vitter [13], who showed matching upper and lower bounds of $\Theta(n \log n)$ replacements

for the case where each client has degree two. In 2009, Chadhuri, Daskalakis, Kleinberg, and Lin [8] showed that for any arbitrary underlying bipartite graph, if the client vertices arrive in a random order, the expected number of replacements (in their terminology, the *switching cost*) is $\Theta(n \log n)$ using SAP, which they also show is tight. They also show that if the bipartite graph remains a forest, there exists an algorithm (not SAP) with $\mathcal{O}(n \log n)$ replacements, and a matching lower bound. Bosek, Leniowski, Sankowski and Zych later analyzed the SAP protocol for forests, giving an upper bound of $\mathcal{O}(n \log^2 n)$ replacements [6], later improved to the optimal $\mathcal{O}(n \log n)$ total replacements [7]. For general bipartite graphs, no analysis of SAP is known that shows better than the trivial $\mathcal{O}(n^2)$ total replacements. Bosek et al. [5] showed a different algorithm that achieves a total of $\mathcal{O}(n\sqrt{n})$ replacements. They also show how to implement this algorithm in total time $\mathcal{O}(m\sqrt{n})$, which matches the best performing combinatorial algorithm for computing a maximum matching in a static bipartite graph (Hopcroft and Karp [19]).

The lower bound of $\Omega(\log n)$ by Grove et al. [13] has not been improved since, and is conjectured by Chadhuri et al. [8] to be tight, even for SAP, in the general case. We take a giant leap towards closing that conjecture.

For the problem of minimizing maximum load, [15] and [2] showed an approximation solution: with only $\mathcal{O}(1)$ amortized changes per client insertion they maintain an assignment $\mathcal{A}$ such that at all times the maximum load is within a factor of 8 of optimum.

The model of online algorithms with replacements – alternatively referred to as online algorithms with recourse – has also been studied for a variety of problems other than matching. The model is similar to that of online algorithms, except that instead of trying to maintain the best possible approximation without making any changes, the goal is to maintain an optimal solution while making as few changes to the solution as possible. This model encapsulates settings in which changes to the solution are possible but expensive. The model originally goes back to online Steiner trees [20], and there have been several recent improvements for online Steiner tree with recourse [14, 17, 24, 25]. There are many papers on online scheduling that try to minimize the number of job reassignments [1, 11, 26, 27, 29, 31]. The model has also been studied in the context of flows [15, 31], and there is a very recent result on online set cover with recourse [16].

1.2 Our results

Theorem 1. *SAP makes at most $\mathcal{O}(n \log^2 n)$ total replacements when n clients are added.*

This is a huge improvement of the $\mathcal{O}(n\sqrt{n})$ bound by [5], and is only a log factor from the lower bound of $\Omega(n \log n)$ by [13]. It is also a huge improvement of the analysis of SAP; previously no better upper bound than $\mathcal{O}(n^2)$ replacements for SAP was known. To attain the result we develop a new tool for analyzing matching-related properties of graphs (the balanced flow in Sections 3 and 4) that is quite general, and that we believe may be of independent interest.

Although SAP is an obvious way of serving the clients as they come, it does not immediately allow for an efficient implementation. Finding an augmenting path may take up to $\mathcal{O}(m)$ time, where m denotes the total number of edges in the final graph. Thus, the naive implementation takes $\mathcal{O}(mn)$ total time. However, short augmenting paths can be found substantially faster, and using the new analytical tools developed in this paper, we are able to exploit this in a data structure that finds the augmenting paths efficiently:

Theorem 2. *There is an implementation of the SAP protocol that runs in total time $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$.*

Note that this is only an $\mathcal{O}(\sqrt{\log n})$ factor from the offline algorithm of Hopcroft and Karp [19]. This offline algorithm had previously been matched in the online setting by the algorithm of Bosek et al. [5], which has total running time $O(m\sqrt{n})$. Our result has the advantage of combining multiple desired properties in a single algorithm: few replacements ($O(n\log^2(n))$ vs. $O(n^{1.5})$ in [5]), fast implementation ($\mathcal{O}(m\sqrt{n}\sqrt{\log n})$ vs. $\mathcal{O}(m\sqrt{n})$ in [5]), and the most natural augmentation protocol (shortest augmenting path).

Extending our result to the case where each server can have multiple clients, we use that the capacitated assignment problem is equivalent to that of matching (see Section 6.1 to obtain:

Theorem 3. *SAP uses at most $\mathcal{O}(n\log^2 n)$ reassignments for the capacitated assignment problem, where n is the number of clients.*

In the case where we wish to minimize the maximum load, such a small number of total reassignments is not possible. Let $\textsc{opt}(G)$ denote the minimum possible maximum load in graph G. We present a lower bound showing that when $\textsc{opt}(G) = L$ we may need as many as $\Omega(nL)$ reassignments, as well as a nearly matching upper bound.

Theorem 4. *For any positive integers n and $L \leq \sqrt{n/2}$ divisible by 4 there exists a graph $G = (C \cup S, E)$ with $|C| = n$ and $\textsc{opt}(G) = L$, along with an ordering in which the clients in C are inserted, such that any algorithm for the exact online assignment problem requires a total of $\Omega(nL)$ changes. This lower bound holds even if the algorithm knows the entire graph G in advance, as well as the order in which the clients are inserted.*

Theorem 5. *Let C be the set of all clients inserted, let $n = |C|$, and let $L = \textsc{opt}(G)$ be the minimum possible maximum load in the final graph $G = (C \cup S, E)$. SAP at all times maintains an optimal assignment while making a total of $\mathcal{O}(n\min\{L\log^2 n, \sqrt{n}\log n\})$ reassignments.*

1.3 High level overview of techniques

Consider the standard setting in which we are given the entire graph from the beginning and want to compute a maximum matching. The classic shortest-augmenting paths algorithm constructs a matching by at every step picking a shortest augmenting path in the graph. We now show a very simple argument that the total length of all these augmenting paths is $\mathcal{O}(n\log n)$. Recall the well-known fact that if all augmenting paths in the matching have length $\geq h$, then the current matching is at most $2n/h$ edges from optimal [19]. Thus the algorithm augments down at most $2n/h$ augmenting paths of length $\geq h$. Let $P_1, P_2, ..., P_k$ denote all the paths augmented down by the algorithm in decreasing order of $|P_i|$; then $k \leq n$, and $|P_i| = h$ implies $i \leq 2n/h$. But then $|P_i| \leq 2n/i$, so $\sum_{1\leq i\leq k}|P_i| \leq 2n\sum_{1\leq i\leq k}\frac{1}{i} = 2n(\ln(k) + \mathcal{O}(1)) = \mathcal{O}(n\log k) = \mathcal{O}(n\log n)$.

In the online setting, the algorithm does not have access to the entire graph. It can only choose the shortest augmenting path from the arriving client c. We are nonetheless able to show a similar bound for this setting:

Lemma 6. *Consider the following protocol for constructing a matching: For each client c in arbitrary order, augment along the shortest augmenting path from c (if one exists). Given any h, this protocol augments down a total of at most $4n\ln(n)/h$ augmenting paths of length $> h$.*

The proof of our main theorem then follows directly from the lemma.

Proof of Theorem 1. Note that the SAP protocol exactly follows the condition of Lemma 6. Now, Given any $0 \leq i \leq \log_2(n) + 1$, we say that an augmenting path is at level i if its length is in the interval $[2^i, 2^{i+1})$. By Lemma 6, the SAP protocol augments down at most $4n\ln(n)/2^i$ paths of level i. Since each of those paths contains at most 2^{i+1} edges, the total length of augmenting paths of level i is at most $8n\ln(n)$. Summing over all levels yields the desired $\mathcal{O}(n\log^2 n)$ bound. □

The entirety of Sections 3 and 4 is devoted to proving Lemma 6. Previous algorithms attempted to bound the total number of reassignments by tracking how some property of the matching M changes over time. For example, the analysis of Gupta et al. [15] keeps track of changes to the "height" of vertices in M, while the algorithm with $\mathcal{O}(n\sqrt{n})$ reassignments [5] takes a more direct approach, and uses a non-SAP protocol whose changes to M depend on how often each particular client has already been reassigned.

Unfortunately such arguments have had limited success because the matching M can change quite erratically. This is especially true under the SAP protocol, which is why it has only been analyzed in very restrictive settings [6, 8, 13]. We overcome this difficulty by showing that it is enough to analyze how new clients change the structure of the graph $G = (C \cup S, E)$, without reference to any particular matching.

Intuitively, our analysis keeps track of how "necessary" each server s is (denoted $\alpha(s)$ below). So for example, if there is a complete bipartite graph with 10 servers and 10 clients, then all servers are completely necessary. But if the complete graph has 20 servers and 10 clients, then while any matching has 10 matched servers and 10 unmatched ones, it is clear that if we abstract away from the particular matching every server is 1/2-necessary. Of course in more complicated graphs different servers might have different necessities, and some necessities might be very close to 1 (say $1 - 1/n^{2/3}$). Note that server necessities depend only on the graph, not on any particular matching. Note also that our algorithm never computes the server necessities, as they are merely an analytical tool.

We relate necessities to the number of reassignments with 2 crucial arguments. **1.** Server necessities only increase as clients are inserted, and once a server has $\alpha(s) = 1$, then regardless of the current matching, no future augmenting path will go through s. **2.** If, *in any matching*, the shortest augmenting path from a new client c is long, then the insertion of c will increase the necessity of servers that already had high necessity. We then argue that this cannot happen too many times before the servers involved have necessity 1, and thus do not partake in any future augmenting paths.

1.4 Paper outline

In Section 2, we introduce the terminology necessary to understand the paper. In Section 3, we introduce and reason about the abstraction of a balanced server flow, a number that reflects the necessity of each server. In Section 4, we use the balanced server flow to prove Lemma 6, which proves our main theorem that SAP makes a total of $\mathcal{O}(n\log^2 n)$ replacements. In Section 5, we give an efficient implementation of SAP. Finally, in Section 6, we present our results on capacitated online assignment, and for minimizing maximum server load in the online assignment problem.

2 Preliminaries and notation

Let (C, S) be the vertices, and E be the edges of a bipartite graph. We call C the *clients*, and S the *servers*. Clients arrive, one at a time, and we must maintain an explicit maximum matching of the clients. For simplicity of notation, we assume for the rest of the paper that $C \neq \emptyset$. For any vertex v, let $N(v)$ denote the neighborhood of v, and for any $V \subseteq C \cup S$ let $N(V) = \bigcup_{v \in V} N(v)$.

Theorem 7 (Halls Marriage Theorem [18])**.** *There is a matching of size $|C|$ if and only if $|K| \leq |N(K)|$ for all $K \subseteq C$.*

Definition 8. Given any matching in a graph $G = (C \cup S, E)$, an alternating path is one which alternates between unmatched and matched edges. An augmenting path is an alternating path that starts and ends with an unmatched vertex. Given any augmenting path P, "flipping" the matched status of every edge on P gives a new larger matching. We call this process *augmenting down P*.

Denote by SAP the algorithm that upon the arrival of a new client c augments down the shortest augmenting path from c; ties can be broken arbitrarily, and if no augmenting path from c exists the algorithm does nothing. Chaudhuri et al. [8] showed that if the final graph contains a perfect matching, then the SAP protocol also returns a perfect matching. We now generalize this as follows

Observation 9. *Because of the nature of augmenting paths, once a client c or a server s is matched by the SAP protocol, it will remain matched during all future client insertions. On the other hand, if a client c arrives and there is no augmenting path from c to a free server, then during the entire sequence of client insertions c will never be matched by the SAP protocol; no alternating path can go through c because it is not incident to any matched edges.*

Lemma 10. *The SAP protocol always maintains a maximum matching in the current graph $G = (C \cup S, E)$.*

Proof. Consider for contradiction the first client c such that after the insertion of c, the matching M maintained by the SAP protocol is not a maximum matching. Let C be the set of clients before c was inserted. Since M is maximum in the graph $G = (C \cup S, E)$ but not in $G' = (C \cup S \cup \{c\}, E)$, it is clear that c is matched in the maximum matching M' of G' but not in M. But this contradicts the well known property of augmenting paths that the symmetric difference $M \oplus M'$ contains an augmenting path in M from c to a free server. □

3 The server flow abstraction

3.1 Defining the Server Flow

We now formalize the notion of server necessities from Section 1.3 by using a flow-based notation. The necessity of a server s will be the value $\alpha(s) \in [0, 1]$ of a balanced server flow α: We will now go on to define a server flow, define what it means for a server flow to be balanced, and then, show that the balanced server flow is unique.

Definition 11. Given any graph $G = (C \cup S, E)$, define a *server flow* α as any map from S to the nonnegative reals such that there exist nonnegative $(x_e)_{e \in E}$ with:

$$\forall c \in C : \sum_{s \in N(c)} x_{cs} = 1 \qquad \forall s \in S : \sum_{c \in N(s)} x_{cs} = \alpha(s)$$

We say that such a set of x-values *realize* the server flow.

A server flow can be thought of as a fractional assignment from C to S; note, however, that is is not necessarily a fractional matching, since servers may have a load greater than 1. Note also that the same server flow may be realized in more than one way. Furthermore, if $|N(c)| = 0$ for some $c \in C$ then $\sum_{s \in N(c)} x_{cs} = 0 \neq 1$, so no server flow is possible. So suppose (unless otherwise noted) that $|N(c)| \geq 1$ for all $c \in C$.

The following theorem can be seen as a generalization of Hall's Marriage Theorem:

Lemma 12. *If* $\max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} = \frac{p}{q}$*, then there exists a server flow where every server* $s \in S$ *has* $\alpha(s) \leq \frac{p}{q}$.

Proof. Let C^* be the original set C but with q copies of each client. Similarly let S^* contain p copies of each server, and let E^* consist of all pq edges between copies of the endpoints of each edge in E.

Now let $K^* \subseteq C^*$, and let $K \subseteq C$ be the originals that the vertices in K^* are copies of. Then $|K^*| \leq q\,|K| \leq p\,|N(K)| = |N(K^*)|$, so the graph $(C^* \cup S^*, E^*)$ satisfies Hall's theorem and thus it has some matching M in which every client in C^* is matched. Now, for $cs \in E$ let

$$x_{cs} = \frac{1}{q} \left| \left\{ c^* s^* \in M \;\middle|\; c^* \text{ is a copy of } c \text{ and } s^* \text{ is a copy of } s \right\} \right|$$

Since for each $c \in C$ all q copies of c are matched, $\sum_{s \in N(c)} x_{cs} = \frac{q}{q} = 1$ for all $c \in C$. Similarly, since for each $s \in S$ at most p copies of s are matched, $\sum_{c \in N(s)} x_{cs} \leq \frac{p}{q}$. Thus, $(x_e)_{e \in E}$ realizes the desired server flow. □

Definition 13. We say that a server flow α is *balanced*, if additionally:

$$\forall c \in C, s \in N(c) \setminus A(c) : x_{cs} = 0 \qquad \text{where } A(c) = \arg\min_{s \in N(c)} \alpha(s)$$

That is, if each client only sends flow to its least loaded neighbours.

We call the set $A(c)$ the *active* neighbors of c, and we call an edge cs *active* when $s \in A(c)$. We extend the definition to sets of clients in the natural way, so for $K \subseteq C$, $A(K) = \bigcup_{c \in K} A(c)$.

3.2 Uniqueness of Loads in a Balanced Server Flow

Note that while there may be more than one server flow, we will show that the balanced server flow α is unique, although there may be many possible x-values x_{cs} that realize α.

Lemma 14. *A unique balanced server flow exists if and only if* $|N(c)| \geq 1$ *for all* $c \in C$.

Clearly, it is necessary for all clients to have at least one neighbor for a server flow to exist, so the "only if" part is obvious. We dedicate the rest of this section to proving that this condition is sufficient. In fact, we provide two different proofs of uniqueness, the first of which is simpler but provides less intuition for what the unique $\alpha(s)$ values signify about the structure of the graph.

3.2.1 Short proof of Lemma 14 via convex optimization

It is not hard to prove uniqueness by showing that a balanced server flow corresponds to the solution to a convex program[1]. Consider the convex optimization problem where the constraints are those

[1]The authors thank Seffi Naor for pointing this out to us.

of a *not necessarily balanced* server flow (Definition 11), and the objective function we seek to minimize is the sum of the squares of the server loads.

To be precise, the convex program contains a variable α_s for each server $s \in S$, and a variable x_{cs} for each edge (c, s) in the graph. Its objective is to minimize the function $\sum_{s\in S} \alpha_s^2$ subject to the constraints:

$$0 \le x_{cs} \le 1 \qquad \forall c \in C : \sum_{s\in N(c)} x_{cs} = 1 \qquad \forall s \in S : \sum_{c\in N(s)} x_{cs} = \alpha_s$$

It is easy to check that because we introduce a separate variable α_s for each server load, the objective function is strictly convex, so the convex program has a unique minimum with respect to the server loads α_s (but not the edge flows).

We now observe that this unique solution is a *balanced* server flow: the constraints of the convex program ensure that it is a server flow, and were it not balanced, there would be some client c that sends non-zero flow to both s and s' where $\alpha(s) < \alpha(s')$, which would be a contradiction because we can decrease the objective function by increasing x_{cs} and decreasing $x_{cs'}$. We have thus proved the existence of a balanced server flow.

We must now prove uniqueness, i.e. that all balanced server flows have the same server loads. We will do this by showing that any balanced server flow optimizes the objective function of the convex function. There are many standard approaches for proving this claim, but the simplest one we know of is based on existing literature on load balancing with selfish agents. In particular, we rely on the following simple auxiliary lemma, which is a simplified version of Lemma 2.2 in [30].

Lemma 15. *Consider any balanced server flow x_{cs}, let $\alpha_s = \sum_{c\in C} x_{cs}$ be the server flow of s. Let x'_{cs} be* any *feasible server flow, and let $\alpha'_s = \sum_{c\in C} x'_{cs}$ be the resulting server loads. Then, we always have:*

$$\sum_{s\in S} \alpha_s^2 \le \sum_{s\in S} \alpha_s \alpha'_s$$

Proof. For any client c, let $\mu(c)$ (μ for minimum) be the minimum server load neighboring c in the balanced solution x_{cs}. That is, $\mu(c) = \min_{s\in N(c)} \alpha_s$. We then have

$$\sum_{s\in S} \alpha_s^2 = \sum_{s\in S}\sum_{c\in C} x_{cs}\alpha_s = \sum_{c\in C}\sum_{s\in S} x_{cs}\alpha_s = \sum_{c\in C}\sum_{s\in S} x_{cs}\mu(c) = \sum_{c\in C} \mu(c),$$

where the last inequality follows from the fact that each client sends one unit of flow, and the before-last inequality follows from the fact that the flow is balanced, so for any edge $(c, s) \in E$ with $x_{cs} \neq 0$ we have $\alpha_s = \mu(c)$.

From the definition of $\mu(c)$ it follows that for *any* edge $(c, s) \in E$, we have $\alpha_s \ge \mu(c)$. This yields:

$$\sum_{s\in S} \alpha'_s\alpha_s = \sum_{s\in S}\sum_{c\in C} x'_{cs}\alpha_s = \sum_{c\in C}\sum_{s\in S} x'_{cs}\alpha_s \ge \sum_{c\in C}\sum_{s\in S} x'_{cs}\mu(c) = \sum_{c\in C} \mu(c).$$

We thus have $\sum_{s\in S} \alpha_s^2 = \sum_{c\in C} \mu(c)$ and $\sum_{s\in S} \alpha'_s\alpha_s \ge \sum_{c\in C} \mu(c)$, which yields the lemma. □

We now argue that any balanced flow is an optimal solution to the convex program, and is thus unique. Consider any balanced flow with loads α_s. To show that α_s is optimum, we need to

show that for any feasible solution α'_s we have $\sum_{s\in S}\alpha_s^2 \le \sum_{s\in S}(\alpha')_s^2$. Equivalently, let $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha'}$ be the vectors of server loads in the two solutions. We want to show that $\|\boldsymbol{\alpha}\| \le \|\boldsymbol{\alpha'}\|$. This follows trivially from Lemma 15, which is equivalent to $\|\boldsymbol{\alpha}\|^2 \le \boldsymbol{\alpha}\cdot\boldsymbol{\alpha'}$.

3.2.2 Longer combinatorial proof of uniqueness

Although the reduction to convex programming is the most direct proof of uniqueness, it has the disadvantage of not providing any insight into what the unique $\alpha(s)$ values actually correspond to. We thus provide a more complicated combinatorial proof which shows that the $\alpha(s)$ correspond to a certain hierarchical decomposition of the graph.

The following lemma will help us upper and lower bound the sum of flow to a subset of servers.

Lemma 16. *If α is a balanced server flow, then*

$$\forall T \subseteq S : \Big|\{c \in C \mid A(c) \subseteq T\}\Big| \le \sum_{s\in T}\alpha(s) \le \Big|\{c \in C \mid A(c) \cap T \neq \emptyset\}\Big|$$

Proof. The first inequality is true because each client in the first set contributes exactly one to the sum (but there may be other contributions). The second inequality is true because every client contributes exactly one to $\sum_{s\in S}\alpha(s)$, and the inequality counts every client that contributes anything to $\sum_{s\in T}\alpha(s)$ as contributing one. □

The first step to proving that every graph has a unique server flow α is to show that the maximum value $\widehat{\alpha} = \max_{s\in S}\alpha(s)$ is uniquely defined. We start by showing that the generalization of Hall's Marriage Theorem from Lemma 12 is "tight" for a balanced server flow in the sense that there does indeed exist a set of p clients with neighbourhood of size q realizing the maximum α-value $\frac{p}{q}$. In fact, the maximally necessary servers and their active neighbours (defined below) form such a pair of sets:

Lemma 17. *Let α be a balanced server flow, let $\widehat{\alpha} = \max_{s\in S}\alpha(s)$ be the maximal necessity, let $\widehat{S} = \{s \in S \mid \alpha(s) = \widehat{\alpha}\}$ be the maximally necessary servers, and let $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \emptyset\}$ be their active neighbours. Then $N(\widehat{K}) = \widehat{S}$ and $|\widehat{K}| = \widehat{\alpha}\,|\widehat{S}|$.*

Proof. Let $K = \{c \in C \mid A(c) \subseteq \widehat{S}\}$, and note that $K \subseteq \widehat{K}$. However, we also have $\widehat{K} \subseteq K$: By definition of $\widehat{S}$, and since we assume the server flow is balanced, $\widehat{K} \neq \emptyset$, and for every $c \in \widehat{K}$, $N(c) = A(c) \subseteq \widehat{S}$. Thus, $K = \widehat{K}$ and $N(\widehat{K}) = \widehat{S}$. Now, note that by Lemma 16

$$|\widehat{K}| = |K| \le \widehat{\alpha}\,|\widehat{S}| \le |\widehat{K}|\,.$$ □

We can thus show that $\widehat{\alpha}$ exactly equals the maximal quotient $\frac{|K|}{|N(K)|}$ over subsets K of clients.

Lemma 18. *Let α be a balanced server flow, and let $\widehat{\alpha} = \max_{s\in S}\alpha(s)$. Then*

$$\widehat{\alpha} = \max_{\emptyset\subset K\subseteq C}\frac{|K|}{|N(K)|}$$

Furthermore, for any $K \subseteq C$, if $|K| = \widehat{\alpha}\,|N(K)|$, then $\alpha(s) = \widehat{\alpha}$ for all $s \in N(K)$.

Proof. By definition of server flow, for $K \subseteq C$, $|K| \leq \sum_{s \in N(K)} \alpha(s) \leq \widehat{\alpha}\,|N(K)|$, so $|K| \leq \widehat{\alpha}\,|N(K)|$. Let $\widehat{K}$ be defined as in Lemma 17. Then $\widehat{\alpha} = \frac{|\widehat{K}|}{|N(\widehat{K})|} \leq \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} \leq \widehat{\alpha}$. Finally, if $|K| = \sum_{s \in N(K)} \alpha(s) = \widehat{\alpha}\,|N(K)|$ then $\alpha(s) \leq \widehat{\alpha}$ for all $s \in S$ implies $\alpha(s) = \widehat{\alpha}$ for $s \in N(K)$. □

Corollary 19. *If* $\max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|} = \frac{|C|}{|S|}$ *there is a unique balanced server flow.*

Proof. By Lemma 12 there exists a server flow with $\alpha(s) \leq \frac{|C|}{|S|}$ for all $s \in S$. Since $\sum_{s \in S} \alpha(s) = |C|$, any such flow must actually have $\alpha(s) = \frac{|C|}{|S|}$ for all $s \in S$, and be balanced. Uniqueness follows from Lemma 18. □

We are now ready to give a combinatorial proof of uniqueness. We will do so by showing that the $\alpha(s)$ in fact express a very nice structural property of the graph, which can be thought of as a hierarchy of "tightness" for the Hall constraint. As shown in Lemma 18, the maximum α value $\widehat{\alpha}$ corresponds to the tightest Hall constraint, i.e. the maximum possible value of $|K| \,/\, |N(K)|$. Now, there may be many sets K with $|K| \,/\, |N(K)| = \widehat{\alpha}$, so let $\widehat{C}$ be a maximal such set; we will show that $\widehat{C}$ is in fact the union of all sets K with $|K| \,/\, |N(K)| = \widehat{\alpha}$. Now, by Lemma 18, every server $s \in N(\widehat{C})$ has $\alpha(s) = \widehat{\alpha}$. We will show that in fact, because $\widehat{C}$ captured *all* sets with tightness $\widehat{\alpha}$, all servers $s \notin N(\widehat{C})$ have $\alpha(s) < \widehat{\alpha}$. Thus, because the flow is balanced, all active edges incident to $\widehat{C}$ or $\widehat{S}$ will be between $\widehat{C}$ and $\widehat{S}$; there will be no active edges coming from the outside. For this reason, any balanced server flow α on $G = (C \cup S)$ can be the thought of as the union of two completely independent server flows: the first (unique) flow assigns $\alpha(s) = \widehat{\alpha} = |\widehat{C}| \,/\, |N(\widehat{C})|$ to all $s \in \widehat{S}$, while the second is a balanced server flow on the remaining graph $G \setminus (\widehat{C} \cup \widehat{S})$. Since this remaining graph is smaller, we can use induction on the size of the graph to claim that this second balanced server flow has unique α-values, which completes the proof of uniqueness. If we follow through the entire inductive chain, we end up with a hierarchy of α-values, which can be viewed as the result of the following peeling procedure: first find the (maximally large) set C_1 that maximizes $\alpha_1 = |C_1| \,/\, |N(C_1|$ and assign every server $s \in N(C_1)$ a value of α_1; then peel off C_1 and $N(C_1)$, find the (maximally large) set C_2 in the remaining graph that maximizes $\alpha_2 = |C_2| \,/\, |N(C_2|$, and assign every server $s \in N(C_2)$ value α_2; peel off C_2 and $N(C_2)$ and continue in this fashion, until every server has some value α_i. These values α_i assigned to each server are precisely the unique $\alpha(s)$ in a balanced server flow.

Remark 20. We were unaware of this when submitting the extended abstract, but a similar hierarchical decomposition was used earlier to compute an approximate matching in the semi-streaming setting: see [12], [21]. Note that unlike those papers, we do not end up relying on this decomposition for our main arguments. We only present it here to give a combinatorial alternative to the convex optimization proof above: regardless of which proof we use, once uniqueness is established, the rest of our analysis is expressed purely in terms of balanced server flows.

Proof of Lemma 14. As already noted, $|N(c)| \geq 1$ for all $c \in C$ is a necessary condition. We will prove that it is sufficient by induction on $i = |S|$. If $|S| = 1$, the flow $\alpha(s) = |C|$ for $s \in S$ is trivially the unique balanced server flow. Suppose now that $i > 1$ and that it holds for all $|S| < i$. Now let $\widehat{\alpha} = \max_{\emptyset \subset K \subseteq C} \frac{|K|}{|N(K)|}$ and let

$$\widehat{C} = \bigcup_{K \in \mathcal{K}} K \qquad \text{where } \mathcal{K} = \left\{ K \subseteq C \;\middle|\; |K| = \widehat{\alpha}\,|N(K)| \right\}$$

Note that for any $K_1, K_2 \in \mathcal{K}$ we have

$$\begin{aligned}
\widehat{\alpha}\,|N(K_1 \cup K_2)| &\geq |K_1 \cup K_2| && \text{(by definition of } \widehat{\alpha}) \\
&= |K_1| + |K_2| - |K_1 \cap K_2| \\
&= \widehat{\alpha}\,|N(K_1)| + \widehat{\alpha}\,|N(K_2)| - |K_1 \cap K_2| && \text{(since } K_1, K_2 \in \mathcal{K}) \\
&\geq \widehat{\alpha}\,|N(K_1)| + \widehat{\alpha}\,|N(K_2)| - \widehat{\alpha}\,|N(K_1 \cap K_2)| && \text{(by definition of } \widehat{\alpha}) \\
&\geq \widehat{\alpha}\,|N(K_1 \cup K_2)| && \text{(since } |N(\cdot)| \text{ is submodular)}
\end{aligned}$$

so $K_1 \cup K_2 \in \mathcal{K}$ and thus $\widehat{C} \in \mathcal{K}$ and $|\widehat{C}| = \widehat{\alpha}\,|N(\widehat{C})|$. If $N(\widehat{C}) = S$ then $\widehat{C} = C$ (otherwise $\frac{|\widehat{C}|}{|N(\widehat{C})|} < \frac{|C|}{|S|} \leq \widehat{\alpha}$) and by Corollary 19 we are done, so suppose $\emptyset \subset N(\widehat{C}) \subset S$. Consider the subgraph G_1 induced by $\widehat{C} \cup N(\widehat{C})$ and the subgraph G_2 induced by $(C \setminus \widehat{C}) \cup (S \setminus N(\widehat{C}))$.

By Corollary 19, G_1 has a unique balanced server flow α_1 with $\alpha_1(s) = \widehat{\alpha}$ for all $s \in N(\widehat{C})$.

By our induction hypothesis, G_2 also has a *unique* balanced server flow α_2.

We proceed to show that the combination of α_1 with α_2 constitutes a unique balanced flow α of the entire graph G, defined as follows:

$$\alpha(s) = \begin{cases} \alpha_1(s) & \text{if } s \in N(\widehat{C}) \\ \alpha_2(s) & \text{otherwise} \end{cases}$$

Note first that α is a balanced server flow for G, because both G_1 and G_2 have a set of x-values that realize them, and by construction these values (together with zeroes for each edge between $C \setminus \widehat{C}$ and $N(\widehat{C})$) realize a balanced server flow for G.

For uniqueness, note that by Lemma 18 any balanced server flow α' for G must have $\alpha'(s) = \widehat{\alpha} = \alpha_1(s)$ for $s \in N(\widehat{C})$. We now show that for any $s \in S \setminus N(\widehat{C})$, any balanced server flow α' must also have $\alpha'(s) = \alpha_2(s)$; then, the uniqueness of α will follow from the uniqueness of α_1 and α_2.

Let $\widehat{S} = \{s \in S \mid \alpha'(s) = \widehat{\alpha}\}$ be the set of maximally necessary servers, and let $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \emptyset\}$ be the set of clients with a maximally necessary server in their active neighborhood. We will show that $\widehat{K} = \widehat{C}$.

"$\subseteq$" By Lemma 17, $|\widehat{K}| = \widehat{\alpha}\,|N(\widehat{K})|$ so by definition of $\widehat{C}$, $\widehat{K} \subseteq \widehat{C}$.

"$\supseteq$" On the other hand, $|\widehat{C}| = \widehat{\alpha}\,|N(\widehat{C})|$ so by Lemma 18 we have $N(\widehat{C}) \subseteq \widehat{S}$ and in particular $A(c) \subseteq \widehat{S}$ for c in $\widehat{C}$ and thus $\widehat{C} \subseteq \widehat{K}$.

Thus, by definition of $\widehat{K}$, $A(c) \cap \widehat{S} = \emptyset$ for all $c \in C \setminus \widehat{C}$. And there are clearly no edges between $\widehat{C}$ and $S \setminus N(\widehat{C})$. But then, for any $(x_e)_{e \in E}$ realizing α', the subset $(x_{cs})_{c \in C \setminus \widehat{C}, s \in S \setminus N(\widehat{C})}$ realizes a balanced server flow in G_2, so since α_2 is the unique balanced server flow in G_2 we have $\alpha'(s) = \alpha_2(s)$ for $s \in S \setminus N(\widehat{C})$. □

3.3 How Server Loads Change as New Clients are Inserted

From now on, let α denote the unique balanced server flow. We want to understand how the balanced server flow changes as new clients are added. For any server s, let $\alpha^{\text{OLD}}(s)$ be the flow in s *before* the insertion of c, and let $\alpha^{\text{NEW}}(s)$ be the flow *after*. Also, let $\Delta\alpha(s) = \alpha^{\text{NEW}}(s) - \alpha^{\text{OLD}}(s)$.

Intuitively, as more clients are added to the graph, the flow on the servers only increases, so no $\alpha(s)$ ever decreases. We now prove this formally.

Lemma 21. *When a new client c is added, $\Delta\alpha(s) \geq 0$ for all $s \in S$.*

Proof. Let $S^* = \{s \in S \mid \alpha^{\text{NEW}}(s) < \alpha^{\text{OLD}}(s)\}$. We want to show that $S^* = \emptyset$. Say for contradiction that $S^* \neq \emptyset$, and let $\alpha^* = \min_{s \in S^*} \alpha^{\text{NEW}}(s)$. We will now partition S into three sets.

$$S^- = \{s \in S \mid \alpha^{\text{OLD}}(s) \leq \alpha^*\}$$
$$S^\Delta = \{s \in S \mid \alpha^{\text{OLD}}(s) > \alpha^* \wedge \alpha^{\text{NEW}}(s) = \alpha^*\}$$
$$S^+ = \{s \in S \mid \alpha^{\text{OLD}}(s) > \alpha^* \wedge \alpha^{\text{NEW}}(s) > \alpha^*\}$$

It is easy to see that these sets form a partition of S, and that $\emptyset \neq S^\Delta \subseteq S^*$.

Now, let C^Δ contain all clients with an active neighbor in S^Δ before the insertion of c. Since each client sends one unit of flow, $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leq \left|C^\Delta\right|$. Now, because we had a balanced flow before the insertion of c there cannot be any edges in G from C^Δ to S^- (any such edge would be from a client $u \in C^\Delta$ to a server $v \in S^-$ with $\alpha^{\text{OLD}}(v) \leq \alpha^* < \alpha^{\text{OLD}}(s)$ for $s \in S^\Delta$ contradicting that u had an active neighbor in S^Δ). Moreover, in the balanced flow after the insertion of c, there are no active edges from C^Δ to S^+ (any such edge would be from a client $u \in C^\Delta$ to a server $v \in S^+$ with $\alpha^{\text{NEW}}(v) > \alpha^* = \alpha^{\text{NEW}}(s)$ for all $s \in S^\Delta$ so is not active). Thus, all active edges incident to C^Δ go to S^Δ, so $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) \geq \left|C^\Delta\right|$. This contradicts the earlier fact that $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leq \left|C^\Delta\right|$, since by definition of S^Δ we have $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) < \sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s)$. □

The next lemma formalizes the following argument: Say that we insert a new client c, and for simplicity say that c is only incident to server s. Now, c will have no choice but to send all of its flow to s, but that does not imply that $\Delta\alpha(s) = 1$, since other clients will balance by retracting their flow from s and sending it elsewhere. But by the assumption that the flow was balanced before the insertion of c, all this new flow can only flow "upward" from s: it cannot end up increasing the flow on some s^- with $\alpha^{\text{OLD}}(s^-) < \alpha^{\text{OLD}}(s)$. Along the same lines of intuition, even if c has several neighbors, inserting c cannot affect the flow of servers whose original flow was less than the lowest original flow among the neighbors of s.

Lemma 22. *When a new client c is added , $\Delta\alpha(s) = 0$ for all s where $\alpha^{\text{OLD}}(s) < \min_{v \in N(c)} \alpha^{\text{OLD}}(v)$.*

Proof. Let us first consider the balanced flow *before* the insertion of c.

Let $S^+ = \left\{s \in S \;\middle|\; \alpha^{\text{OLD}}(s) \geq \min_{v \in N(c)} \alpha^{\text{OLD}}(v)\right\}$ and define $S^- = S \setminus S^+$. We want to show that $\Delta\alpha(s) = 0$ for all servers s in S^-.

Define C^+ to be the set of client vertices whose neighbors are all in S^+; that is, $C^+ = \{c \in C \mid N(c) \subseteq S^+\}$. Note that the following holds before the insertion of c: by definition of C^+ there are no edges in G from C^+ to S^-, and because the flow is balanced, there are no *active* edges from C^- to S^+. Thus, $\sum_{s \in S^-} \alpha^{\text{OLD}}(s) = |C^-|$.

Now consider the insertion of c. By definition of S^- the new client c has no neighbors in S^-, so it is still the case that only clients in C^- have neighbors in S^-. Thus, in the new balanced flow we still have have that $\sum_{s \in S^-} \alpha^{\text{NEW}}(s) \leq |C^-|$. But this means that $\sum_{s \in S^-} \Delta\alpha(s) \leq 0$, so if $\Delta\alpha(s_1) > 0$ for some $s_1 \in S^-$ then $\Delta\alpha(s_2) < 0$ for some $s_2 \in S^-$, which contradicts Lemma 21. □

4 Analyzing replacements in maximum matching

We now consider how server flows relate to the length of augmenting paths.

Lemma 23. *The graph $(C \cup S, E)$ contains a matching of size $|C|$, if and only if $\alpha(s) \leq 1$ for all $s \in S$.*

Proof. Let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$. It follows directly from Lemma 18 that $|K| \leq |N(K)|$ for all $K \subseteq C$ if and only if $\widehat{\alpha} \leq 1$. The corollary then follows from Hall's Theorem (Theorem 7) □

It is possible that in the original graph $G = (C \cup S, E)$, there are many clients that cannot be matched. But recall that by Observation 9, if a client cannot be matched when it is inserted, then it can be effectively ignored for the rest of the algorithm. This motivates the following definition:

Definition 24. We define the set $C_M \subseteq C$ as follows. When a client c is inserted, consider the set of clients C' before c is inserted: then $c \in C_M$ if the maximum matching in $(C' \cup \{c\} \cup S, E)$ is greater than the maximum matching in $(C' \cup S, E)$. Define $G_M = (C_M \cup S, E)$.

Observation 25. *When a client $c \in C_M$ is inserted the SAP algorithm finds an augmenting path from c to a free server; this follows from the fact that SAP always maintains a maximum matching (Lemma 10). By Observation 9, if $c \notin C_M$ then no augmenting path goes through c during the entire sequence of insertions. By the same observation, once a vertex $c \in C_M$ is inserted it remains matched through the entire sequence of insertions.*

Definition 26. Given any $s \in S$, let $\alpha_M(s)$ be the flow into s in some balanced server flow in G_M; by Lemma 14 $\alpha_M(s)$ is uniquely defined.

Observation 27. *By construction G_M contains a matching of size $|C_M|$, so by Lemma 23 $\alpha_M(s) \leq 1$ for all $s \in S$. Finally, note that since $C_M \subseteq C$, we clearly have $\alpha_M(s) \leq \alpha(s)$*

Definition 28. Define an *augmenting tail* from a vertex v to be an alternating path that starts in v and ends in an unmatched server. We call an augmenting tail *active* if all the edges on the alternating path that are not in the matching are active.

Note that augmenting tails as defined above are an obvious extension of the concept of augmenting paths: Every augmenting path for a newly arrived client c consists of an edge (c, s), plus an augmenting tail from some server $s \in N(c)$.

We are now ready to prove our main lemma connecting the balanced server flow to augmenting paths. We show that if some server s has small $\alpha(s)$, then regardless of the particular matching at hand, there is guaranteed to be a *short* active augmenting tail from s. Since every *active* augmenting tail is by definition an augmenting tail, this implies that any newly newly inserted client c that is incident to s has a short augmenting path to an unmatched server.

Lemma 29 (Expansion Lemma)**.** *Let $s \in S$, and suppose $\alpha_M(s) = 1 - \epsilon$ for some $\epsilon > 0$. Then there is an active augmenting tail for s of length at most $\frac{2}{\epsilon} \ln(|C_M|)$.*

Proof. By our definition of active edges, it is not hard to see that any server s' reachable from s by an active augmenting tail has $\alpha_M(s') \leq 1 - \epsilon$.

For $i \geq 1$, let K_i be the set of clients c such that there is an active augmenting tail $s_0, c_0, \ldots, c_{k-1}, s_k$ from s with $c = c_j$ for some $j < i$. Let $k_i = |K_i|$. Note that $k_1 = 1$, $K_1 \subseteq K_2 \subseteq \ldots \subseteq K_i$, and

$$k_i = |K_i| \leq \sum_{s' \in A(K_i)} \alpha_M(s') \leq \sum_{s' \in A(K_i)} (1 - \epsilon) = |A(K_i)|\,(1 - \epsilon)$$

Thus

$$|A(K_i)| \geq \frac{k_i}{1-\epsilon}$$

Suppose there is no active augmenting tail from s of length $\leq 2(i-1)$, then every server in $A(K_i)$ is matched, and the clients they are matched to are exactly K_{i+1}. There is a bijection between $A(K_i)$ and K_{i+1} given by the perfect matching, so we have $k_{i+1} = |A(K_i)|$ and thus $|C_M| \geq k_{i+1} \geq \frac{1}{1-\epsilon} k_i \geq (\frac{1}{1-\epsilon})^i k_1 = (\frac{1}{1-\epsilon})^i$. It follows that $i \leq \frac{\ln |C_M|}{\ln \frac{1}{1-\epsilon}} \leq \frac{1}{\epsilon} \ln |C_M|$, where the last inequality follows from $1 - \epsilon \leq e^{-\epsilon}$. Thus for any $i > \frac{1}{\epsilon} \ln |C_M|$ there exists an active augmenting tail of length at most $2(i-1)$, and the result follows. □

We are now able to prove the key lemma of our paper, which we showed in Section 1.3 implies Theorem 1.

Lemma 6. *Consider the following protocol for constructing a matching: For each client c in arbitrary order, augment along the shortest augmenting path from c (if one exists). Given any h, this protocol augments down a total of at most $4n \ln(n)/h$ augmenting paths of length $> h$.*

Proof. Recall that $n = |C| \geq |C_M|$. The lemma clearly holds for $h \leq 4 \ln(n)$ because there at most n augmenting paths in total. We can thus assume for the rest of the proof that $h > 4 \ln(n)$. Recall by Observation 25 that any augmenting path is contained entirely in G_M. Now, let $C^* \subseteq C_M$ be the set of clients whose shortest augmenting path have length at least $h+1$ when they are added. Our goal is to show that $|C^*| \leq 4n \ln(n)/h$. For each $c \in C^*$ the shortest augmenting tail from each server $s \in N(c)$ has length at least h and so by the Expansion Lemma 29, each server $s \in N(c)$ has $\alpha_M(s) \geq 1 - 2 \ln(n)/h$. Let S^* be the set of all servers that at some point have $\alpha_M(s) \geq 1 - 2 \ln(n)/h$; by Lemma 21, this is exactly the set of servers s such that $\alpha_M(s) \geq 1 - 2 \ln(n)/h$ after all clients have been inserted. By Lemma 22, if $c \in C^*$, the insertion of c only increases the flow on servers in S^* that already had flow at least $1 - 2 \ln(n)/h$. Since by Observation 27 $\alpha_M(s) \leq 1$ for all $s \in S$, the flow of each server in S^* can only increase by at most $2 \ln(n)/h$. But then, since the client c contributes with exactly one unit of flow, the total number of such clients is $|C^*| \leq (2 \log(n)/h) |S^*|$. We complete the proof by showing that $|S^*| < 2n$. This follows from the fact that each client $c \in C_M$ sends one unit of flow, so $n \geq |C_M| \geq (1 - 2 \ln(n)/h) |S^*| > |S^*|/2$, where the last inequality follows from the assumption that $h > 4 \ln(n)$. □

5 Implementation

In the previous section we proved that augmenting along a shortest augmenting path yields a total of $\mathcal{O}(n \log^2 n)$ replacements. But the naive implementation would spend $\mathcal{O}(m)$ time per inserted vertex, leading to total time $\mathcal{O}(mn)$ for actually maintaining the matching. In this section, we show how to find the augmenting paths more quickly, and thus maintain the optimal matching at all times in $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$ total time, differing only by an $\mathcal{O}(\sqrt{\log n})$ factor from the classic offline algorithm of Hopcroft and Karp algorithm for static graphs [19].

Definition 30. Define the height of a vertex v (server or client) to be the length of the shortest augmenting tail (Definition 28) from v. If no augmenting tail exists, we set the height to $2n$.

At a high level, our algorithm is very similar to the standard $\mathcal{O}(m\sqrt{n})$ blocking flow algorithm. We will keep track of heights to find shortest augmenting paths of length at most $\sqrt{n}\sqrt{\log n}$. We will find longer augmenting paths using the trivial $\mathcal{O}(m)$ algorithm, and use Lemma 6 to bound the number of such paths. Our analysis will also require the following lemma:

Lemma 31. *For any server $s \in S$, there is an augmenting tail from s to an unmatched server if and only if $\alpha_M(s) < 1$.*

Proof. If $\alpha_M(s) < 1$, then the existence of *some* tail follows directly from the Expansion Lemma 29. Now let us consider $\alpha_M(s) = 1$. Let $S_1 = \{s \in S \mid \alpha_M(s) = 1\}$. Since 1 is the maximum possible value of $\alpha_M(s)$ (Observation 27), Lemma 17 implies that there is a set of clients $C_1 \in C_M$ such that $N(C_1) = S_1$ and $|C_1| = |S_1|$. Now since every client in C_1 is matched, every server S_1 is matched to some client in C_1. Every augmenting tail from some $s \in S_1$ must start with a matched edge, so it must go through C_1, so it never reaches a server outside of $N(C_1) = S_1$, so it can never reach a free server. □

We now turn to our implementation of the SAP protocol. We will use a dynamic single-source shortest paths algorithm as a building block. We start by defining a directed graph D such that maintaining distances in D will allow us to easily find shortest augmenting paths as new clients are inserted.

Let D be the directed graph obtained from $G = (C \cup S, E)$ by directing all unmatched edges from C to S, and all matched edges from S to C, and finally adding a *sink* t with an edge from all unmatched vertices, as well as edge from all clients in C that have not yet arrived. Any alternating path in G corresponds to a directed path in $D \setminus \{t\}$ and vice-versa. In particular, it is easy to see that if P is a shortest path in D from a matched server s to the sink t, then $P \setminus \{t\}$ is a shortest augmenting tail from s to a free server. Similarly, for any client c that has arrived (so edge (c, t) is deleted) but is not yet matched, if P is the shortest path from c to t in D, then $P \setminus \{t\}$ is a shortest augmenting path for c in G. Furthermore, augmenting down this path in G corresponds (in D) to changing the direction of all edges on $P \setminus \{t\}$ and deleting the edge on P incident to t.

We can thus keep track of shortest augmenting paths by using a simple dynamic shortest path algorithm to maintain shortest paths to t in the changing graph D. We will use a modification of Even and Shiloach (See [28]) to maintain a shortest path tree in D to t from all vertices of height at most $h = \sqrt{n}\sqrt{\log n}$. The original version by Even and Shiloach worked only for undirected graphs, and only in the decremental setting where the graph only undergoes edge deletions, never edge insertions. This was later extended by King [23] to work for directed graphs. The only-deletions setting is too constrained for our purposes because we will need to insert edges into D; augmenting down a path P corresponds to deleting the edges on P and inserting the reverse edges. Fortunately, it is well known that the Even and Shiloach tree can be extended to the setting where there are both deletions and insertions, as long as the latter are guaranteed not to decrease distances; we will show that this in fact applies to our setting.

Lemma 32 (Folklore. See e.g. [3, 4])**.** *Let $G = (V, E)$ be a dynamic directed or undirected graph with positive integer weights, let t be a fixed sink, and say that for every vertex v we are guaranteed that the distance* $\text{dist}(v, t)$ *never decreases due to an edge insertion. Then we can maintain a tree of shortest paths to t up to distance d in total time $\mathcal{O}(m \cdot d + \Delta)$, where m is the total number of edges (u, v) such that (u, v) is in the graph at any point during the update sequence, and Δ is the total number of edge changes.*

Theorem 2. *There is an implementation of the SAP protocol that runs in total time* $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$.

Proof. We will explicitly maintain the graph D, and use the extended Even-Shiloach tree structure from Lemma 32 to maintain a tree T of shortest paths to t up to distance $h = \sqrt{n}\sqrt{\log n}$. Every vertex will either be in this tree (and hence have height less than h), or be marked as a *high* vertex. When a new client c arrives, we update D (and T) by first adding edges to $N(c)$ from c, and then deleting the dummy edge from c to t. Note that because the deletion of edge (c, t) comes last, the inserted edges do not change any distances to t. We then use D and T to find a shortest augmenting path. We consider two cases.

The first case is when c is not high. Then T contains a shortest path P from c to t.

The second case is when c is high. In this case we can just brute-force search for a shortest path P from c to t in time $\mathcal{O}(m)$. *If we do not find a path from c to t, then we remove all servers and clients encountered during the search, and continue the algorithm in the graph with these vertices removed.*

In either case, if a shortest path P from c to t is found, we augment down P and then make the corresponding changes to D: we first reverse the edges on $P \setminus \{t\}$ in order starting with the edge closest to c, and then we delete the edge (s, t) on P incident to t (because the server s is now matched). Each edge reversal is done by first inserting the reversed edge, and then deleting the original. Note that since P is a shortest path, none of these edge insertions change the distances to t.

Correctness: We want to show that our implementation chooses a shortest augmenting path at every step. This is clearly true if we always find an augmenting path, but otherwise becomes a bit more subtle as we delete vertices from the graph after a failed brute-force search. We must thus show that any vertex deleted in this way cannot have participated in any future augmenting path.

To see this, note that when our implementation deletes a server $s \in S$, there must have been no augmenting path through s at the time that s was deleted. By Lemma 31, this implies that $\alpha_M(s) = 1$. But then by Lemma 21 we have $\alpha_M(s) = 1$ for all future client insertions as well. (Recall that by Observation 27 we never have $\alpha_M(s) > 1$.) Thus by Lemma 31 there is never an augmenting path through s after this point, so s can safely be deleted from the graph. Similarly, if a client c is deleted from the graph, then all of its neighboring servers had no augmenting tails at that time, so they all have $\alpha_M(s) = 1$, so there will never be an augmenting path through c.

Running time: There are three factors to consider.

1. the time to follow the augmenting paths and maintain D.
2. the time to maintain T.
3. the time to brute-force search for augmenting paths.

Item 1 takes $\mathcal{O}(m + n\log^2 n)$ time because we need $\mathcal{O}(1)$ time to add each of the m edges and to follow and reverse each edge in the augmenting paths, and by Theorem 1 the total length of augmenting paths is $\mathcal{O}(n\log^2 n)$.

For Item 2, it is easy to see that the total number of edges ever to appear in D is $m = O(|E|)$; D consists only of dummy edges to the sink t, and edges in the original graph oriented in one of

two directions. By Item 1, the number of changes to D is $\mathcal{O}(m + n\log^2 n)$. Thus by Lemma 32 the total time to maintain T is $\mathcal{O}(mh + n\log^2 n)$.

For Item 3 we consider two cases. The first is brute-force searches which result in finding an augmenting path. These take a total of $\mathcal{O}(mn\log(n)/h)$ time because by Lemma 6 during the course of the entire algorithm there are at most $\mathcal{O}(n\log(n)/h)$ augmenting paths of length $\geq h$, and each such path requires $\mathcal{O}(m)$ time to find. The second case to consider is brute-force searches that do not result in an augmenting path. These take total time $\mathcal{O}(m)$ because once a vertex participates in such a search, it is deleted from the graph with all its incident edges.

Summing up, the total time used is $\mathcal{O}(mh + n\log^2 n + mn\log(n)/h + m)$, which for our choice of $h = \sqrt{n}\sqrt{\log n}$ is $O(m\sqrt{n}\sqrt{\log n})$. □

6 Extensions

In many applications of online bipartite assignments, it is natural to consider the extension in which each server can serve multiple clients. Recall from the introduction that we examine two variants: capacitated assignment, where each server comes with a fixed capacity which we are not allowed to exceed, and minimizing maximum server load, in which there is no upper limit to the server capacity, but we wish to minimize the maximum number of clients served by any server. We show that there is a substantial difference between the number of reassignments: Capacitated assignment is equivalent to uncapacitated online matching with replacements, but for minimizing maximum load, we show a significantly higher lower bound.

6.1 Capacitated assignment

We first consider the version of the problem where each server can be matched to multiple clients. Each server comes with a positive integer capacity $u(s)$, which denotes how many clients can be matched to that server. The greedy algorithm is the same as before: when a new client is inserted, find the shortest augmenting path to a server s that currently has less than $u(s)$ clients assigned.

Theorem 3. *SAP uses at most $\mathcal{O}(n\log^2 n)$ reassignments for the capacitated assignment problem, where n is the number of clients.*

Proof. There is a trivial reduction from any instance of capacitated assignment to one of uncapacitated matching where each server can only be matched to one client: simple create $u(s)$ copies of each server s. This reduction was previously used in [2]. When a client c is inserted, if there is an edge (c, s) in the original graph, then add edges from c to every copy of s. It is easy to see that the number of flips made by the greedy algorithm in the capacitated graph is exactly equal to the number made in the uncapacitated graph, which by Theorem 1 is $\mathcal{O}(n\log^2 n)$. (Note that although the constructed uncapacitated graph has more servers than the original capacitated graph, the number of clients n is exactly the same in both graphs.) □

6.2 Minimizing maximum server load

In this section, we analyze the online assignment problem. Here, servers may have an unlimited load, but we wish to minimize maximum server load.

Definition 33. Given a bipartite graph $G = (C \cup S, E)$, an assignment $\mathcal{A} : C \to S$ assigns each client c to a server $\mathcal{A}(c) \in S$. Given some assignment $\mathcal{A}$, for any $s \in S$ let the *load* of s, denoted $\ell_{\mathcal{A}}(s)$, be the number of clients assigned to s; when the assignment $\mathcal{A}$ is clear from context we just write $\ell(s)$. Let $\ell(\mathcal{A}) = \max_{s \in S} \ell_{\mathcal{A}}(s)$. Let $\textsc{opt}(G)$ be the minimum load among all possible assignments from C to S.

In the online assignment problem, clients are again inserted one by one with all their incident edges, and the goal is to maintain an assignment with minimum possible load. More formally, define $G_t = (C_t \cup S, E_t)$ to be the graph after exactly t clients have arrived, and let $\mathcal{A}_t$ be the assignment at time t. Then we must have that for all t, $\ell(\mathcal{A}_t) = \textsc{opt}(G_t)$. The goal is to make as few changes to the assignment as possible.

[15] and [2] showed how to solve this problem with approximation: namely, with only $\mathcal{O}(1)$ amortized changes per client insertion they can maintain an assignment $\mathcal{A}$ such that for all t, $\ell(\mathcal{A}_t) \leq 8\textsc{opt}(G_t)$. Maintaining an approximate assignment is thus not much harder than maintaining an approximate maximum matching, so one might have hoped that the same analogy holds for the exact case, and that it is possible to maintain an optimal assignment with amortized $\mathcal{O}(\log^2 n)$ changes per client insertion. We now present a lower bound disproving the existence of such an upper bound. The lower bound is not specific to the greedy algorithm, and applies to any algorithm for maintaining an assignment $\mathcal{A}$ of minimal load. In fact, the lower bound applies even if the algorithm knows the entire graph G in advance; by contrast, if the goal is only to maintain a maximum matching, then knowing G in advance trivially leads to an online matching algorithm that never has to rematch any vertex.

Theorem 4. *For any positive integers n and $L \leq \sqrt{n/2}$ divisible by 4 there exists a graph $G = (C \cup S, E)$ with $|C| = n$ and $\textsc{opt}(G) = L$, along with an ordering in which the clients in C are inserted, such that any algorithm for the exact online assignment problem requires a total of $\Omega(nL)$ changes. This lower bound holds even if the algorithm knows the entire graph G in advance, as well as the order in which the clients are inserted.*

The main ingredient of the proof is the following lemma:

Lemma 34. *For any positive integer L divisible by 4, there exists a graph $G = (C \cup S, E)$ along with an ordering in which clients in C are inserted, such that $|C| = L^2$, $|S| = L$, $\textsc{opt}(G) = L$, and any algorithm for maintaining an optimal assignment $\mathcal{A}$ requires $\Omega(L^3)$ changes to $\mathcal{A}$.*

Proof. Let $S = \{s_1, s_2, ..., s_L\}$. We partition the clients in C into L blocks $C_1, C_2, ..., C_L$, where all the clients in a block have the same neighborhood. In particular, clients in C_L only have a single edge to server s_L, and clients in C_i for $i < L$ have an edge to s_i and s_{i+1}.

The online sequence of client insertions begins by adding $L/2$ clients to each block C_i. The online sequence then proceeds to alternate between *down-heavy* epochs and *up-heavy* epochs, where a down-heavy epoch inserts 2 clients into blocks $C_1, C_2, ..., C_{L/2}$ (in any order), while an up-heavy epoch inserts 2 clients into blocks $C_{L/2+1}, ..., C_L$. The sequence then terminates after $L/2$ such epochs: $L/4$ up-heavy ones and $L/4$ down-heavy ones in alternation. Note that a down-heavy epoch followed by an up-heavy one simply adds two clients to each block. Thus the final graph has $|C_i| = L$ for each i, so the graph $G = (C \cup S, E)$ satisfies the desired conditions that $|C| = L^2$ and $\textsc{opt}(G) = L$.

We complete the proof by showing that all the client insertions during a single down-heavy epoch cause the algorithm to make at least $\Omega(L^2)$ changes to the assignment; the same analysis

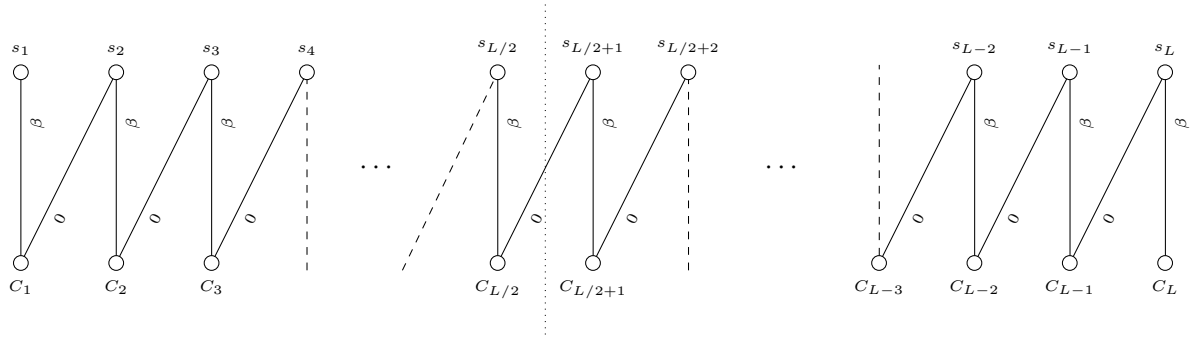


Figure 1: Number of assignments of each type after first $L/2$ clients added to each block, and after each up-heavy phase. Each C_i has β clients. Each server has β clients assigned.

applies to the up-heavy epochs as well. Consider the kth down-heavy epoch of client insertions. Let $\beta = L/2 + 2(k-1)$ and consider the graph $G^{\text{OLD}} = (C^{\text{OLD}} \cup S, E^{\text{OLD}})$ before the down-heavy epoch: it is easy to see that every block C_i has exactly β clients, that $\text{OPT}(G^{\text{OLD}}) = \beta$, and that there is exactly one assignment $\mathcal{A}^{\text{OLD}}$ that adheres to this maximum load: $\mathcal{A}^{\text{OLD}}$ assigns all clients in block C_i to server s_i (see Figure 1).

Now, consider the graph $G^{\text{NEW}} = (C^{\text{NEW}} \cup S, E^{\text{NEW}})$ after the down-heavy epoch. Blocks $C_1, C_2, ..., C_{L/2}$ now have $\beta + 2$ clients, while blocks $C_{L/2+1}, ..., C_L$ still only have β. We now show that $\text{OPT}(G^{\text{NEW}}) = \beta + 1$. In particular, recall that $\beta \geq L/2$ and consider the following assignment $\mathcal{A}^{\text{NEW}}$: for $i \leq L/2$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + 2 - i \geq 2$ clients from C_i to s_i and i clients in C_i to s_{i+1}; for $L/2 < i \leq L$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + i - L \geq 0$ clients in C_i to s_i, and $L - i$ clients from C_i to s_{i+1}. (In particular, all β clients in C_L are assigned to s_L, which is necessary as there is no server s_{L+1}). It is easy to check that for every $s \in S$, $\ell_{\mathcal{A}^{\text{NEW}}}(s) = \beta + 1$ (see Figure 2).

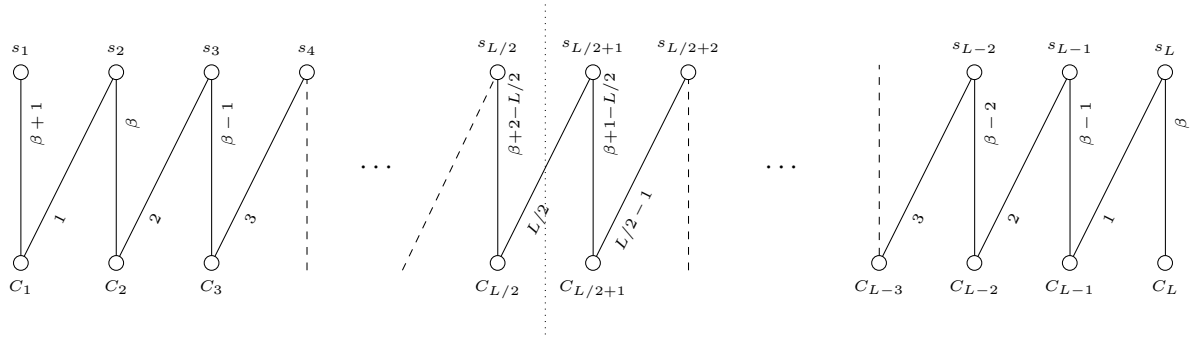


Figure 2: Number of assignments of each type after each down-heavy phase. Each C_i has $\beta + 2$ clients for $1 \leq i \leq L/2$ and β clients for $L/2 + 1 \leq i \leq L$. Each server has $\beta + 1$ clients assigned.

We now argue that $\mathcal{A}^{\text{NEW}}$ is in fact the only assignment in G^{NEW} with $\ell(\mathcal{A}^{\text{NEW}}) = \beta + 1$. Consider any assignment $\mathcal{A}$ for C^{NEW} with $\ell(\mathcal{A}) = \beta + 1$. Observe that since the total number of clients in C^{NEW} is exactly $(\beta + 1)L$, we must have that every server $s \in S$ has $\ell(s) = \beta + 1$ in $\mathcal{A}$. We now argue by induction that for $i \leq \beta/2$, $\mathcal{A}$ assigns assigns $\beta + 2 - i$ clients from C_i to s_i and i clients in C_i to s_{i+1} (exactly as $\mathcal{A}^{\text{NEW}}$ does). The claim holds for $i = 1$ because the only way s_1 can end up with load $\beta + 1$ is if $\beta + 1$ clients from C_1 are assigned to it. Now say the claim is true for some $i < \beta/2$. By the induction hypothesis, s_{i+1} has i clients from C_i assigned to it. Since s_{i+1}

must have total load $\beta + 1$, and all clients assigned to it come from C_i or C_{i+1}, s_{i+1} must have $\beta + 1 - i = \beta + 2 - (i + 1)$ clients assigned to it from C_{i+1}.

We now prove by induction that for all $L/2 < i \leq L$, $\mathcal{A}$ assigns $\beta + i - L$ clients in C_i to s_i, and $L - i$ clients from C_i to s_{i+1}, which proves that $\mathcal{A} = \mathcal{A}^{\text{NEW}}$. The claim holds for $i = L/2 + 1$ because we have already shown that in the above paragraph that $L/2$ clients assigned to $s_i = s_{L/2+1}$ come from $C_{L/2}$, so since $\ell(s_i) = \beta + 1$, it must have $\beta + 1 - L/2 = \beta + i - L$ clients from C_i assigned to it. Now, say that the claim is true for some $i > L/2$. Then by the induction step s_{i+1} has $L - i$ clients assigned to it from C_i, so since $\ell(s_{i+1}) = \beta + 1$, it has $\beta + (i + 1) - L$ clients assigned to it from C_{i+1}, as desired. The remaining $L - (i + 1)$ clients in C_{i+1} must then be assigned to s_{i+2}.

We have thus shown that the online assignment algorithm is forced to have assignment $\mathcal{A}^{\text{OLD}}$ before the down-heavy epoch, and assignment $\mathcal{A}^{\text{NEW}}$ afterwards. We now consider how many changes the algorithm must make to go from one to another. Consider block C_i for some $L/2 < i \leq L$. Note that because the epoch of client insertions was down-heavy, $|C_i| = \beta$ before and after the epoch. Now, in $\mathcal{A}^{\text{OLD}}$ all of the clients in C_i are matched to s_i. But in $\mathcal{A}^{\text{NEW}}$, $L - i$ of them are matched to s_{i+i}. Thus, the total number of reassignments to get from $\mathcal{A}^{\text{OLD}}$ to $\mathcal{A}^{\text{NEW}}$ is at least $\sum_{L/2<i\leq L}(L - i) = \Omega(L^2)$. Since there are $L/4$ down-heavy epochs, the total number of reassignments over the entire sequence of client insertions is $\Omega(L^3)$. □

Proof of Theorem 4. Recall the assumption of the Theorem that $n/2 \geq L^2$. Simply let the graph G consist of $\lfloor n/L^2 \rfloor$ separate instances of the graph in Lemma 34, together with sufficient copies of $K_{1,1}$ to make the total number of clients n. The algorithm will have to make $\Omega(L^3)$ changes in each such instance, leading to $\Omega(L^3 \lfloor n/L^2 \rfloor) = \Omega(nL)$ changes in total. □

We now show a nearly matching upper bound which is off by a $\log^2 n$ factor. As with the case of matching, this upper bound is achieved by the most natural SAP algorithm, which we now define in this setting. Since $\text{OPT}(G)$ may change as clients are inserted into C, whenever a new client is inserted, the greedy algorithm must first compute $\text{OPT}(G)$ for the next client set. Note that the algorithm does not do any reassignments at this stage, it simply figures out what the max load should be. $\text{OPT}(G)$ can easily be computed in polynomial time: for example we could just compute the maximum matching when every server has capacity b for every $b = 1, 2, ..., |C|$, and then $\text{OPT}(G)$ is the minimum b for which every client in C is matched; for a more efficient approach see [2]. Now, when a new client c is inserted, the algorithm first checks if $\text{OPT}(G)$ increases. If yes, the maximum allowable load on each server increases by 1 so c can just be matched to an arbitrary neighbor. Otherwise, SAP finds the shortest alternating path from c to a server s with $\ell(s) < \text{OPT}(G)$: an augmenting path is defined exactly the same way as in Definition 8, though there may now be multiple matching edges incident to every server. The proof of the upper bound will rely on the following very simple observation:

Observation 35. *For the uncapacitated problem of online maximum matching with replacements, let us say that instead of starting with $C = \emptyset$, the algorithm starts with some initial set of clients $C_0 \subset C$ already inserted, and an initial matching between C_0 and S. Then the total number of replacement made during all future client insertions is still upper bounded by the same $\mathcal{O}(n \log^2 n)$ as in Theorem 1, where n is the number of clients in the final graph (so n is $|C_0|$ plus the number of clients inserted).*

Proof. Intuitively, we could simply let our protocol start by unmatching all the clients in C_0, and then rematching them according the SAP protocol, which would lead to $\mathcal{O}(n \log^2 n)$ replacements.

In fact this initial unmatching is not actually necessary. Recall that the proof of Theorem 1 follows directly from the key Lemma 6, which in term follows from the expansion argument in Lemma 29. The expansion argument only refers to server necessities, not to the particular matching maintained by the algorithm, so it will hold no matter what initial matching we start with. □

Theorem 5. *Let C be the set of all clients inserted, let $n = |C|$, and let $L = \text{OPT}(G)$ be the minimum possible maximum load in the final graph $G = (C \cup S, E)$. SAP at all times maintains an optimal assignment while making a total of $\mathcal{O}(n \min\{L \log^2 n, \sqrt{n} \log n\})$ reassignments.*

Proof. Let us define epoch i to contain all clients c such that after the insertion of c we have $\text{OPT}(G) = i$. We now define n_i as the total number of clients added by the end of epoch i (so n_i counts clients from previous epochs as well). Extend the reduction in the proof of Theorem 3 from [2] as follows: between any two epochs, add a new copy of each server, along with all of its edges. For the following epoch, say, the ith epoch, Observation 35 tells us that regardless of what matching we had at the beginning of the epoch, the total number of reassignments performed by SAP during the epoch will not exceed $\mathcal{O}(n_i \log^2 n_i) \subseteq \mathcal{O}(n \log^2 n)$. We thus make at most $\mathcal{O}(nL \log^2 n)$ reassignments in total, which completes the proof if $L < \sqrt{n}/\log n$. If $L \geq \sqrt{n}/\log n$, we make $\mathcal{O}(n\sqrt{n} \log n)$ reassignments during the first $\sqrt{n}/\log n$ epochs. In all future epochs, note that a server at its maximum allowable load has at least $\sqrt{n}/\log n$ clients assigned to it, so there are at most $\sqrt{n} \log n$ such servers, and whenever a client is inserted the shortest augmenting path to a server below maximum load will have length $\mathcal{O}(\sqrt{n} \log n)$. This completes the proof because there are only n augmenting paths in total. □

6.3 Approximate semi-matching

Though our result on minimizing maximum load *exactly* is tight, we conclude this section on extensions with a short and cute improvement for *approximate* load balancing, which follows from the Expansion Lemma (Lemma 29).

We study a setting similar to that of [2], in which one wishes to minimize not only the maximum load, but the p-norm $|X|_p = (\sum_{s \in S} l(s)^p)^{\frac{1}{p}}$, where $l(s)$ is the load of the server s in the assignment X, for every $p \geq 1$.

First, observe that a lower bound on the p-norm comes from our necessity values α from Section 3. That is, $(\sum_{s \in S} \alpha(s)^p)^{\frac{1}{p}} \leq |X|_p$, for any assignment X. For $p = 1$, we even have equality, as we simply count the number of clients. For $p > 1$, the proof is almost identical to that of uniqueness in Section 3.2.1.

In Section 6.2, we saw that even for the ∞-norm, we cannot obtain $\lceil \alpha(s) \rceil$ with logarithmic recourse. This motivates the use of approximation, and motivates the following definition:

Definition 36 ($(1+\varepsilon)$-approximate semi-matching)**.** For each server s in the current graph, let $L(s) = \lceil (1+\varepsilon)\alpha(s) \rceil$. We say that a semi-matching is $(1+\varepsilon)$-approximate, if each server s is assigned at most $L(s)$ clients.

Assigning $(1+\varepsilon)\alpha(s)$ clients to server s would indeed give a $(1+\varepsilon)$-approximation for every p-norm. Unfortunately, $(1+\varepsilon)\alpha(s)$ may not be an integer, which is why we apply the natural ceiling operation. Under the further assumption that $\alpha(s) \geq \frac{1}{\varepsilon}$ for all s, we have

$$\frac{\lceil (1+\varepsilon)\alpha(s) \rceil}{\alpha(s)} < \frac{(1+\varepsilon)\alpha(s) + 1}{\alpha(s)} = 1 + \varepsilon + \frac{1}{\alpha(s)} \leq 1 + 2\varepsilon$$

And thus, under the assumption that all necessities are $\geq \frac{1}{\varepsilon}$, then for any $(1+\varepsilon)$-approximate assignment X, where we let $l(s)$ denote the number of clients assigned to server $s \in S$, we have:

$$\left(\sum_{s\in S} l(s)^p\right)^{\frac{1}{p}} \leq \left(\sum_{s\in S} \lceil(1+\varepsilon)\alpha(s)\rceil^p\right)^{\frac{1}{p}} < \left((1+2\varepsilon)^p \sum_{s\in S} \alpha(s)^p\right)^{\frac{1}{p}} = (1+2\varepsilon)\left(\sum_{s\in S} \alpha(s)^p\right)^{\frac{1}{p}}$$

But as already noted, the p-norm of the α-vector is a lower bound on any assignment, including the optimal assignment X_{OPT}, so $|X|_p \leq (1+2\varepsilon)\,|X_{\text{OPT}}|$.

In the following, let n denote the number of clients that have arrived thus far.

Theorem 37. *$(1+\varepsilon)$-approximate semi-matching has worst-case $O(\frac{1}{\varepsilon}\log n)$ reassignments with SAP.*

The proof of this theorem relies again on the Expansion Lemma. In this case, however, we do not use the α-values as part of an amortization argument, but only to bound the lengths of the shortest augmenting paths.

Proof. Given our graph G, let G' denote a similar graph with $L(s)$ copies of each server. Then any maximum matching in G' corresponds to an $(1+\varepsilon)$-approximate semi-matching in G. Now, note that each client-set K in G' has a neighborhood of at least $(1+\varepsilon)$ times its own size:

$$|N_{G'}(K)| \quad = \quad \sum_{s\in N_G(K)} L(s) \quad \geq \quad (1+\varepsilon) \sum_{s\in N_G(K)} \alpha_G(s) \quad \geq \quad (1+\varepsilon)\,|K|$$

Where the last inequality follows from the fact that the neighborhood of K receives at least all the flow from K, and thus, at least K flow. Thus, by Lemma 18 we can upper bound the highest alpha-value $\widehat{\alpha}$ in G' by

$$\widehat{\alpha} = \max_{\emptyset\subset K\subseteq C} \frac{|K|}{|N_{G'}(K)|} \leq \frac{1}{1+\varepsilon} = 1 - \frac{\varepsilon}{1+\varepsilon}$$

By setting $\varepsilon' = \frac{\varepsilon}{1+\varepsilon}$, all servers of G' has necessity $\leq 1-\varepsilon'$. The Expansion Lemma (Lemma 29) then gives that any active augmenting tail has length at most $\frac{2}{\varepsilon'}\ln(n) = 2\frac{1+\varepsilon}{\varepsilon}\ln n$, which is $O(\frac{1}{\varepsilon}\log n)$. □

7 Conclusion

We showed that in the online matching problem with replacements, where vertices on one side of the bipartition are fixed (the servers), while those the other side arrive one at a time with all their incoming edges (the n clients), the shortest augmenting path protocol maintains a maximum matching while only making amortized $\mathcal{O}(\log^2 n)$ changes to the matching per client insertion. This almost matches the $\Omega(\log n)$ lower bound of Grove et al. [13]. Ours is the first paper to achieve polylogarithmic changes per client; the previous best of Bosek et al. required $\mathcal{O}(\sqrt{n})$ changes, and used a non-SAP strategy [5]. The SAP protocol is especially interesting to analyze because it is the most natural greedy approach to maintaining the matching. However, despite the conjecture of Chaudhuri et al. [8] that the SAP protocol only requires $\mathcal{O}(\log n)$ amortized changes per client, our analysis is the first to go beyond the trivial $\mathcal{O}(n)$ bound for general bipartite graphs; previous results were only able to analyze SAP in restricted settings. Using our new analysis technique,

we were also able to show an implementation of the SAP protocol that requires total update time $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$, which almost matches the classic offline $\mathcal{O}(m\sqrt{n})$ running time of Hopcroft and Karp [19].

The main open problem that remains is to close the gap between our $\mathcal{O}(\log^2 n)$ upper bound and the $\Omega(\log n)$ lower bound. This would be interesting for any replacement strategy, but it would also be interesting to know what the right bound is for the SAP protocol in particular. Another open question is to remove the $\sqrt{\log n}$ factor in our implementation of the SAP protocol. Note that both of these open questions would be resolved if we managed to improve the bound in Lemma 6 from $\mathcal{O}(n \ln(n)/h)$ to $\mathcal{O}(n/h)$. (In the implementation of Section 5 we would then set $h = \sqrt{n}$ instead of $h = \sqrt{n}\sqrt{\log n}$.)

8 Acknowledgements

The first author would like to thank Cliff Stein for introducing him to the problem. The authors would like to thank Seffi Naor for pointing out to us that uniqueness of server loads can be proved via convex optimization (Section 3.2.1), and to thank Martin Skutella and Guilamme Sagnol for very helpful pointers regarding the details of this proof.

References

[1] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. *Algorithmica*, 23(4):278–301, Apr 1999.

[2] A. Bernstein, T. Kopelowitz, S. Pettie, E. Porat, and C. Stein. Simultaneously load balancing for every p-norm, with reassignments. In *Proceedings of the 8th Conference on Innovations in Theoretical Computer Science (ITCS)*, 2017.

[3] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: beyond the O(mn) bound. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 389–397, 2016.

[4] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1355–1365, 2011.

[5] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych. Online bipartite matching in offline time. In *55th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 384–393. IEEE Computer Society, 2014.

[6] B. Bosek, D. Leniowski, P. Sankowski, and A. Zych. Shortest augmenting paths for online matchings on trees. In *Approximation and Online Algorithms: 13th International Workshop, WAOA 2015, Patras, Greece, September 17-18, 2015. Revised Selected Papers*, pages 59–71, Cham, 2015. Springer International Publishing.

[7] B. Bosek, D. Leniowski, A. Zych, and P. Sankowski. The shortest augmenting paths for online matchings on trees. *CoRR*, abs/1704.02093, 2017.

[8] K. Chaudhuri, C. Daskalakis, R. D. Kleinberg, and H. Lin. Online bipartite perfect matching with augmentations. In *The 31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1044–1052. IEEE, 2009.

[9] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation. *Soviet Math Doklady*, 11:1277–1280, 1970.

[10] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972.

[11] Leah Epstein and Asaf Levin. *Robust Algorithms for Preemptive Scheduling*, pages 567–578. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[12] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 468–485, 2012.

[13] E. F. Grove, M.-Y. Kao, P. Krishnan, and J. S. Vitter. Online perfect matching and mobile computing. In S. G. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, pages 194–205. Springer, Berlin„ 1995.

[14] Albert Gu, Anupam Gupta, and Amit Kumar. The power of deferral: Maintaining a constant-competitive steiner tree online. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 525–534, New York, NY, USA, 2013. ACM.

[15] A. Gupta, A. Kumar, and C. Stein. Maintaining assignments online: Matching, scheduling, and flows. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 468–479, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[16] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 537–550, New York, NY, USA, 2017. ACM.

[17] Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 455–467, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[18] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

[19] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[20] Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM Journal on Discrete Mathematics*, 4(3):369–384, 1991.

[21] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1679–1697, 2013.

[22] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

[23] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 81–91. IEEE Computer Society, 1999.

[24] Jakub Łącki, Jakub Ocwieja, Marcin Pilipczuk, Piotr Sankowski, and Anna Zych. The power of dynamic distance oracles: Efficient dynamic algorithms for the steiner tree. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 11–20, 2015.

[25] Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online mst and tsp. *SIAM Journal on Computing*, 45(3):859–880, 2016.

[26] S. Phillips and J. Westbrook. On-line load balancing and network flow. *Algorithmica*, 21(3):245–261, Jul 1998.

[27] Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.

[28] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981.

[29] Martin Skutella and José Verschae. A robust PTAS for machine covering and packing. In Mark de Berg and Ulrich Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2010.

[30] Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Selfish load balancing and atomic congestion games. *Algorithmica*, 47(1):79–96, 2007.

[31] Jeffery Westbrook. Load balancing for response time. *Journal of Algorithms*, 35(1):1 – 16, 2000. Announced at ESA'95.

Dynamic Bridge-Finding in $\widetilde{O}(\log^2 n)$ Amortized Time

Jacob Holm*, Eva Rotenberg, and Mikkel Thorup*

University of Copenhagen (DIKU),
jaho@di.ku.dk, eva@rotenberg.dk, mthorup@di.ku.dk

August 28, 2018

Abstract

We present a deterministic fully-dynamic data structure for maintaining information about the bridges in a graph. We support updates in $\widetilde{O}((\log n)^2)$ amortized time, and can find a bridge in the component of any given vertex, or a bridge separating any two given vertices, in $\mathcal{O}(\log n/\log\log n)$ worst case time. Our bounds match the current best for bounds for deterministic fully-dynamic connectivity up to $\log\log n$ factors.

The previous best dynamic bridge finding was an $\widetilde{O}((\log n)^3)$ amortized time algorithm by Thorup [STOC2000], which was a bittrick-based improvement on the $\mathcal{O}((\log n)^4)$ amortized time algorithm by Holm et al.[STOC98, JACM2001].

Our approach is based on a different and purely combinatorial improvement of the algorithm of Holm et al., which by itself gives a new combinatorial $\widetilde{O}((\log n)^3)$ amortized time algorithm. Combining it with Thorup's bittrick, we get down to the claimed $\widetilde{O}((\log n)^2)$ amortized time.

Essentially the same new trick can be applied to the biconnectivity data structure from [STOC98, JACM2001], improving the amortized update time to $\widetilde{O}((\log n)^3)$.

We also offer improvements in space. We describe a general trick which applies to both of our new algorithms, and to the old ones, to get down to linear space, where the previous best use $\mathcal{O}(m + n\log n\log\log n)$.

Our result yields an improved running time for deciding whether a unique perfect matching exists in a static graph.

*This research is supported by Mikkel Thorup's Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

1 Introduction

In graphs and networks, connectivity between vertices is a fundamental property. In real life, we often encounter networks that change over time, subject to insertion and deletion of edges. We call such a graph *fully dynamic*. Dynamic graphs call for dynamic data structures that maintain just enough information about the graph in its current state to be able to promptly answer queries.

Vertices of a graph are said to be *connected* if there exists a path between them, and *k-edge connected* if no sequence of $k-1$ edge deletions can disconnect them. A *bridge* is an edge whose deletion would disconnect the graph. In other words, a pair of connected vertices are 2-edge connected if they are not separated by a bridge. By Menger's Theorem [20], this is equivalent to saying that a pair of connected vertices are 2-edge connected if there exist two edge-disjoint paths between them. By edge-disjoint it is meant that no edge appears in both paths.

For dynamic graphs, the first and most fundamental property to be studied was that of dynamic connectivity. In general, we can assume the graph has a fixed set of n vertices, and we let m denote the current number of edges in the graph. The first data structure with sublinear $\mathcal{O}(\sqrt{n})$ update time is due to Frederickson [6] and Eppstein et al. [5]. Later, Frederickson [7] and Eppstein et al. [5] gave a data structure with $\mathcal{O}(\sqrt{n})$ update time for 2-edge connectivity. Henzinger and King achieved polylogarithmic expected amortized time [11], that is, an expected amortized update time of $\mathcal{O}((\log n)^3)$, and $\mathcal{O}(\log n/\log\log n)$ query time for connectivity. And in [12], $\mathcal{O}((\log n)^5)$ expected amortized update time and $\mathcal{O}(\log n)$ worst case query time for 2-edge connectivity. The first polylogarithmic deterministic result was by Holm et al. announced in [13], see [14] for a journal version; an amortized deterministic update time of $\mathcal{O}((\log n)^2)$ for connectivity, and $\mathcal{O}((\log n)^4)$ for 2-edge connectivity. The update time for deterministic dynamic connectivity has later been improved to $\mathcal{O}((\log n)^2/\log\log n)$ by Wulff-Nilsen [24]. Sacrificing determinism, an $\mathcal{O}(\log n(\log\log n)^3)$ structure for connectivity was presented by Thorup [23], and later improved to $\mathcal{O}(\log n(\log\log n)^2)$ by Huang et al. [15]. In the same paper, Thorup obtains an update time of $\mathcal{O}((\log n)^3\log\log n)$ for deterministic 2-edge connectivity. Interestingly, Kapron et al. [16] gave a Monte Carlo-style randomized data structure with polylogarithmic worst case update time for dynamic connectivity, namely, $\mathcal{O}((\log n)^4)$ per edge insertion, $\mathcal{O}((\log n)^5)$ per edge deletion, and $\mathcal{O}(\log n/\log\log n)$ per query. This was later improved by Gibbs et al. [10] to $\mathcal{O}((\log n)^4)$ worst case update time and sublinear $O(n\log^2 n)$ space. We know of no similar worst-case result for bridge finding. The same paper [10] also gives the first sublinear-space $O(n\log^2 n)$ space data structure for (amortized) 2-edge connectivity, by using the sublinear-space connectivity data structure to maintain a sparse subgraph preserving 2-edge connectivity and then using the existing 2-edge connectivity data structure from Holm et al. [14] as a black box on that subgraph.

The best lower bound known is by Pătraşcu et al. [21], which shows a trade-off between update time t_u and query time t_q of $t_q \lg \frac{t_u}{t_q} = \Omega(\lg n)$ and $t_u \lg \frac{t_q}{t_u} = \Omega(\lg n)$.

1.1 Our results

We obtain an update time of $\mathcal{O}((\log n)^2(\log\log n)^2)$ and a query time of $\mathcal{O}(\log n/\log\log n)$ for the bridge finding problem:

Theorem 1. *There exists a deterministic data structure for dynamic multigraphs in the word RAM model with $\Omega(\log n)$ word size, that uses $\mathcal{O}(m+n)$ space, and can handle the following updates, and queries for arbitrary vertices v or arbitrary connected vertices v, u:*

- *insert and delete edges in* $\mathcal{O}((\log n)^2(\log\log n)^2)$ *amortized time,*
- *find a bridge in* v*'s connected component or determine that none exists, or find a bridge that separates* u *from* v *or determine that none exists. Both in* $\mathcal{O}(\log n/\log\log n)$ *worst-case time.*
- *find the size of* v*'s connected component in* $\mathcal{O}(\log n/\log\log n)$ *worst-case time, or the size of its* 2*-edge connected component in* $\mathcal{O}(\log n(\log\log n)^2)$ *worst-case time.*

Since a pair of connected vertices are 2-edge connected exactly when there is no bridge separating them, we have the following corollary:

Corollary 2. *There exists a data structure for dynamic multigraphs in the word RAM model with* $\Omega(\log n)$ *word size, that can answer* 2*-edge connectivity queries in* $\mathcal{O}(\log n/\log\log n)$ *worst case time and handle insertion and deletion of edges in* $\mathcal{O}((\log n)^2(\log\log n)^2)$ *amortized time, with space consumption* $\mathcal{O}(m+n)$*.*

Note that the query time is optimal with respect to the trade-off by Pătraşcu et al. [21]
As a stepping stone on the way to our main theorem, we show the following:

Theorem 3. *There exists a combinatorial deterministic data structure for dynamic multigraphs on the pointer-machine without the use of bit-tricks, that uses* $\mathcal{O}(m+n)$ *space, and can handle insertions and deletions of edges in* $\mathcal{O}((\log n)^3\log\log n)$ *amortized time, find bridges and determine connected component sizes in* $\mathcal{O}(\log n)$ *worst-case time, and find* 2*-edge connected component sizes in* $\mathcal{O}((\log n)^2\log\log n)$ *worst-case time.*

Our results are based on modifications to the 2-edge connectivity data structure from [14]. Applying the analoguous modification to the biconnectivity data structure from the same paper yields a structure with $\mathcal{O}((\log n)^3(\log\log n)^2)$ amortized update time and $\mathcal{O}((\log n)^2(\log\log n)^2)$ worst case query time. The details of this modification are beyond the scope of this paper.

1.2 Applications

Although our data structure is deterministic and uses linear space, it entails an improvement of the current best sublinear-space data structure. Namely, the Monte-Carlo randomized sublinear-space 2-edge connectivity data structure by Gibbs et al. [10] uses the data structure from [14] as a black box: For each update the data structure uses worst case $O(\log^5 n)$ time by itself, and makes $O(\log^2 n)$ updates in the sparse graph seen by the black box. Thus, with the $O(\log^4 n)$ amortized update time from [14], this gives a sublinear-space data structure with amortized $O(\log^6 n)$ update time. Using the data structure from [23] or our new purely combinatorial data structure, this drops to $\mathcal{O}((\log n)^5\log\log n)$ amortized time. With our new $\tilde{O}(\log^2 n)$ update time data structure, this improves to $\mathcal{O}(\log^5 n)$ amortized time (and the bottleneck is now in the reduction).

While dynamic graphs are interesting in their own right, many algorithms and theorem proofs for static graphs rely on decremental or incremental graphs. Take for example the problem of whether or not a graph has a unique perfect matching. The following theorem by Kotzig immediately yields a near-linear time algorithm if implemented together with a decremental 2-edge connectivity data structure with poly-logarithmic update time:

Theorem 4 (A. Kotzig '59 [19])**.** *Let* G *be a connected graph with a unique perfect matching* M*. Then* G *has a bridge that belongs to* M*.*

The near-linear algorithm for finding a unique perfect matching by Gabow, Kaplan, and Tarjan [9] is straight-forward: Find a bridge and delete it. If deleting it yields connected components of odd size, it must belong to the matching, and all edges incident to its endpoints may be deleted—if the components have even size, the bridge cannot belong to the matching. Recurse on the components. Thus, to implement Kotzig's Theorem, one has to implement three operations: One that finds a bridge, a second that deletes an edge, and a third returning the size of a connected component.

Another example is Petersen's theorem [22] which states that any cubic, 2-edge connected graph contains a perfect matching. An algorithm by Biedl et al. [3] finds a perfect matching in such graphs in $\mathcal{O}(n \log^4 n)$ time, by using the Holm et al 2-edge connectivity data structure as a subroutine. In fact, one may implement their algorithm and obtain running time $\mathcal{O}(nf(n))$, by using as subroutine a data structure for amortized decremental 2-edge connectivity with update-time $f(n)$. Here, we thus improve the running time from $\mathcal{O}(n(\log n)^3 \log\log n)$ to $\mathcal{O}(n(\log n)^2(\log\log n)^2)$.

In 2010, Diks and Stanczyk [4] improved Biedl et al.'s algorithm for perfect matchings in 2-edge connected cubic graphs, by having it rely only on dynamic connectivity, not 2-edge connectivity, and thus obtaining a running time of $\mathcal{O}(n(\log n)^2/\log\log n)$ for the deterministic version, or $\mathcal{O}(n \log n(\log\log n)^2)$ expected running time for the randomized version. However, our data structure still yields a direct improvement to the original algorithm by Biedl et al.

Note that all applications to static graphs have in common that it is no disadvantage that our running time is amortized.

1.3 Techniques

As with the previous algorithms, our result is based on top trees [2] which is a hierarchical tree structure used to represent information about a dynamic tree — in this case, a certain spanning tree of the dynamic graph. The original $\mathcal{O}((\log n)^4)$ algorithm of Holm et al. [14] stores $\mathcal{O}((\log n)^2)$ counters with each top tree node, where each counter represent the size of a certain subgraph. Our new $\mathcal{O}((\log n)^3)$ algorithm applies top trees the same way, representing the same $\mathcal{O}((\log n)^2)$ sizes with each top tree node, but with a much more efficient implicit representation of the sizes.

Reanalyzing the algorithm of Holm et al. [14], we show that many of the sizes represented in the top nodes are identical, which implies that that they can be represented more efficiently as a list of actual differences. We then need additional data structures to provide the desired sizes, and we have to be very careful when we move information around as the top tree changes, but overall, we gain almost a log-factor in the amortized time bound, and the algorithm remains purely combinatorial.

Our combinatorial improvement can be composed with the bittrick improvement of Thorup [23]. Thorup represents the same sizes as the original algorithm of Holm et al., but observes that we don't need the exact sizes, but just a constant factor approximation. Each approximate size can be represented with only $\mathcal{O}(\log\log n)$ bits, and we can therefore pack $\Omega(\log n/\log\log n)$ of them together in a single $\Omega(\log n)$-bit word. This can be used to reduce the cost of adding two $\mathcal{O}(\log n)$-dimensional vectors of approximate sizes from $\mathcal{O}(\log n)$ time to $\mathcal{O}(\log\log n)$ time. It may not be obvious from the current presentation, but it was a significant technical difficulty when developing our $\mathcal{O}((\log n)^3 \log\log n)$ algorithm to make sure we could apply this technique and get the associated speedup to $\mathcal{O}((\log n)^2(\log\log n)^2)$.

The "natural" query time of our algorithm is the same as its update time. In order to reduce the query time, we observe that we can augment the main algorithm to maintain a secondary structure that can answer queries much faster. This can be used to reduce the query time for the combinatorial algorithm to $\mathcal{O}(\log n)$, and for the full algorithm to the optimal $\mathcal{O}(\log n/\log\log n)$.

The secondary structure needed for the optimal $\mathcal{O}(\log n/\log\log n)$ query time uses top trees of degree $\mathcal{O}(\log n/\log\log n)$. While the use of non-binary trees is nothing new, we believe we are the first to show that such top trees can be maintained in the "natural" time.

Finally, we show a general technique for getting down to linear space, using top trees whose base clusters have size $\Theta(\log^c n)$.

1.4 Article outline

In Section 2, we recall how [14] fundamentally solves 2-edge connectivity via a reduction to a certain set of operations on a dynamic forest. In Section 3, we recall how top trees can be used to maintain information in a dynamic forest, as shown in [2]. In Sections 4, 5, and 6, we describe how to support the operations on a dynamic tree needed to make a combinatorial $\mathcal{O}((\log n)^3 \log\log n)$ algorithm for bridge finding, as stated in Theorem 3. Then, in Section 7, we show how to use Approximate Counting to get down to $\mathcal{O}((\log n)^2(\log\log n)^2)$ update time, thus, reaching the update time of Theorem 1. We then revisit top trees in Section 8, and introduce the notion of B-ary top trees, as well as a general trick to save space in complex top tree applications. We proceed to show how to obtain the optimal $\Theta(\log n/\log\log n)$ query time in Section 9. Finally, in Section 10, we show how to achieve optimal space, by only storing cluster information with large clusters, and otherwise calculating it from scratch when needed.

2 Reduction to operations on dynamic trees

In [14], 2-edge connectivity was maintained via operations on dynamic trees, as follows. For each edge e of the graph, the algorithm explicitly maintains a *level*, $\ell(e)$, between 0 and $\ell_{\max} = \lfloor \log_2 n \rfloor$ such that the edges at level $\ell_{\max}$ form a spanning forest T, and such that the 2-edge connected components in the subgraph induced by edges at level at least i have at most $\lfloor n/2^i \rfloor$ vertices. For each edge e in the spanning forest, define the *cover level*, $c(e)$, as the maximum level of an edge crossing the cut defined by removing e from T, or -1 if no such edge exists. The cover levels are only maintained implicitly, because each edge insertion and deletion can change the cover levels of $\Omega(n)$ edges. Note that the bridges are exactly the edges in the spanning forest with cover level -1. The algorithm explicitly maintains the spanning forest T using a dynamic tree structure supporting the following operations:

1. Link(v, w). Add the edge (v, w) to the dynamic tree, implicitly setting its cover level to -1.
2. Cut(v, w). Remove the edge (v, w) from the dynamic tree.
3. Connected(v, w). Returns **true** if v and w are in the same tree, **false** otherwise.
4. Cover(v, w, i). For each edge e on the tree path from v to w whose cover level is less than i, implicitly set the cover level to i. (Called when an edge was inserted at level $i = 0$ or had its level incresed to $i > 0$.)
5. Uncover(v, w, i). For each edge e on the tree path from v to w whose cover level is at most i, implicitly set the cover level to -1. (Called when the knowledge we had about whether the edges on the tree path from v to w were covered at level $\leq i$ is no longer valid because some edge was deleted. This may temporarily set some cover levels too low, but the algorithm fixes that using subsequent calls to Cover.)
6. CoverLevel(v). Return the minimal cover level of any edge in the tree containing v.

7. CoverLevel(v, w). Return the minimal cover level of an edge on the path from v to w. If $v = w$, we define CoverLevel$(v, w) = \ell_{\max}$.
8. MinCoveredEdge(v). Return any edge in the tree containing v with minimal cover level. (Find a bridge, anywhere in the tree.)
9. MinCoveredEdge(v, w). Returns a tree-edge on the path from v to w whose cover level is CoverLevel(v, w). (Find a bridge on the given path.)
10. AddLabel(v, l, i). Associate the *user label* l to the vertex v at level i. (Insert a non-tree edge.)
11. RemoveLabel(l). Remove the user label l from its vertex vertex(l). (Delete a non-tree edge.)
12. FindFirstLabel(v, w, i). Find a user label at level i such that the associated vertex u has[1] CoverLevel$(u, \text{meet}(u, v, w)) \geq i$; among such user labels, return the one that minimizes the distance from v to meet(u, v, w). (Find the first candidate witness that part of the path $v \cdots w$ is covered on level i, or a non-tree edge to swap with when deleting a covered tree edge.)
13. FindSize(v, w, i). Find the number of vertices u such that CoverLevel$(u, \text{meet}(u, v, w)) \geq i$. (Determine the size of the 2-edge connected component at level i that would result from increasing the level of (v, w) to i, or for $v = w$ just find the size of the 2-edge connected component at level i that contains $v = w$. Thus the size of the 2-edge component of v in the whole graph is FindSize$(v, v, 0)$.). Note that FindSize$(v, v, -1)$ is just the number of vertices in the tree containing v (which is also the size of the connected component of v).

Lemma 5 (Essentially the high level algorithm from [14])**.** *There exists a deterministic reduction for dynamic graphs with n nodes, that, when starting with an empty graph, supports any sequence of m Insert or Delete operations using:*

- $\mathcal{O}(m)$ *calls to Link, Cut, Uncover, and CoverLevel.*
- $\mathcal{O}(m \log n)$ *calls to Connected, Cover, AddLabel, RemoveLabel, FindFirstLabel, and FindSize.*

And that can answer FindBridge queries using a constant number of calls to Connected, CoverLevel, and MinCoveredEdge, and size queries using a single call to FindSize.

Proof. See Appendix A for a proof and pseudocode. □

The algorithm in [14] used a dynamic tree structure supporting all the operations in $\mathcal{O}((\log n)^3)$ time, leading to an $\mathcal{O}((\log n)^4)$ algorithm for bridge finding. Thorup [23] showed how to improve the time for the dynamic tree structure to $\mathcal{O}((\log n)^2 \log\log n)$ leading to an $\mathcal{O}((\log n)^3 \log\log n)$ algorithm for bridge finding.

Throughout this paper, we will show a number of data structures for dynamic trees, implementing various subsets of these operations while ignoring the rest (See Table 1). Define a *CoverLevel* structure to be one that implements operations 1–9, and a *FindSize* structure to be a CoverLevel structure that additionally implements the FindSize operation. Finally, we define a *FindFirstLabel* structure to be one that implements operations 1–12 (all except for FindSize).

The point is that we can get different trade-offs between the operation costs in the different structures, and that we can combine them into a single structure supporting all the operations using the following

[1] meet(u, v, w) is defined as the unique vertex that is on all simple paths beween any two of u, v, and w.

#	Operation	Asymptotic worst case time per call, using structure in section				
		4	5	6	7	9
1	Link(v, w, e)	$\log n$	$(\log n)^2 \log\log n$	$\log n \log\log n$	$\log n(\log\log n)^2$	$\frac{f(n)\log n}{\log f(n)}$
2	Cut(e)					
3	Connected(v, w)					$\frac{\log n}{\log f(n)}$
4	Cover(v, w, i)					
5	Uncover(v, w, i)					
6	CoverLevel(v)					
7	CoverLevel(v, w)					
8	MinCoveredEdge(v)					
9	MinCoveredEdge(v, w)					
10	AddLabel(v, l, i)	-	-	$\log n \log\log n$	-	-
11	RemoveLabel(l)					
12	FindFirstLabel(v, w, i)					
13	FindSize(v, w, i)	-	$(\log n)^2 \log\log n$	-	$\log n(\log\log n)^2$	-
	FindSize$(v, v, -1)$	$\log n$	$\log n$	$\log n$	$\log n$	$\frac{\log n}{\log f(n)}$
		Space cost, using structure in section				
natively		n	$n \log n$	$m + n$	$n \log\log n$	n
when modified as in Section 10			n		n	

Table 1: Data structures presented in this paper. In the last column, $f(n) \in \mathcal{O}(\frac{\log n}{\log\log n})$ can be chosen arbitrarily.

Lemma 6 (Folklore)**.** *Given two data structures S and S' for the same problem consisting of a set U of update operations and a set Q of query operations. If the respective update times are $f_u(n)$ and $f'_u(n)$ for $u \in U$, and the query times are $g_q(n)$ and $g'_q(n)$ for $q \in Q$, we can create a combined data structure running in $\mathcal{O}(f_u(n) + f'_u(n))$ time for update operation $u \in U$, and $\mathcal{O}(\min\{g_q(n), g'_q(n)\})$ time for query operation $q \in Q$.*

Proof. Simply maintain both structures in parallel. Call all update operations on both structures, and call only the fastest structure for each query. □

Proof of Theorem 3. Use the CoverLevel structure from Section 4, the FindSize structure from Section 5, and the FindFirstLabel structure from Section 6, and combine them into a single structure using Lemma 6. Then the reduction from Lemma 5 gives the correct running times but uses $\mathcal{O}(m + n \log n)$ space. To get linear space, modify the FindSize and FindFirstLabel structures as described in Section 10. □

Proof of Theorem 1. Use the CoverLevel structure from Section 9, the FindSize structure from Section 5, as modified in Section 7 and 10, and the FindFirstLabel structure from Section 6, and combine them into a single structure using Lemma 6. Then the reduction from Lemma 5 gives the required bounds. □

3 Top trees

A *top tree* is a data structure for maintaining information about each tree of a dynamic forest. Let T be a tree, and let ∂T be an arbitrary set of 1 or 2 vertices of T, which we will call the *external boundary vertices* of T. For any subgraph S of T, define the *boundary vertices* of S (denoted $\partial_{(T,\partial T)} S$

or just ∂S) as the set of vertices in S that are either in ∂T or are incident to an edge not in S. A cluster C is a connected subgraph of T with 1 or 2 boundary vertices[2]. A top tree $\mathcal{T}$ is a rooted tree representing a recursive partition of T into clusters. The root of $\mathcal{T}$ corresponds to all of T, and each non-leaf node is an edge-disjoint union of the clusters of its children. The leaves of $\mathcal{T}$ are called *base clusters* and (usually[3]) correspond to the edges of T.

For every cluster C the *cluster path* of C, denoted $\pi(C)$, is the tree path in T connecting ∂C. If $|\partial C| = 2$ then $\pi(C)$ contains at least one edge, and we call C a *path cluster*. Otherwise $|\pi(C)| = 1$ and we call C a *point cluster*. If $|\partial C| = 1$ then $\pi(C)$ is the trivial path consisting of the single boundary vertex.

A top tree is *binary* if each node has at most two children. We call a non-leaf node *heterogeneous* if it has both a point cluster and a path cluster among its children, and *homogeneous* otherwise.

A path cluster D is called a *path child* of its parent C if $\pi(D) \subseteq \pi(C)$. Note that for binary top trees, a path cluster D is a path child if and only if its parent C is also a path cluster. But for non-binary top trees, even if C and D are both path clusters, ∂D may intersect ∂C only in a point, or not at all.

The top forest supports dynamic changes to the forest: insertion (link) or deletion (cut) of edges. Furthermore, it supports the *expose* operation: expose(v), or expose(v_1, v_2), returns a top tree where v, or v_1, v_2, are external boundary vertices. All operations are supported by performing a series of *destroy*, *create*, *split*, and *merge* operations: *split* destroys a node of the top tree and replaces it with its children, while merge creates a parent as a union of its children. Destroy and create are the base cases for split and merge, respectively. Note that clusters can only be merged if they are edge-disjoint and their union is a cluster (i.e. is connected and has a boundary of size at most 2).

Theorem 7 (Alstrup, Holm, de Lichtenberg, Thorup [2])**.** *For a dynamic forest on n vertices we can maintain binary top trees of height $\mathcal{O}(\log n)$ supporting each link, cut or expose with a sequence of $\mathcal{O}(1)$ calls to create or destroy, and $\mathcal{O}(\log n)$ calls to merge or split. These top tree modifications are identified in $\mathcal{O}(\log n)$ time. The space usage of the top trees is linear in the size of the dynamic forest.*

4 A CoverLevel structure

In this section we show how to maintain a top tree supporting the CoverLevel operations. This part is essentially the same as in [13, 14] (with minor corrections), but is included here for completeness because the rest of the paper builds on it. Pseudocode for maintaining this structure is given in Appendix B.

For each cluster C we want to maintain the following two integers and up to two edges:

$$\begin{aligned}
\text{cover}_C &:= \min\{c(e) \mid e \in \pi(C)\} \cup \{\ell_{\max}\} \\
\text{globalcover}_C &:= \min\{c(e) \mid e \in C \setminus \pi(C)\} \cup \{\ell_{\max}\} \\
\text{minpathedge}_C &:= \operatorname*{arg\,min}_{e \in \pi(C)} c(e) \text{ if } |\partial C| = 2, \text{ and } \textbf{nil} \text{ otherwise} \\
\text{minglobaledge}_C &:= \operatorname*{arg\,min}_{e \in C \setminus \pi(C)} c(e) \text{ if } C \neq \pi(C), \text{ and } \textbf{nil} \text{ otherwise}
\end{aligned}$$

[2]Note that this deviates from the existing literature, which introduces a special class of cluster with 0 boundary vertices, which can only be present in the root [2]

[3]We will look at generalized top trees where this is not the case in Section 8

Then

$$\left.\begin{aligned}\text{CoverLevel}(v) &= \text{globalcover}_C\\ \text{MinCoveredEdge}(v) &= \text{minglobaledge}_C\end{aligned}\right\}\text{ where } C \text{ is the point cluster returned by Expose}(v)$$

$$\left.\begin{aligned}\text{CoverLevel}(v,w) &= \text{cover}_C\\ \text{MinCoveredEdge}(v,w) &= \text{minpathedge}_C\end{aligned}\right\}\text{ where } C \text{ is the path cluster returned by Expose}(v,w)$$

The problem is that when handling Cover or Uncover we cannot afford to propagate the information all the way down to the edges. When these operations are called on a path cluster C, we instead implement them directly in C, and then store "lazy information" in C about what should be propagated down in case we want to look at the descendants of C. The exact additional information we store for a path cluster C is

$$\begin{aligned}\text{cover}_C^- &:= \text{max level of a pending Uncover, or } -1\\ \text{cover}_C^+ &:= \text{max level of a pending Cover, or } -1\end{aligned}$$

We maintain the invariant that $\text{cover}_C \geq \text{cover}_C^+$, and if $\text{cover}_C \leq \text{cover}_C^-$ then $\text{cover}_C = \text{cover}_C^+$.

This allows us to implement $\text{Cover}(v,w,i)$ by first calling $\text{Expose}(v,w)$, and then updating the returned path cluster C as follows:

$$\text{cover}_C = \max\{\text{cover}_C, i\} \qquad \text{cover}_C^+ = \max\left\{\text{cover}_C^+, i\right\}$$

Similarly, we can implement $\text{Uncover}(v,w,i)$ by first calling $\text{Expose}(v,w)$, and then updating the returned path cluster C as follows if $\text{cover}_C \leq i$:

$$\text{cover}_C = -1 \qquad \text{cover}_C^+ = -1 \qquad \text{cover}_C^- = \max\left\{\text{cover}_C^-, i\right\}$$

Together, cover_C^- and cover_C^+ represent the fact that for each path descendant D of C, if $\text{cover}_D \leq \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$[4], we need to set $\text{cover}_D = \text{cover}_C^+$. In particular whenever a path cluster C is split, for each path child D of C, if $\max\left\{\text{cover}_D, \text{cover}_D^-\right\} \leq \text{cover}_C^-$ we need to set

$$\text{cover}_D^- = \text{cover}_C^-$$

Furthermore, if $\text{cover}_D \leq \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$ we need to set

$$\text{cover}_D = \text{cover}_C^+ \qquad \text{cover}_D^+ = \text{cover}_C^+$$

Note that only cover_D is affected. None of globalcover_D, minpathedge_D, or minglobaledge_D depend directly on the lazy information.

Now suppose we have k clusters[5] $A_1, \ldots, A_k$ that we want to merge into a single new cluster C. For $1 \leq i \leq k$ define

$$\text{globalcover}'_{C,A_i} := \begin{cases}\text{globalcover}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \text{ or } \text{globalcover}_{A_i} \leq \text{cover}_{A_i}\\ \text{cover}_{A_i} & \text{otherwise}\end{cases}$$

$$\text{minglobaledge}'_{C,A_i} := \begin{cases}\text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \text{ or } \text{globalcover}_{A_i} \leq \text{cover}_{A_i}\\ \text{minpathedge}_{A_i} & \text{otherwise}\end{cases}$$

[4] In [13, 14] this condition is erroneously stated as $\text{cover}_D \leq \text{cover}_C^-$.

[5] $k = 2$ for now, but we will reuse this in section 9 with a higher-degree top tree.

Note that for a point-cluster A_i, globalcover_{A_i} is always $\leq \text{cover}_{A_i} = l_{\max}$.

We then have the following relations between the data of the parent and the data of its children:

$$\begin{aligned}
\text{cover}_C &= \ell_{\max} \text{ if } |\partial C| < 2, \text{ otherwise } \min_{1\leq i<k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\
\text{minpathedge}_C &= \textbf{nil} \quad \text{if } |\partial C| < 2, \text{ otherwise } \text{minpathedge}_{A_j} \text{ where } j = \operatorname*{arg\,min}_{1\leq i<k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\
\text{globalcover}_C &= \min_{1\leq i<k} \text{globalcover}'_{C,A_i} \\
\text{minglobaledge}_C &= \text{minglobaledge}'_{C,A_j} \quad \text{where } j = \operatorname*{arg\,min}_{1\leq i<k} \text{globalcover}'_{C,A_i} \\
\text{cover}^-_C &= -1 \\
\text{cover}^+_C &= -1
\end{aligned}$$

Analysis For any constant-degree top tree, Merge and Split with this information takes constant time, and thus, all operations in the CoverLevel structure in this section take $\mathcal{O}(\log n)$ time. Each cluster uses $\mathcal{O}(1)$ space, so the total space used is $\mathcal{O}(n)$.

5 A FindSize structure

We now proceed to show how to extend the CoverLevel structure from Section 4 to support FindSize in $\mathcal{O}(\log n \log\log n)$ time per Merge and Split. Later, in Section 7 we will show how to reduce this to $\mathcal{O}((\log\log n)^2)$ time per Merge and Split. See Appendix C for pseudocode.

We will use the idea of having a single *vertex label* for each vertex, which is a point cluster with no edges, having that vertex as boundary vertex and containing all relevant information about the vertex. The advantage of this is that it simplifies handling of the common boundary vertex during a merge by making sure it is uniquely assigned to (and accounted for by) one of the children.

Let C be a cluster in T, let v be a vertex in $\pi(C)$, and let $0 \leq i < \ell_{\max}$. Define

$$\text{pointset}_{C,v,i} := \left\{ u \in C \;\middle|\; \begin{array}{c} \pi(C) \cap u \cdots v = \{v\} \\ \wedge\ \text{CoverLevel}(u,v) \geq i \end{array} \right\}$$

Intuitively, $\text{pointset}_{C,v,i}$ is the set of vertices in C whose path to v is covered at level $\geq i$ independently of the cover levels on $\pi(C)$. Information (such as the size or the existence of certain marked vertices) about this set stays constant for as long as C exists, no matter what happens with the lazy information in the ancestors to C. In this section we only care about the size

$$\text{pointsize}_{C,v,i} := \left|\text{pointset}_{C,v,i}\right|$$

For convenience, we will combine all the $\mathcal{O}(\log n)$ levels together into a single vector[6]

$$\text{pointsize}_{C,v} := \left(\text{pointsize}_{C,v,i}\right)_{\{0\leq i<\ell_{\max}\}}$$

Then we can define the vector

$$\text{size}_C := \sum_{u\in\pi(C)} \text{pointsize}_{C,u}$$

[6] All vectors and matrices in this section have indices ranging from 0 to $\ell_{\max} - 1$.

Note that with this definition, if $\partial C = \{v\}$ then $\text{pointsize}_{C,v} = \text{size}_C$ so even when $v = w$ we have

$$\text{FindSize}(v, w, i) = \text{size}_{C,i} \qquad \text{where } C = \text{Expose}(v, w)$$

So for any cluster C, the size_C vector is what we want to maintain.

The main difficulty turns out be computing the size_C vector for the heterogeneous point clusters. To help with that we will for each cluster C and boundary vertex $v \in \partial C$ break $\pi(C)$ into $\ell_{\max} + 2$ *parts*. For each $-1 \le i \le \ell_{\max}$ define

$$\text{partpath}_{C,v,i} := \{u \in \pi(C) \mid \text{CoverLevel}(u, v) = i\}$$

Then $\text{partpath}_{C,v,i}$ (if nonempty) is a contiguous subset of the vertices on $\pi(C)$. Furthermore, $\text{partpath}_{C,v,\ell_{\max}} = \{v\}$, $\partial C \setminus \{v\} \subseteq \text{partpath}_{C,v,-1}$, and for all $0 \le i < \ell_{\max}$ the set $\text{partpath}_{C,v,i}$ lies between the closest edge e to v with $c(e) \le i$ and the closest edge e' to v with $c(e') < i$. In addition to the size_C vector, we will maintain the following two size vectors for each part:

$$\text{partsize}_{C,v,i} := \sum_{u \in \text{partpath}_{C,v,i}} \text{pointsize}_{C,u} \qquad \text{diagsize}_{C,v,i} := M(i) \cdot \text{partsize}_{C,v,i}$$

Where $M(i)$ is a diagonal matrix whose entries are defined by[7]

$$M(i)_{jj} = [j \le i]$$

The $M(i)$ matrix is purely a notational convenience whose purpose is to "zero out" some elements in a vector. In particular, for $0 \le j < \ell_{\max}$

$$\text{diagsize}_{C,v,i,j} = (M(i) \cdot \text{partsize}_{C,v,i})_j = \begin{cases} \text{partsize}_{C,v,i,j} & \text{if } j \le i \\ 0 & \text{otherwise} \end{cases}$$

Note that these vectors do not take cover_C^- and cover_C^+ (as defined in Section 4) into account. The corresponding "clean" vectors are not explicitly stored, but computed when needed as follows

$$\left.\begin{aligned} \text{partsize}'_{C,v,i} &= \begin{cases} \text{partsize}_{C,v,i} & \text{if } i > \ell \\ \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ \vec{0} & \text{otherwise} \end{cases} \\ \text{diagsize}'_{C,v,i} &= \begin{cases} \text{diagsize}_{C,v,i} & \text{if } i > \ell \\ M(i) \cdot \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ \vec{0} & \text{otherwise} \end{cases} \end{aligned}\right\} \text{where } \ell = \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$$

The point of these definitions is that each path cluster inherits most of its partsize and diagsize vectors from its children, and we can use this fact to get an $\mathcal{O}(\ell_{\max}/\log \ell_{\max}) = \mathcal{O}(\log n/\log\log n)$ speedup compared to [14].

[7] Here, $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the *Iverson Bracket* (see [17]).

Merging along a path (the general case) Let A, B be clusters that we want to merge into a new cluster C, and suppose $\partial A \cup \partial B \subseteq \pi(C)$. This covers both types of homogeneous merges (two point or two path clusters), as well as the heterogeneous merge (one point and one path cluster) where the result is a path cluster. The only type of merge not covered is the heterogeneous merge resulting in a point cluster, which is handled in the next section. Let $\partial A \cap \partial B = \{c\}$. If $|\partial C| = 1$, let $a = b = c$, otherwise let $\partial C = \{a, b\}$ with $a \in \partial A$, $b \in \partial B$. Then

$$\begin{aligned}
\text{size}_C &= \text{size}_A + \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{partsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{partsize}'_{A,a,i} + \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{partsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases} \\
\text{diagsize}_{C,a,i} &= \begin{cases} \text{diagsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{diagsize}'_{A,a,i} + M(i) \cdot \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{diagsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases}
\end{aligned}$$

The formulas for $\text{partsize}_{C,b,i}$ and $\text{diagsize}_{C,b,i}$ are analoguous. The important thing to note is that if we have already computed and stored the $\text{partsize}'_{A,a,i}$, $\text{partsize}'_{B,c,i}$, $\text{diagsize}'_{A,a,i}$, and $\text{diagsize}'_{B,c,i}$ vectors for all i, then the only new value we need to compute is for $i = \text{cover}_A$. The rest can be inherited.

Merging off the path (heterogeneous point clusters) Now let A be a path cluster with $\partial A = \{a, b\}$, let B be a point cluster with $\partial B = \{b\}$, and suppose we want to merge A, B into a new point cluster C with $\partial C = \{a\}$. Then

$$\begin{aligned}
\text{size}_C &= \left(\sum_{i=-1}^{\ell_{\max}} \text{diagsize}'_{A,a,i} \right) + M(\text{cover}_A) \cdot \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{size}_C & \text{if } i = \ell_{\max} \\ \vec{0} & \text{otherwise} \end{cases} \\
\text{diagsize}_{C,a,i} &= \text{partsize}_{C,a,i}
\end{aligned}$$

Analysis The advantage of our new approach is that each merge or split is a *constant* number of splits, concatenations, searches, and sums over $\mathcal{O}(\ell_{\max})$-length lists of $\ell_{\max}$-dimensional vectors. By representing each list as an augmented balanced binary search tree (see e.g. [18, pp. 471–475]), we can implement each of these operations in $\mathcal{O}(\ell_{\max} \log \ell_{\max})$ time, and using $\mathcal{O}(\ell_{\max})$ space per cluster, as follows. Let C be a cluster and let $v \in \partial C$. The tree has one node for each key $i, -1 \leq i \leq \ell_{\max}$

such that $\text{partsize}_{C,v,i}$ is nonzero, augmented with the following additional information:

$$\begin{aligned}
\text{key} &:= i \\
\text{partsize} &:= \text{partsize}_{C,v,i} \\
\text{diagsize} &:= \text{diagsize}_{C,v,i} \\
\text{partsizesum} &:= \sum_{j \text{ descendant of } i} \text{partsize}_{C,v,j} \\
\text{diagsizesum} &:= \sum_{j \text{ descendant of } i} \text{partsize}_{C,v,j}
\end{aligned}$$

Each split, concatenate, search, or sum operation can be implemented such that it touches $\mathcal{O}(\log \ell_{\max})$ nodes, and the time for each node update is dominated by the time it takes to add two $\ell_{\max}$-dimensional vectors, which is $\mathcal{O}(\ell_{\max})$. The total time for each Cover, Uncover, Link, Cut, or FindSize is therefore $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max}) = \mathcal{O}((\log n)^2 \log\log n)$, and the total space used for the structure is $\mathcal{O}(n \cdot \ell_{\max}) = \mathcal{O}(n \log n)$.

Comparison to previous algorithms For any path cluster C and vertex $v \in \partial C$, let $S_{C,v}$ be the matrix whose jth column $0 \le j < \ell_{\max}$ is defined by

$$(S^T_{C,v})_j := \sum_{k=j}^{\ell_{\max}} \text{partsize}'_{C,v,k}$$

Then $S_{C,v}$ is essentially the size matrix maintained for path clusters in [13, 14, 23]. Notice that

$$\text{diag}(S_{C,v}) = \sum_{k=-1}^{\ell_{\max}} \text{diagsize}'_{C,v,k}$$

which explains our choice of the "diag" prefix.

6 A FindFirstLabel structure

We will show how to maintain information that allows us to implement FindFirstLabel; the function that allows us to inspect the replacement edge candidates at a given level. The implementation uses a "destructive binary search, with undo" strategy, similar to the non-local search introduced in [2].

The idea is to maintain enough information in each cluster to determine if there is a result. Then we can start by using $\text{Expose}(v, w)$, and repeatedly split the root containing the answer until we arrive at the correct label. After that, we simply undo the splits (using the appropriate merges), and finally undo the Expose.

Just as in the FindSize structure, we will use vertex labels to store all the information pertinent to a vertex. We store all the added *user labels* for each vertex in the label object for that vertex in the base level of the top tree. For each level where the vertex has an associated user label, we keep a doubly linked list of those labels, and we keep a singly-linked list of these nonempty lists. Thus, $\text{FindFirstLabel}(v, w, i)$ boils down to finding the first vertex label that has an associated user label at the right level. Once we have that vertex label, the desired user label can be found in $\mathcal{O}(\ell_{\max})$ time.

Let C be a cluster in T, and let $v \in \partial C$. Define bit vectors[8]

$$\text{pointincident}_{C,v} := \left(\left[\exists v \in \text{pointset}_{C,v,i} \colon \begin{matrix} v \text{ has labels} \\ \text{at level } i \end{matrix}\right]\right)_{\{0 \le i < \ell_{\max}\}}$$
$$\text{incident}_C := \bigvee_{u \in \pi(C)} \text{pointincident}_{C,u}$$

Maintaining the incident_C bit vectors, and the corresponding $\text{partincident}_{C,v}$ and $\text{diagincident}_{C,v}$ bit vectors, can be done completely analogous to the way we maintain the size vectors used for FindSize, with the minor change that we use bitwise OR on bit vectors instead of vector addition.

Updating the vertex label cluster C in the top tree during AddLabel(v, l, i), or a RemoveLabel(l) where $v = \text{vertex}(l)$ and $\ell(l) = i$ can be done by first calling detach(C), then updating the linked lists containing the user labels and setting

$$\text{incident}_C = ([v \text{ has labels at level } j])_{\{0 \le j < \ell_{\max}\}}$$
$$\text{partincident}_{C,v,i} = \begin{cases} \text{incident}_C & \text{if } i = \ell_{\max} \\ \vec{0} & \text{otherwise} \end{cases}$$
$$\text{diagincident}_{C,v} = \text{partincident}_C$$

and then reattaching C. Finally FindFirstLabel(v,w,i) can be implemented in the way already described, by examining $\text{pointincident}_{C,v,i}$ for each cluster. Note that even though we don't explicitly maintain it, for any cluster C and any $v \in \partial C$ we can easily compute

$$\begin{aligned} \text{pointincident}_{C,v} &= \bigvee_{i=-1}^{\ell_{\max}} \text{diagincident}'_{C,v,i} \\ &= \left(\bigvee_{i=\ell+1}^{\ell_{\max}} \text{diagincident}_{C,v,i}\right) + M(\text{cover}_C^+) \cdot \left(\bigvee_{i=-1}^{\ell} \text{partincident}_{C,v,i}\right) \\ &\quad \text{where } \ell := \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\} \end{aligned}$$

In general, let $A_1, \ldots, A_k$ be the clusters resulting from an expose or split, let $v, w \in \bigcup_{i=1}^k \partial A_i$

[8] Again, using the Iverson bracket.

(not necessarily distinct). Then we can define

$$\text{FindFirstLabel}((A_1,\ldots,A_k);v,w,i) = \begin{cases} \text{userlabels}_{v_x,i} & \text{if } A_x \text{ is a vertex label} \\ \text{FindFirstLabel}(\text{Split}(A_x);v_x,w_x,i) & \text{otherwise} \end{cases}$$

where for $1 \le j \le k$

$$v_j = \arg\min_{u\in\partial A_j} \text{dist}(v,u)$$

$$w_j = \arg\max_{u\in\partial A_j} \text{dist}(v,u)$$

and

$$I = \left\{ 1 \le j \le k \;\middle|\; \begin{array}{c} \text{CoverLevel}(v,v_j) \ge i \\ \wedge \quad \text{pointincident}_{A_j,v_j,i} = 1 \end{array} \right\}$$

$$x = \arg\min_{j\in I} \left(3\cdot\text{dist}(v,\text{meet}(v_j,v,w)) + |\partial A_j \cap v\cdots w|\right)$$

$$\text{FindFirstLabel}(v,w,i) = \text{FindFirstLabel}(\text{Expose}(v,w);v,w,i)$$

What happens here is that, for each j, the vertices v_j and w_j are the boundary vertices of A_j closest to v, and farthest from v, respectively. Thus, if A_j is a path cluster, $\partial A_j = \{v_j, w_j\}$, otherwise $\partial A_j = \{v_j\} = \{w_j\}$. The set I defined above is the set of indices of the clusters that contain labels at level i. Then, x is picked from I to minimize $D(x) = \text{dist}(v, \text{meet}(v_x, v, w))$. If there are more than one cluster minimizing $D(x)$, we prefer clusters with at most one boundary vertex on $\pi(C)$, since any vertex u in such a cluster will have $\text{dist}(v, \text{meet}(u, v, w)) = D(x)$, which is minimal. A path cluster A_x with both boundary vertices on $\pi(C)$ is only picked if it is the only cluster minimizing $D(x)$. In either case, we know that A_x contains a vertex u with the desired label, and that any vertex in A_x minimizing $\text{dist}(v_x, \text{meet}(u, v_x, w_x))$ will suffice. Now if A_x is a vertex label, it has only one vertex, and it stores the desired user label. Otherwise, we simply split A_x and recurse

Analysis By the method described in this section, AddLabel, RemoveLabel, and FindFirstLabel are maintained in $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max}) = \mathcal{O}((\log n)^2 \log\log n)$ worst-case time.

This can be reduced to $\mathcal{O}(\log n \cdot \log \ell_{\max}) = \mathcal{O}(\log n \log\log n)$ by realizing that each $\ell_{\max}$-dimensional bit vector fits into $\mathcal{O}(1)$ words, and that each bitwise OR therefore only takes constant time.

The total space used for a FindFirstLabel structure with n vertices and m labels is $\mathcal{O}(m+n)$ plus the space for $\mathcal{O}(n)$ bit vectors. If we assume a word size of $\Omega(\log n)$, this is just $\mathcal{O}(m+n)$ in total. If we disallow bit packing tricks, we may have to use $\mathcal{O}(m + n\cdot\ell_{\max}) = \mathcal{O}(m + n\log n)$ space.

7 Approximate counting

As noted in [23], we don't need to use the exact component sizes at each level. If s is the actual correct size, it is sufficient to store an approximate value s' such that $s' \le s \le e^\epsilon s'$, for some constant $0 < \epsilon < \ln 2$. Then we are no longer guaranteed that component sizes drop by a factor of $\frac{1}{2}$ at each level, but rather get a factor of $\frac{e^\epsilon}{2}$. This increases the number of levels to $\ell_{\max} = \lfloor \ln n/(\ln 2 - \epsilon)\rfloor$ (which is still $\mathcal{O}(\log n)$), but leaves the algorithm otherwise unchanged. Suppose we represent each size as a floating point value with a b-bit mantissa, for some b to be determined later. For each

addition of such numbers the relative error increases. The relative error at the root of a tree of additions of height h is $(1+2^{-b})^h \leq e^{2^{-b}h}$, thus to get the required precision it is sufficient to set $b = \log_2 \frac{h}{\epsilon}$. In our algorithm(s) the depth of calculation is clearly upper bounded by $h \leq h(n) \cdot \ell_{\max}$, where $h(n) = \mathcal{O}(\log n)$ is the height of the top tree. It follows that some $b \in \mathcal{O}(\log \log n)$ is sufficient. Since the maximum size of a component is n, the exponent has size at most $\lceil \log_2 n \rceil$, and can be represented in $\lceil \log_2 \lceil \log_2 n \rceil \rceil$ bits. Thus storing the sizes as $\mathcal{O}(\log \log n)$ bit floating point values is sufficient to get the required precision. Assuming a word size of $\Omega(\log n)$ this lets us store $\mathcal{O}(\frac{\log n}{\log \log n})$ sizes in a single word, and to add them in parallel in constant time.

Analysis We will show how this applies to our FindSize structure from Section 5. The bottlenecks in the algorithm all have to do with operations on $\ell_{\max}$-dimensional size vectors. In particular, the amortized update time is dominated by the time to do $\mathcal{O}(\log n \cdot \log \ell_{\max})$ vector additions, and $\mathcal{O}(\log n)$ multiplications of a vector by the $M(i)$ matrix. With approximate counting, the vector additions each take $\mathcal{O}(\log \log n)$ time. Multiplying a size vector x by $M(i)$ we get:

$$(M(i) \cdot x)_j = \begin{cases} x_j & \text{if } j \leq i \\ 0 & \text{otherwise} \end{cases}$$

And clearly this operation can also be done on $\mathcal{O}(\frac{\log n}{\log \log n})$ sizes in parallel when they are packed into a single word. With approximate counting, each multiplication by $M(i)$ therefore also takes $\mathcal{O}(\log \log n)$ time. Thus the time per operation is reduced to $\mathcal{O}(\log n (\log \log n)^2)$.

The space consumption of the data structure is $\mathcal{O}(n)$ plus the space needed to store $\mathcal{O}(n)$ of the $\ell_{\max}$-dimensional size vectors. With approximate counting that drops to $\mathcal{O}(\log \log n)$ per vector, or $\mathcal{O}(n \log \log n)$ in total.

Comparison to previous algorithms Combining the modified FindSize structure with the CoverLevel structure from Section 4 and the FindFirstLabel structure from Section 6 gives us the first bridge-finding structure with $\mathcal{O}((\log n)^2 (\log \log n)^2)$ amortized update time. This structure uses $\mathcal{O}(m + n \log \log n)$ space, and uses $\mathcal{O}(\log n)$ time for FindBridge and Size queries, and $\mathcal{O}(\log n (\log \log n)^2)$ for 2-size queries.

For comparison, applying this trick in the obvious way to the basic $\mathcal{O}((\log n)^4)$ time and $\mathcal{O}(m + n(\log n)^2)$ space algorithm from [13, 14] gives the $\mathcal{O}((\log n)^3 \log n)$ time and $\mathcal{O}(m + n \log n \log \log n)$ space algorithm briefly mentioned in [23].

8 Top trees revisited

We can combine the tree data structures presented so far to build a data structure for bridge-finding that has update time $\mathcal{O}((\log n)^2 (\log \log n)^2)$, query time $\mathcal{O}(\log n)$, and uses $\mathcal{O}(m + n \log \log n)$ space.

In order to get faster queries and linear space, we need to use top-trees in an even smarter way. For this, we need the full generality of the top trees described in [2].

8.1 Level-based top trees, labels, and fat-bottomed trees

As described in [2], we may associate a level with each cluster, such that the leaves of the top tree have level 0, and such that the parent of a level i cluster is on level $i + 1$. As observed in Alstrup et

al. [2, Theorem 5.1], one may also associate one or more *labels* with each vertex. For any vertex, v, we may handle the label(s) of v as point clusters with v as their boundary vertex and no edges. Furthermore, as described in [2], we need not have single edges on the bottom most level. We may generalize this to instead have clusters of *size* $\leq Q$, that is, with at most Q edges, as the leaves of the top tree.

Theorem 8 (Alstrup, Holm, de Lichtenberg, Thorup [2])**.** *Consider a fully dynamic forest and let Q be a positive integer parameter. For the trees in the forest, we can maintain levelled top trees whose base clusters are of size at most Q and such that if a tree has size s, it has height $h = \mathcal{O}(\log s)$ and $\lceil \mathcal{O}(s/(Q(1+\varepsilon)^i)) \rceil$ clusters on level $i \leq h$. Here, ε is a positive constant. Each link, cut, attach, detach, or expose operation is supported with $\mathcal{O}(1)$ creates and destroys, and $\mathcal{O}(1)$ joins and splits on each positive level. If the involved trees have total size s, this involves $\mathcal{O}(\log s)$ top tree modifications, all of which are identified in $\mathcal{O}(Q + \log s)$ time. For a composite sequence of k updates, each of the above bounds are multiplied by k. As a variant, if we have parameter S bounding the size of each underlying tree, then we can choose to let all top roots be on the same level $H = \mathcal{O}(\log S)$.*

8.2 High degree top trees

Top trees of degree two are well described and often used. However, it turns out to be useful to also consider top trees of higher degree B, especially for $B \in \omega(1)$.

Lemma 9. *Given any $Q \geq 1$ and $B \geq 2$, one can maintain top trees of degree B and height $\mathcal{O}(\log n / \log B)$ with base clusters of size at most Q. Each expose, link, or cut is handled by $\mathcal{O}(1)$ calls to create or destroy and $\mathcal{O}(\log n / \log B)$ calls to split or merge. The operations are identified in $\mathcal{O}(B(\log n / \log B) + Q)$ time.*

Proof. Given a binary levelled top tree $\mathcal{T}_2$ of height h with base clusters of size at most Q as in Theorem 8, we can create a B-ary levelled top tree $\mathcal{T}_B$, where the leaves of $\mathcal{T}_B$ are the leaves of $\mathcal{T}_2$, and where the clusters on level i of $\mathcal{T}_B$ are the clusters on level $i \cdot \lfloor \log_2 B \rfloor$ of $\mathcal{T}_2$. Edges in $\mathcal{T}_B$ correspond to paths of length $\lfloor \log_2 B \rfloor$ in $\mathcal{T}_2$. Thus, given a binary top tree, we may create a B-ary top tree bottom-up in linear time.

We may implement link, cut and expose by running the corresponding operation in $\mathcal{T}_2$. Each cut, link or expose operation will affect clusters on a constant number of root-paths in $\mathcal{T}_2$. There are thus only $\mathcal{O}(\log n / \log B)$ calls to split or merge of a cluster on a level divisible by $\lfloor \log_2 B \rfloor$. Thus, since each split or merge in $\mathcal{T}_B$ corresponds to a split or merge of a cluster in $\mathcal{T}_2$ whose level is divisible by $\lfloor \log_2 B \rfloor$, we have only $\mathcal{O}(\log n / \log B)$ calls to split and merge in $\mathcal{T}_B$.

However, since there are $\mathcal{O}(B)$ clusters whose parent pointers need to be updated after a merge, the total running time becomes $\mathcal{O}(B(\log n / \log B) + Q)$. □

8.3 Saving space with fat-bottomed top trees

In this section we present a general technique for reducing the space usage of a top tree based data structure to linear. For convenience, we will call any $b(n)$-ary top tree data structure that can be implemented using the top trees from Lemma 9 *well-behaved*. Loosely speaking, any well-behaved top tree data structure can be modified to use linear space.

The properties of the technique are captured in the following:

Lemma 10. *Suppose we have a well-behaved $b(n)$-ary top tree data structure, that uses $s(n)$ space per cluster, and spends $t(n)$ worst-case time per merge or split. Suppose further that there exists an algorithm that takes any subgraph of size q that forms a cluster, say, C, and calculates the complete information for C in $t_0(q,n)$ time, and suppose that the complete information for C has size at most $s_0(q,n)$. Finally, suppose that there exists a function q of n such that $s(n) < s_0(q(n),n) \in \mathcal{O}(q(n))$.*

Then, there exists a data structure maintaining the same information in top trees of height $h = O(\log n/\log b(n))$, such that the top trees use linear space in total, and have $\mathcal{O}(t(n)\cdot h(n)+t_0(q(n),n))$ update time for link, cut, and expose.

Proof. This follows directly from Lemma 9 by setting $Q = q(n)$ and $B = b(n)$. Then the top tree will have $\mathcal{O}(n/q(n))$ clusters of size at most $s_0(q(n),n) = \mathcal{O}(q(n))$ so the total size is linear. The time per update follows because the top tree uses $\mathcal{O}(h(n))$ merges or splits and $\mathcal{O}(1)$ create and destroy per link cut and expose. These take $t(n)$ and $t_0(q(n),n)$ time respectively. □

9 A faster CoverLevel structure

If we allow ourselves to use bit tricks, we can improve the CoverLevel data structure from Section 4. The main idea is, for some $0 < \epsilon < 1$, to use top trees of degree $b(n) = (\log n)^\epsilon \in \mathcal{O}(w/\log \ell_{\max})$. As noted in Lemma 9, such top trees have height $h(n) \in \mathcal{O}(\frac{\log n}{\epsilon \log\log n})$, and finding the sequence of merges and splits for a given link, cut or expose takes $\mathcal{O}(b(n)\cdot h(n)) \in \mathcal{O}(\frac{(\log n)^{1+\epsilon}}{\epsilon\log\log n}) \subseteq o((\log n)^{1+\epsilon})$ time.

The high-level algorithm makes at most a constant number of calls to link and cut for each insert or delete, so we are fine with the time for these operations. However, we can no longer use Expose to implement Cover, Uncover, CoverLevel and MinCoveredEdge, as that would take too long.

In this section, we will show how to overcome this limitation by working directly with the underlying tree.

The data The basic idea is to have each parent cluster store a *buffer* for each of its children, containing all the cover, cover^-, cover^+ and globalcover values. Since the degree is $\mathcal{O}(w/\log \ell_{\max})$, and each value uses at most $\mathcal{O}(\log \ell_{\max})$ bits, these fit into a constant number of words, and so we can use standard bit tricks[9] to operate on the buffers for all children of a node in parallel. We will show how to implement Cover, Uncover, CoverLevel, and MinCoveredEdge, such that each of them only touches $\mathcal{O}(h(n))$ nodes and the buffers stored in those nodes.

Let C be a cluster with children $A_1,\dots,A_k$. Since $k \le w/\log \ell_{\max}$, we can define the following vectors that each fit into a constant number of words.

$$\begin{aligned}
\text{packedcover}_C &:= (\text{cover}_{A_i})_{\{1\le i\le k\}}\\
\text{packedcover}^-_C &:= (\text{cover}^-_{A_i})_{\{1\le i\le k\}}\\
\text{packedcover}^+_C &:= (\text{cover}^+_{A_i})_{\{1\le i\le k\}}\\
\text{packedglobalcover}_C &:= (\text{globalcover}_{A_i})_{\{1\le i\le k\}}
\end{aligned}$$

The description of Split and Merge from Section 4 still apply, if we think of the "packed" values as a separate layer of degree 1 clusters between each pair of "real" clusters.

For concreteness, let C be a cluster with children $A_1,\dots,A_k$, and define operations

[9] See e.g. [8] or [1].

- CleanToBuffer(C). For each $1 \le i \le k$: If A_i is a path child of C and $\max\left\{\text{packedcover}_{C,i}, \text{packedcover}^-_{C,i}\right\} \le \text{cover}^-_C$, set:

$$\text{packedcover}^-_{C,i} = \text{cover}^-_C$$

  Then if $\text{packedcover}_{C,i} \le \max\left\{\text{cover}^-_C, \text{cover}^+_C\right\}$ set

$$\text{packedcover}_{C,i} = \text{cover}^+_C$$
$$\text{packedcover}^+_{C,i} = \text{cover}^+_C$$

  After updating all k children, set $\text{cover}^-_C = \text{cover}^+_C = -1$. Note that this can be done in parallel for all $1 \le i \le k$ in constant time using bit tricks.

- CleanToChild(C, i). If A_i is a path child of C and $\max\left\{\text{cover}_{A_i}, \text{cover}^-_{A_i}\right\} \le \text{packedcover}^-_{C,i}$, set

$$\text{cover}^-_{A_i} = \text{packedcover}^-_{C,i}$$

  Then if $\text{cover}_{A_i} \le \max\left\{\text{packedcover}^-_{C,i}, \text{packedcover}^+_{C,i}\right\}$ set

$$\text{cover}_{A_i} = \text{packedcover}^+_{C,i}$$
$$\text{cover}^+_{A_i} = \text{packedcover}^+_{C,i}$$

  Finally set $\text{packedcover}^-_{C,i} = \text{packedcover}^+_{C,i} = -1$. Again, note that this takes constant time.

- ComputeFromChild(C, i). Set

$$\text{packedcover}_{C,i} = \text{cover}_{A_i}$$
$$\text{packedcover}^-_{C,i} = -1$$
$$\text{packedcover}^+_{C,i} = -1$$
$$\text{packedglobalcover}_{C,i} = \text{globalcover}_{A_i}$$

- ComputeFromBuffer(C). For $1 \le i \le k$ define

$$\text{packedglobalcover}'_{C,i} = \begin{cases} \text{packedglobalcover}_{C,i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \text{or } \text{packedglobalcover}_{C,i} \le \text{packedcover}_{C,i} \\ \text{packedcover}_{C,i} & \text{otherwise} \end{cases}$$

$$\text{minglobaledge}'_{C,i} = \begin{cases} \text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \text{or } \text{globalcover}_{A_i} \le \text{cover}_{A_i} \\ \text{minpathedge}_{A_i} & \text{otherwise} \end{cases}$$

We can then compute the data for C from the buffer as follows:

$$\text{cover}_C = \begin{cases} \min\limits_{\substack{1\le i<k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i} & \text{if } |\partial C| = 2 \\ \ell_{\max} & \text{otherwise} \end{cases}$$

$$\text{minpathedge}_C = \begin{cases} \text{minpathedge}_{A_j} & \text{if } |\partial C| = 2 \\ \quad \text{where } j = \arg\min\limits_{\substack{1\le i<k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i} & \\ \textbf{nil} & \text{otherwise} \end{cases}$$

$$\text{globalcover}_C = \min_{1\le i<k} \text{packedglobalcover}'_{C,i}$$

$$\text{minglobaledge}_C = \text{minglobaledge}'_{C,j} \quad \text{where } j = \arg\min_{1\le i<k} \text{packedglobalcover}'_{C,i}$$

$$\text{cover}^-_C = -1$$

$$\text{cover}^+_C = -1$$

This can be computed in constant time, because $(\text{packedglobalcover}'_{C,i})_{\{1\le i\le k\}}$ fits into a constant number of words that can be computed in constant time using bit tricks, and thus each "min" or "arg min" is taken over values packed into a constant number of words.

Then Split(C) can be implemented by first calling CleanToBuffer(C), and then for each $1 \le i \le k$ calling CleanToChild(C, i). This ensures that all the lazy cover information is propagated down correctly. Similarly, Merge($C; A_1, \ldots, A_k$) can be implemented by first calling ComputeFromChild(C, i) for each $1 \le i \le k$, and then calling ComputeFromBuffer(C). Thus Split and Merge each take $\mathcal{O}(b(n))$ time.

Computing CoverLevel(v) **and** MinCoveredEdge(v) With the data described in the previous section, we can now answer the "global" queries as follows

$$\text{CoverLevel}(v) = \text{globalcover}_C$$

$$\text{MinCoveredEdge}(v) = \text{minglobaledge}_C \quad \text{where } C \text{ is the point cluster returned by root}(v)$$

Note that, for simplicity, we assume the top tree always has a single vertex exposed. This can easily be arranged by a constant number of calls to Expose after each link or cut, without affecting the asymptotic running time. Computing CoverLevel(v) or MinCoveredEdge(v) therefore takes $\mathcal{O}(h(n))$ worst case time.

Computing CoverLevel(v, w) **and** MinCoveredEdge(v, w) Since we can no longer use Expose to implement Cover and Uncover, we need a little more machinery.

What saves us is that all the information we need to find CoverLevel(v, w) is stored in the $\mathcal{O}(h(n))$ clusters that have v or w as internal vertices, and that once we have that, we can find a single child X of one of these clusters such that MinCoveredEdge(v, w) = minpathedge$_X$.

Before we get there, we have to deal with the complication of cover$^-$ and cover$^+$. Fortunately, all we need to do is make $\mathcal{O}(h(n))$ calls to CleanToBuffer and CleanToChild, starting from the root and going down towards v and w. Since each of these calls take constant time, we use only $\mathcal{O}(h(n))$ time on cleaning.

Now, the path $v \cdots w$ consists of $\mathcal{O}(h(n))$ edge-disjoint fragments, such that:

- Each fragment f is associated with, and contained in, a single cluster C_f whose parent has v or w as an internal vertex.
- For each fragment f, the endpoints are either in $\{v, w\}$ (and then C_f is a base cluster) or are boundary vertices of children of C_f.

We can find the fragments in $\mathcal{O}(h(n))$ time, and for each fragment f, we can in constant time find its cover level by examining packedcover$_{C_f}$.

Let $f_1, \ldots, f_k$ be the fragments of the path, and for $1 \le i \le k$ let v_i, w_i be the endpoints of the fragment closest to v, w respectively. Then[10]

$$\begin{aligned}
\text{CoverLevel}(v, w) &= \min_{1 \le i \le k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v, w) &= \text{MinCoveredEdge}(v_j, w_j) \\
&\quad \text{where } j = \arg\min_{1 \le i \le k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v_j, w_j) &= \text{minpathedge}_X \\
&\quad \text{where } X = \arg\min_{Y \text{ path child of } C_{f_j}} \text{cover}_Y
\end{aligned}$$

So computing CoverLevel(v, w) or MinCoveredEdge(v, w) takes $\mathcal{O}(h(n))$ worst case time.

Cover and Uncover We are now ready to handle Cover(v, w, i) and Uncover(v, w, i). First we make $\mathcal{O}(h(n))$ calls to CleanToBuffer and CleanToChild. Then let $f_1, \ldots, f_k$ be the fragments of the $v \cdots w$ path, and for $1 \le i \le k$ let v_i, w_i be the endpoints of the fragment closest to v, w respectively. Then for each $f \in f_1, \ldots, f_k$, and each path child A_j of C_f, Cover(v, w, i) needs to set

$$\begin{aligned}
\text{packedcover}_{C_f,j} &= \max\left\{\text{packedcover}_{C_f,j}, i\right\} \\
\text{packedcover}^+_{C_f,j} &= \max\left\{\text{packedcover}^+_{C_f,j}, i\right\}
\end{aligned}$$

Similarly, for each $f \in f_1, \ldots, f_k$, and for each path child A_j of C_f, if packedcover$_{C_f,j} \le i$, Uncover(v, w, i) needs to set

$$\begin{aligned}
\text{packedcover}_{C_f,j} &= -1 \\
\text{packedcover}^+_{C_f,j} &= -1 \\
\text{packedcover}^-_{C_f,j} &= \max\left\{\text{packedcover}^-_{C_f,j}, i\right\}
\end{aligned}$$

In each case, we can use bit tricks to make this take constant time per fragment. Finally, we need to update all the $\mathcal{O}(h(n))$ ancestors to the clusters we just changed. We can do this bottom-up using $\mathcal{O}(h(n))$ calls to ComputeFromChild and ComputeFromBuffer.

We conclude that Cover(v, w, i) and Uncover(v, w, i) each take worst case $\mathcal{O}(h(n))$ time.

[10] Recall that a *path child* of C is defined as a child that contains at least one edge of $\pi(C)$.

Analysis Choosing any $b(n) \in \mathcal{O}(w/\log \ell_{\max})$ we get height $h(n) \in \mathcal{O}(\frac{\log n}{\log b(n)})$, so Link and Cut take worst case $\mathcal{O}(\frac{b(n)\log n}{\log b(n)})$ time with this CoverLevel structure. The remaining operations, Connected, Cover, Uncover, CoverLevel and MinCoveredEdge all take $\mathcal{O}(\frac{\log n}{\log b(n)})$ worst case time. For the purpose of our main result, choosing $b(n) \in \Theta(\sqrt{\log n})$ is sufficient. Each cluster uses $\mathcal{O}(1)$ space, so the total space used is $\mathcal{O}(n)$.

10 Saving space

We now apply the space-saving trick from Lemma 10 to the FindSize structures from Section 5 and 7. Let D be the number of words used for each size vector in our FindSize structure. This is $\mathcal{O}(\log n)$ for the purely combinatorial version, and $\mathcal{O}(\log \log n)$ in the version using approximate counting. As shown previously these use $s(n) = \mathcal{O}(D)$ space per cluster and $t(n) = \mathcal{O}(\log n \cdot D)$ worst case time per merge and split.

Lemma 11. *The complete information for a cluster of size q in the FindSize structure, including information that would be shared with its children, has total size $s_0(q, n) = \mathcal{O}(q + \ell_{\max} \cdot D)$.*

Proof. The complete information for a cluster C with $|C| = q$ consists of

- $c(e)$ for all $e \in C$.
- cover_C, cover_C^-, cover_C^+, globalcover_C, size_C.
- $\text{partsize}_{C,v,i}$ and $\text{diagsize}_{C,v,i}$ for $v \in \partial C$ and $-1 \leq i \leq \ell_{\max}$.

The total size for all of these is $s_0(q, n) = \mathcal{O}(q + \ell_{\max} \cdot D)$ □

Note that when keeping n fixed, this is clearly $\mathcal{O}(q)$. In particular, we can choose $q(n) \in \Theta(\ell_{\max} \cdot D)$ such that $s(n) < s_0(q(n), n) \in \mathcal{O}(q(n))$.

Lemma 12. *The complete information for a cluster of size q in the FindSize structure, including information that would be shared with its children, can be computed directly in time $t_0(q, n) = \mathcal{O}(q \log q + \ell_{\max} \cdot D)$.*

Proof. Let C be the cluster of size $|C| = q$. For each $v \in \partial C$, we can in $\mathcal{O}(q)$ time find and partition the cluster path into the at most $\ell_{\max}$ parts such that in part i, each vertex m on the cluster path have $\text{CoverLevel}(v, m) = i$. For each part i, run the following algorithm:

```
Vector x ← 0⃗
Initialize empty max-queue Q
j ← ℓ_max
for w ← each vertex in the fragment that is on π(C) do
    Mark w as visited
    x_j ← x_j + 1
    for e ← each edge incident to w that is not on π(C) do
        if c(e) ≥ 0 then
            Add e to Q with key c(e)
while Q is not empty do
```

```
    e ← EXTRACT-MAX(Q)
    while c(e) < j do
        x_{j-1} = x_j
        j ← j − 1
    w ← the unvisited vertex at the end of e
    Mark w as visited
    x_j ← x_j + 1
    for e ← each edge incident to w that has an unvisited end do
        if c(e) ≥ 0 then
            Add e to Q with key c(e)
partsize_{C,v,i} ← x
diagsize_{C,v,i} ← M(i) · x
```

If the ith part has size q_i than it can be processed this way in $\mathcal{O}(q_i \log q_i + D)$ time. Summing over all $\mathcal{O}(\ell_{\max})$ parts gives the desired result. □

Analysis Applying Lemma 10 with the $s(n)$, $t(n)$, $s_0(q,n)$, $t_0(q,n)$ and $q(n)$ derived in this section immediately gives a FindSize structure with $\mathcal{O}(\log n \cdot D \cdot \log \ell_{\max})$ worst case time per operation and using $\mathcal{O}(n)$ space. A completely analogous argument shows that we can convert the bitpacking-free version of the FindFirstLabel structure from $\mathcal{O}(\log n \cdot \ell_{\max} \cdot \log \ell_{\max})$ time and $\mathcal{O}(m + n \cdot \ell_{\max})$ space to one using linear space. (If bitpacking is allowed the structure already used linear space). In either case is the same time per operation as the original versions, so using the modified version here does not affect the overall running time, but reduces the total space of each bridge-finding structure to $\mathcal{O}(m+n)$.

Note that we can explicitly store lists with all the least-covered edges for these large base clusters, so this does not change the time to report the first k least-covered edges.

References

[1] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136(1):25–51, 1997.

[2] Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, October 2005.

[3] Therese C. Biedl, Prosenjit Bose, Erik D. Demaine, and Anna Lubiw. Efficient algorithms for petersen's matching theorem. *Journal of Algorithms*, 38(1):110 – 134, 2001.

[4] Krzysztof Diks and Piotr Stanczyk. *Perfect Matching for Biconnected Cubic Graphs in $O(n \log^2 n)$ Time*, pages 321–333. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Improved sparsification. Technical report, 1993.

[6] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[7] Greg N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[8] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[9] Harold N. Gabow, Haim Kaplan, and Robert Endre Tarjan. Unique maximum matching algorithms. *J. Algorithms*, 40(2):159–183, 2001. Announced at STOC '99.

[10] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[11] Monika R. Henzinger and Valerie King. *Maintaining minimum spanning trees in dynamic graphs*, pages 594–604. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[12] Monika Rauch Henzinger and Valerie King. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation, 1997.

[13] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 79–89, New York, NY, USA, 1998. ACM.

[14] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, July 2001.

[15] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in o(log n(log log n)2) amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 510–520, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

[16] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1131–1142, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[17] Donald E. Knuth. Two notes on notation. *The American Mathematical Monthly*, 99(5):403–422, 1992.

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

[19] Anton Kotzig. *On the theory of finite graphs with a linear factor II.* 1959.

[20] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10, 1927.

[21] Mihai Patrascu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.

[22] Julius Petersen. Die Theorie der regulären graphs. *Acta Math.*, 15:193–220, 1891.

[23] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 343–350, New York, NY, USA, 2000. ACM.

[24] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*, pages 738–741. 2016.

A Details of the high level algorithm

Lemma 5 (Essentially the high level algorithm from [14])**.** *There exists a deterministic reduction for dynamic graphs with n nodes, that, when starting with an empty graph, supports any sequence of m Insert or Delete operations using:*

- $\mathcal{O}(m)$ *calls to Link, Cut, Uncover, and CoverLevel.*
- $\mathcal{O}(m \log n)$ *calls to Connected, Cover, AddLabel, RemoveLabel, FindFirstLabel, and FindSize.*

And that can answer FindBridge queries using a constant number of calls to Connected, CoverLevel, and MinCoveredEdge, and size queries using a single call to FindSize.

Proof. The only part of the high level algorithm from [14] that does not directly and trivially translate into a call of the required dynamic tree operations (see pseudocode below) is in the Swap method where given a tree edge $e = (v, w)$ we need to find a nontree edge e' covering e with $\ell(e') = i = \text{CoverLevel}(e)$. We can find this e' by using FindFirstLabel and increasing the level of each non-tree edge we examine that does not cover e. For at least one side of (v, w), all non-tree edges at level i incident to that side will either cover e or can safely have their level increased without violating the size invariant. So we can simply search the side where the level i component is smallest until we find the required edge (which must exist since e was covered on level i). The amortized cost of all operations remain unchanged with this implementation. Counting the number of operations (see Table 2) gives the desired bound. □

```
1: function 2-edge-connected(v, w)
2:     return T.Connected(v, w) ∧ T.CoverLevel(v, w)≥ 0
3: function FindBridge(v)
4:     if T.CoverLevel(v)= −1 then
5:         return T.MinCoveredEdge(v)
6:     else
7:         return nil
8: function FindBridge(v, w)
9:     if T.CoverLevel(v, w)= −1 then
10:        return T.MinCoveredEdge(v, w)
11:    else
12:        return nil
13: function Size(v)
14:    return T.FindSize(v,v,−1)
15: function 2-Size(v)
16:    return T.FindSize(v,v,0)
```

#	Operation	#Calls during				
		Insert+Delete	FindBridge(v)	FindBridge(v, w)	Size(v)	2-Size(v)
1	Link(v, w, e)	1	0	0	0	0
2	Cut(e)	1	0	0	0	0
3	Connected(v, w)	$\log n$	0	1	0	0
4	Cover(v, w, i)	$\log n$	0	0	0	0
5	Uncover(v, w, i)	1	0	0	0	0
6	CoverLevel(v)	0	1	0	0	0
7	CoverLevel(v, w)	1	0	1	0	0
8	MinCoveredEdge(v)	0	1	0	0	0
9	MinCoveredEdge(v, w)	0	0	1	0	0
10	AddLabel(v, l, i)	$\log n$	0	0	0	0
11	RemoveLabel(l)	$\log n$	0	0	0	0
12	FindFirstLabel(v, w, i)	$\log n$	0	0	0	0
13	FindSize(v, w, i)	$\log n$	0	0	0	1
	FindSize($v, v, -1$)	0	0	0	1	0

Table 2: Overview of how many times each tree operation is called for each graph operation, ignoring constant factors. The "Insert+Delete" column is amortized over any sequence starting with an empty set of edges. The remaining columns are worst case.

```
17: function INSERT(v, w, e)
18:     if ¬T.CONNECTED(v, w) then
19:         T.LINK(v, w, e)
20:         ℓ(e) ← ℓ_max
21:     else
22:         T.ADDLABEL(v, e.label1, 0)
23:         T.ADDLABEL(w, e.label2, 0)
24:         ℓ(e) ← 0
25:         T.COVER(v, w, 0)
26: function DELETE(e)
27:     (v, w) ← e
28:     α ← ℓ(e)
29:     if α = ℓ_max then
30:         α ← T.COVERLEVEL(v, w)
31:         if α = −1 then
32:             T.CUT(e)
33:             return
34:         SWAP(e)
35:     T.REMOVELABEL(e.label1)
36:     T.REMOVELABEL(e.label2)
37:     T.UNCOVER(v, w, α)
38:     for i ← α, ..., 0 do
39:         RECOVER(w,v,i)
40: function SWAP(e)
41:     (v, w) ← e
```

```
42:     α ← T.CoverLevel(v, w)
43:     T.Cut(e)
44:     e′ ←FindReplacement(v,w,α)
45:     (x, y) ← e′
46:     T.RemoveLabel(e′.label1)
47:     T.RemoveLabel(e′.label2)
48:     T.Link(x, y, e′)
49:     ℓ(e′) ← ℓ_max
50:     T.AddLabel(v,e.label1, α)
51:     T.AddLabel(w,e.label2, α)
52:     ℓ(e) ← α
53:     T.Cover(v,w,α)
54: function FindReplacement(v,w,i)
55:     s_v ← T.FindSize(v, v, i)
56:     s_w ← T.FindSize(w, w, i)
57:     if s_v ≤ s_w then
58:         return RecoverPhase(v, v, i, s_v)
59:     else
60:         return RecoverPhase(w, w, i, s_w)
61: function Recover(v,w,i)
62:     s ← ⌊T.FindSize(v, w, i)/2⌋
63:     RecoverPhase(v,w,i,s)
64:     RecoverPhase(w,v,i,s)
65: function RecoverPhase(v, w, i, s)
66:     l ← T.FindFirstLabel(v, w, i)
67:     while l ≠ nil do
68:         e ← l.edge
69:         (q, r) ← e
70:         if ¬T.Connected(q, r) then
71:             return e
72:         if T.FindSize(q, r, i + 1) ≤ s then
73:             T.RemoveLabel(e.label1)
74:             T.RemoveLabel(e.label2)
75:             T.AddLabel(q, e.label1, i + 1)
76:             T.AddLabel(r, e.label2, i + 1)
77:             ℓ(e) = i + 1
78:             T.Cover(q,r,i + 1)
79:         else
80:             T.Cover(q,r,i)
81:             return nil
82:         l ← T.FindFirstLabel(v, w, i)
83:     return nil
```

B Pseudocode for the CoverLevel structure

```
1: function CL.Cover(v,w,i)
2:     C ← TopTree.Expose(v, w)
3:     cover_C ← max{cover_C, i}
4:     cover⁺_C ← max{cover⁺_C, i}
5: function CL.Uncover(v,w,i)
6:     C ← TopTree.Expose(v, w)
7:     cover_C ← −1
8:     cover⁺_C ← −1
9:     cover⁻_C ← max{cover⁻_C, i}
10: function CL.CoverLevel(v)
11:     C ← TopTree.Expose(v)
12:     return globalcover_C
13: function CL.CoverLevel(v, w)
14:     C ← TopTree.Expose(v, w)
15:     return cover_C
16: function CL.MinCoveredEdge(v)
17:     C ← TopTree.Expose(v)
18:     return minglobaledge_C
19: function CL.MinCoveredEdge(v, w)
20:     C ← TopTree.Expose(v, w)
21:     return minpathedge_C
22: function CL.Split(C)
23:     for each path child D of C do
24:         if max{cover_D, cover⁻_D} ≤ cover⁻_C then
25:             cover⁻_D ← cover⁻_C
26:         if cover_D ≤ max{cover⁻_D, cover⁺_D} then
27:             cover_D ← cover⁺_C
28:             cover⁺_D ← cover⁺_C
29: function CL.Merge(C; A_1, ..., A_k)
30:     cover_C ← ℓ_max
31:     minpathedge_C ← nil
32:     globalcover_C ← ℓ_max
33:     minglobaledge_C ← nil
34:     for i ← 1, ..., k do
35:         if ∂A_i ⊆ π(C) then
36:             if cover_{A_i} < cover_C then
37:                 cover_C ← cover_{A_i}
38:                 minpathedge_C ← minpathedge_{A_i}
39:         else
40:             if cover_{A_i} < globalcover_C then
41:                 globalcover_C ← cover_{A_i}
```

```
42:                 minglobaledge_C ← minpathedge_{A_i}
43:         if globalcover_{A_i} < globalcover_C then
44:             globalcover_C ← globalcover_{A_i}
45:             minglobaledge_C ← minglobaledge_{A_i}
46:     cover_C^- ← −1
47:     cover_C^+ ← −1
48: function CL.Create(C; edge e)
49:     cover_C ← −1
50:     globalcover_C ← −1
51:     if C is a point cluster then
52:         minpathedge_C ← nil
53:         minglobaledge_C ← e
54:     else
55:         minpathedge_C ← e
56:         minglobaledge_C ← nil
57:     cover_C^- ← −1
58:     cover_C^+ ← −1
```

C Pseudocode for the FindSize structure

In the following, we use the notation

$$[\text{key} : \text{partsize}, \text{diagsize}]$$

to denote the root of a new tree consisting of a single node with the given values. And for a given tree root and given x, y

$$(\text{tree}_{\{x \le i \le y\}})$$

is the root of the subtree consisting of all nodes whose keys are in the given range. Similarly, for any given i, let

$$(\text{tree}_i)$$

denote the node in the tree having the given key.

```
1: function FS.FindSize(v, w, i)
2:     C ← TopTree.Expose(v, w)
3:     return size_{C,i}
4: function FS.Merge(C; A, B)
5:     {c} ← ∂A ∩ ∂B
6:     if c ∈ π(C) then                                  ▷ Merge along path
7:         if |∂C| <= 1 then
8:             a ← c, b ← c
9:         else
10:            {a, b} ← ∂C with a ∈ ∂A and b ∈ ∂B.
```

$\text{size}_C \leftarrow \text{size}_A + \text{size}_B$
for $(x, X) \leftarrow (a, A), (b, B)$ **do**
if $x = c$ **then**
$\text{tree}'_{X,x} \leftarrow \text{tree}_{X,x}$, $\text{undo}'_{X,x} \leftarrow$ **nil**
else
for $v \leftarrow x, c$ **do**
$\ell \leftarrow \max\{\text{cover}^-_X, \text{cover}^+_X\}$
$s \leftarrow (\text{tree}_{X,v}).\,\text{partsizesum}$
$d \leftarrow M(\text{cover}^+_X) * s$
$\text{tree}'_{X,v} \leftarrow \text{tree}_{X,v,\{i>\ell\}}$, $\text{undo}'_{X,v} \leftarrow \text{tree}_{X,v,\{i\le\ell\}}$
$\text{tree}'_{X,v} \leftarrow \text{tree}'_{X,v} + [\text{cover}^+_X : s, d]$
for $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ **do**
$s \leftarrow (\text{tree}'_{Y,c,\{\text{cover}_X \le i \le \ell_{\max}\}}).\,\text{partsizesum}$
$p \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{partsize} + s$
$d \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{diagsize} + M(\text{cover}_X) * s$
if $x = c$ **then**
$\text{tree}''_{X,x} \leftarrow [\ell_{\max} : \text{size}_X, \text{size}_X]$, $\text{undo}''_{X,x} \leftarrow$ **nil**
else
$\text{tree}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i>\text{cover}_X\}}$, $\text{undo}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i\le\text{cover}_X\}}$
if $y = c$ **then**
$\text{tree}'''_{Y,c} \leftarrow$ **nil**, $\text{undo}'''_{Y,c} \leftarrow [\ell_{\max} : \text{size}_Y, \text{size}_Y]$
else
$\text{tree}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i<\text{cover}_X\}}$, $\text{undo}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i\ge\text{cover}_X\}}$
$\text{tree}_{C,x} \leftarrow \text{tree}''_{X,x} + [\text{cover}_X : p, d] + \text{tree}'''_{Y,c}$
else ▷ Merge off path
$\{a\} \leftarrow \partial C \setminus \{c\}$
if $a \notin \partial A$ **then**
Swap A and B
$\ell \leftarrow \max\{\text{cover}^-_A, \text{cover}^+_A\}$
$d \leftarrow (\text{tree}_{A,a,\{\ell<i\le\ell_{\max}\}}).\,\text{diagsizesum}$
$p \leftarrow (\text{tree}_{A,a,\{-1\le i\le\ell\}}).\,\text{partsizesum}$
$\text{size}_C \leftarrow d + M(\text{cover}^+_A) * p + M(\text{cover}_A) * \text{size}_B$
$\text{tree}_{C,a} \leftarrow [\ell_{\max} : \text{size}_C, \text{size}_C]$
function FS.SPLIT(C)
$A, B \leftarrow$ the children of C
$\{c\} \leftarrow \partial A \cap \partial B$
if $c \in \pi(C)$ **then** ▷ Split along path
if $|\partial C| <= 1$ **then**
$a \leftarrow c$, $b \leftarrow c$
else
$\{a, b\} \leftarrow \partial C$ with $a \in \partial A$ and $b \in \partial B$.
for $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ **do**
$\text{tree}''_{X,x} \leftarrow \text{tree}_{C,x,\{i>\text{cover}_X\}}$, $\text{tree}'''_{Y,c} \leftarrow \text{tree}_{C,x,\{i<\text{cover}_X\}}$

54: **if** $y \neq c$ **then**
55: $\text{tree}'_{Y,c} \leftarrow \text{tree}'''_{Y,c} + \text{undo}'''_{Y,c}$
56: **if** $x \neq c$ **then**
57: $\text{tree}'_{X,x} \leftarrow \text{tree}''_{X,x} + \text{undo}''_{X,x}$
58: **for** $(x, X) \leftarrow (a, A), (b, B)$ **do**
59: **if** $x \neq c$ **then**
60: **for** $v \leftarrow x, c$ **do**
61: $\text{tree}_{X,v} \leftarrow \text{tree}'_{X,v,\{i>\text{cover}^+_X\}} + \text{undo}'_{X,v}$
62: **function** FS.CREATE(C; edge e)
63: $\text{size}_C \leftarrow \vec{0}$
64: **for** $v \in \partial C$ **do**
65: $\text{tree}_{C,v} \leftarrow [\ell_{\max} : \vec{0}, \vec{0}]$
66: **function** FS.CREATE(C; vertex label l)
67: $\text{size}_C \leftarrow (1)_{\{0 \leq i < \ell_{\max}\}}$
68: **for** $v \in \partial C$ **do**
69: $\text{tree}_{C,v} = [\ell_{\max} : \text{size}_C, \text{size}_C]$

One-Way Trail Orientations

Anders Aamand, Niklas Hjuler *, Jacob Holm†, and
Eva Rotenberg

University of Copenhagen

Abstract

Given a graph, does there exist an orientation of the edges such that the resulting directed graph is strongly connected? Robbins' theorem [Robbins, Am. Math. Monthly, 1939] states that such an orientation exists if and only if the graph is 2-edge connected. A natural extension of this problem is the following: Suppose that the edges of the graph is partitioned into trails. Can we orient the trails such that the resulting directed graph is strongly connected?

We show that 2-edge connectivity is again a sufficient condition and we provide a linear time algorithm for finding such an orientation, which is both optimal and the first polynomial time algorithm for deciding this problem.

The generalised Robbins' theorem [Boesch, Am. Math. Monthly, 1980] for mixed multigraphs states that the undirected edges of a mixed multigraph can be oriented making the resulting directed graph strongly connected exactly when the mixed graph is connected and the underlying graph is bridgeless. We show that as long as all cuts have at least 2 undirected edges or directed edges both ways, then there exists an orientation making the resulting directed graph strongly connected. This provides the first polynomial time algorithm for this problem and a very simple polynomial time algorithm to the previous problem.

*This work is supported by the Innovation Fund Denmark through the DABAI project.

†This research is supported by Mikkel Thorup's Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research under the Sapere Aude research career programme.

1 Introduction and motivation

Suppose that the mayor of a small town decides to make all the streets one-way in such a way that it is possible to get from any place to any other place without violating the orientations of the streets[1]. If initially all the streets are two-way then Robbins' theorem [9] asserts that this can be done exactly when the corresponding graph is 2-edge connected. If, on the other hand some of the streets were already one-way in the beginning then the generalised Robbins' theorem [1] states that it can be done exactly when the corresponding graph is strongly connected and the underlying graph is 2-edge connected.

However, the proofs of both of these results assume that every street of the city corresponds to exactly one edge in the graph. This assumption hardly holds in any city in the world and therefore a much more natural assumption is that every street corresponds to a trail in the graph and that the edges of each trail must be oriented consistently[2].

In this paper we prove that Robbins' Theorem continues to hold even when the set of edges is partitioned into trails. In other words a necessary and sufficient condition for an orientation to exist is that the graph is 2-edge connected. We also provide a linear time algorithm for finding such an orientation.

Finally we will consider the generalised Robbins' theorem in this new setting i.e. we allow some edges to be oriented initially and suppose that the remaining edges are partitioned into trails. We will show that if any cut (V_1, V_2) in the graph has either at least 2 undirected edges going between V_1 to V_2 or a directed edge in each direction then it is possible to orient the trails making the resulting graph strongly connected. Although this condition is not necessary it does give a simple algorithm for deciding the problem. Indeed, the only cuts containing an undirected edge which we allow are the ones where this edge (and hence its trail) is *forced* in one direction. Hence for deciding the problem we can start by orienting all the forced trails until there are no more forced trails. Then the trails can be oriented making the graph strongly connected exactly if the resulting graph satisfies our condition.

Note that when some edges are initially oriented the answer to the problem depends on the trail decomposition which is not the case for the other results. That the condition from the generalised Robbins' theorem is not sufficient can be seen from figure 1.

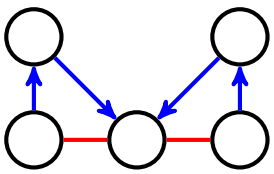


Figure 1: The graph is strongly connected and the underlying graph is 2-edge connected, but no matter the orientation of the red trail, the graph will lose its strong connectivity

Earlier methods Several methods have already been applied for solving orientation problems in graphs where the goal is to make the resulting graph strongly connected.

[1]The motivation for doing so is that the streets of the town are very narrow and thus it is a great hassle when two cars unexpectedly meet.

[2]This version of the problem was given to us through personal communication with Professor Robert E. Tarjan

One approach used by Robbins [9] is to use that a 2-edge connected graph has an *ear-decomposition*. An ear decomposition of a graph is a partition of the set of edges into a cycle C and paths $P_1, \ldots, P_t$ such that P_i has its two endpoints but none of its internal vertices on $C \cup \left(\bigcup_{j=1}^{i-1} P_j\right)$. Assuming the existence of an ear decomposition of 2-edge connected graphs it is easy to prove Robbins' theorem. Indeed, it is easy to see by induction that any consistent orientations of the paths and the cycle give a strongly connected graph.

A second approach introduced by Tarjan [3] gives another simple proof of Robbins' theorem. One can make a DFS tree in the graph rooted at a vertex v and orient all edges in the DFS tree away from v. The remaining edges are oriented towards v and if the graph is 2-edge connected it is easily verified that this gives a strong orientation.

A similar approach was used by Chung et al. [2] in the context of the generalized Robbins theorem for mixed multigraphs.

The above methods not only prove Robbins' theorem, they also provide linear time algorithms for finding strong orientations of undirected or mixed multigraphs.

However, none of the above methods have proven fruitful in our case. In case of the ear decomposition one needs a such which is somehow compatible with the partitioning into trails and this seems hard to guarantee. The original proof by Roberts is essentially similar to using the ear decomposition. Similar problems appear when trying a DFS-approach. Neither does the proof by Boesch [1] of Robbins' theorem for mixed multigraphs generalise to prove our result. Most importantly the corresponding theorem is no longer true for trail orientations as is shown by the example above.

Since the classical linear time algorithms rely on ear-decompositions and DFS searches, and since these approaches do not immediately work for trail partitions, our linear time algorithm will be a completely new approach to solving orientation problems.

2 Preliminaries

Let us briefly review the concepts from graph theory that we will need. Recall that a *walk* in a graph is an alternating sequence of vertices and edges $v_0, e_1, v_1, e_2, \ldots, v_k$, such that for $1 \leq i \leq k$ the edge e_i has v_{i-1} and v_i as its two endpoints. In a directed or mixed graph the ordering of the endpoints of each edge in the sequence must be consistent with the direction of the edge in case it is oriented. A *trail* is a walk without repeated edges. A *path* is a trail without repeated vertices (except possibly $v_0 = v_k$). Finally a *cycle* is a path for which $v_0 = v_k$

Next, recall that a mixed multigraph $G = (V, E)$ is called *strongly connected* if for any vertices $u, v \in V$ there exists a walk from u to v. In case that the graph contains no directed edges this is equivalent to saying that it consists of exactly one connected component.

We also recall the definition of k-edge connectivity. A graph $G = (V, E)$ is said to be *k-edge connected* if and only if $G' = (V, E - X)$ is connected for all $X \subseteq E$ where $|X| < k$. A trivially equivalent condition is that any edge-cut (V_1, V_2) in the graph has at least k edges going between V_1 and V_2.

Finally, if $G = (V, E)$ is a mixed multigraph and $A \subseteq V$ we define G/A to be that graph obtained by contracting A to a single vertex and $G[A]$ to be the subgraph of G induced by A. The following simple observation will be used repeatedly in this paper.

Observation 1. *If $G = (V, E)$ is k-edge connected and $A \subseteq V$ then G/A is k-edge connected. Also if G is a strongly connected mixed multigraph then G/A is too.*

The structure of this paper is as follows. In section 3 we prove our generalisation of Robbins' theorem for undirected graphs partitioned into trails. In section 4 we study what happens in the case of mixed graphs. Finally in section 5 we provide our linear time algorithm for trail orientation in an undirected graph.

3 Robbins Theorem Revisited

We are now ready to state our generalisation of Robbins' theorem.

Theorem 2. *Let $G = (V, E)$ be a multigraph with E partitioned into trails. An orientation of each trail such that the resulting directed graph is strongly connected exists if and only if G is 2-edge connected.*

Proof. If G is not 2-edge connected, such an orientation obviously doesn't exist so we need to prove the converse. Suppose therefore that G is 2-edge connected.

Our proof is by induction on the number of edges in G. If there are no edges, the graph is a single vertex, and the statement is obviously true. Assume now the statement holds for all graphs with strictly fewer edges than G. Pick an arbitrary edge e that is at the end of its corresponding trail.

If $G - e$ is 2-edge connected, then by the induction hypothesis there is a strong orientation of $G - e$ that respects the trails of G. Such an orientation clearly extends to the required orientation of G.

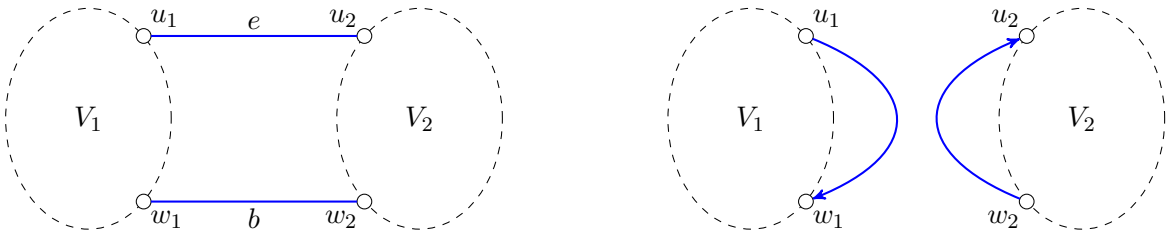


Figure 2: A two edge cut and the two graphs G_1 and G_2.

If $G - e$ is not 2-edge connected, there exists a bridge b in $G - e$ (see figure 2). Let V_1, V_2 be the two connected components of $G - \{e, b\}$, and let $e = (u_1, u_2)$ and $b = (w_1, w_2)$ such that for $i \in \{1, 2\}$, $u_i, w_i \in V_i$ (note that we don't necessarily have that u_i and w_i are distinct for $i \in \{1, 2\}$). Now for $i \in \{1, 2\}$ construct the graph $G_i = G[V_i] \cup \{(u_i, w_i)\}$, and define the trails in G_i to be the trails of G that are completely contained in G_i, together with a single trail combined from the (possibly empty) partial trail of e contained in G_i and ending at e_i, followed by the edge (u_i, w_i), followed by the (possibly empty) partial trail of b contained in G_i starting at b_i. Both G_1 and G_2 are 2-edge connected since they can each be obtained as a contraction of G. Furthermore, they each have strictly fewer edges than G, so inductively each has a strong orientation that respects the given trails. Further, we can assume that the orientations are such that the new edges are oriented as (u_1, w_1) and (w_2, u_2) by flipping the orientation of all edges in either graph if necessary. We claim that this orientation, together with e oriented as (u_1, u_2) and b oriented as (w_2, w_1), is the required orientation of G. To see this first note that (by construction) this orientation respects the trails. Secondly suppose $v_1 \in V_1$ and $v_2 \in V_2$ are arbitrary. Since G_1 is strongly connected $G[V_1]$

contains a directed path from v_1 to u_1. Similarly, $G[V_2]$ contains a directed path from u_2 to v_2. Thus G contains a directed path from v_1 to v_2. A similar argument gives a directed path from v_2 to v_1 and since v_1 and v_2 were arbitrary this proves that G is strongly connected and our induction is complete.

□

The construction in the proof can be interpreted as a naive algorithm for finding the required orientation when it exists.

Corollary 3. *The one-way trail orientation problem on a graph with n vertices and m edges can be solved in $\mathcal{O}(n + m \cdot f(m, n))$ time, where $f(m, n)$ is the time per operation for fully dynamic bridge finding (a.k.a. 2-edge connectivity).*

At the time of this writing[3], this is $\mathcal{O}(n + m(\log n \log\log n)^2)$. In Section 5 we will show a less naive algorithm that runs in linear time.

4 Extension to Mixed graphs

Now we will extend our result to the case of mixed graphs. We are going to prove the following.

Theorem 4. *Let $G = (V, E)$ be a strongly connected mixed multigraph. Then $G - e$ is strongly connected for all undirected $e \in E$ if and only if for any partition $\mathcal{P}$ of the undirected edges of G into trails, and any $T \in \mathcal{P}$, any orientation of T can be extended to a strong trail orientation of $(G, \mathcal{P})$.*

Suppose $G = (V, E)$ is as in the theorem. We will say that $e \in E$ is *forced* if it is undirected and satisfies that $G - e$ is not strongly connected[4]. Note that this is a proper extension of Theorem 2 since if G is undirected and 2-edge connected then no $e \in E$ is forced.

For proving the result we'll need the following lemma.

Lemma 5. *Let G be a directed graph, and let (A, B) be a cut with exactly one edge crossing from A to B and at least one edge crossing from B to A. Then G is strongly connected if and only if G/A and G/B are.*

Proof. Strong connectivity is preserved by contractions, so if G is strongly connected then G/A and G/B both are. For the other direction, let (a_1, b_1) be the edge going from A to B, and let (b_2, a_2) be any edge from B to A. Since G/A is strongly connected and (a_1, b_1) is the only edge from A to B, G/A contains a path from b_1 to b_2 that stays in B. Since this holds for any edge going from B to A, and since G/B is strongly connected, A is strongly connected in G. By a symmetric argument, B is also strongly connected in G and since the cut has edges in both directions, G must be strongly connected. □

Now we provide the proof of Theorem 4.

[3] Separate paper submitted to SODA'18 by Holm, Rotenberg and Thorup.

[4] This terminology is natural since it is equivalent to saying that there exists a cut (V_1, V_2) in G such that e is the only undirected edge in this cut and such that all the directed edges go from V_1 to V_2. If one wants an orientation of the trails making the graph strongly connected we are clearly forced to orient e from V_2 to V_1.

Proof of theorem 4. If $G - e$ is not strongly connected, the trail T containing e can at most be directed one way since e is forced, so there is an orientation of T which not extend to a strong trail orientation of $(G, \mathcal{P})$. To prove the converse suppose $G - e$ is strongly connected for all undirected $e \in E$.

The proof is by induction on $|\mathcal{P}|$. If $|\mathcal{P}| \leq 1$ the result is trivial. So suppose $|\mathcal{P}| > 1$ and that the theorem holds for all $(G', \mathcal{P}')$ with $|\mathcal{P}'| < |\mathcal{P}|$.

Consider a trail $T \in \mathcal{P}$. Suppose there is no cut (A, B) that T crosses exactly once, which has exactly one other undirected edge crossing it, and has every directed edge crossing it going from A to B. Then regardless of the orientation of T, the resulting graph G' has no undirected edge e such that $G - e$ is not strongly connected. Thus, by induction $(G', \mathcal{P} \setminus \{T\})$ has a strong trail orientation, which is also a strong trail orientation of $(G, \mathcal{P})$, as desired.

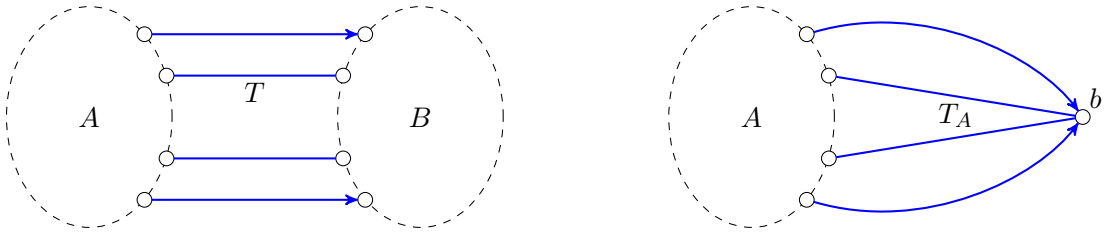


Figure 3: A cut with two undirected edges and all directed edges going from A to B followed by a contraction of B.

Now suppose there is such a cut (A, B) (see figure 3). Construct a graph G/B by contracting every vertex in B into a single new vertex b. Let $\mathcal{P}_A$ consist of all trails in $\mathcal{P}$ that are completely contained in A, together with a single trail T_A combined from the (possibly empty) fragments of the two trails that crossed the cut, joined at b. Since any cut in G/B corresponds to a cut in G, G/B is strongly connected and remains so after deletion of any single undirected edge. By induction any orientation of T_A in G/B extends to a strong orientation of $(G/B, \mathcal{P}_A)$. Let G/A, a, $\mathcal{P}_\mathcal{B}$ and T_B be defined symmetrically, then by the same argument any orientation of T_B in G/A extends to a strong orientation of $(G/A, \mathcal{P}_B)$. Now for any orientation of T, we can choose orientations of T_A and T_B that are compatible. The result follows by Lemma 5. □

Notice that the partitioning of edges into trails does not matter in the case when no edge is forced. Since any undirected graph has no forced edges if it is 2-edge connected, the theorem implies that the most naive algorithm: "directing trails that are forced and if none are forced direct an arbitrary trail" works for undirected graphs. In general for mixed graphs algorithm 1 below can clearly be implemented in polynomial time and does solve the trail orientation problem for mixed graphs.

Theorem 4 gives a sufficient condition for when a strong orientation exists and we deal with the other cases by dealing with the forced edges first. However, the generalised Robbins' Theorem provides a simple equivalent condition, which we lack. Finding such an equivalent condition when you have trail decomposition is an essential open problem for strong graph orientations. Due to figure 1 in this setting one has to take into account the structure of the trail partition.

Algorithm 1: Algorithm for mixed graphs.

Input: A mixed multigraph G and a partition $\mathcal{P}$ of the undirected edges of G into trails.
Output: True if $(G, \mathcal{P})$ has a strong trail orientation, otherwise false. G is modified in place, either to have such a strong trail orientation, or to a forced graph that is not strongly connected.

```
if G has a bridge or is not strongly connected then
    return false
end
while |P| > 0 do
    if for some undirected edge e, G − e is not strongly connected then
        Let T ∈ P be the trail containing e.
        if some orientation of T leaves G strongly connected then
            Apply such an orientation of T to G
        else
            return false
        end
    else
        Let T ∈ P be arbitrary.
        Update G by orienting T in an arbitrary direction.
    end
    Remove T from P.
end
return true
```

5 Linear time algorithm

In this section we provide our linear time algorithm for solving the trail orientation problem in undirected graphs. For this, we make two crucial observations. First, we show that there is an easy linear time reduction from general graphs or multigraphs to cubic multigraphs. Second, we show that in a cubic multigraph with n vertices, we can in linear time find and delete a set of edges that are at the end of their trails, such that the resulting graph has $\Omega(n)$ 3-edge connected components. We further show that we can compute the required orientation recursively from an orientation of each 3-edge connected component together with the cactus of 3-edge connected components. Since the average size of these components is constant, we can compute the orientations of most of them in linear time. The rest contains at most a constant fraction of the vertices, and so a simple geometric sum argment tells us that the total time is also linear.

We start out by making the following reduction.

Lemma 6. *The one-way trail problem on a 2-edge connected graph or multigraph with n vertices and m edges, reduces in $\mathcal{O}(m+n)$ time to the same problem on a 2-edge connected cubic multigraph with $2m$ vertices and $3m$ edges.*

Proof. Order the edges adjacent to each vertex such that two edges that are adjacent on the same trail are consecutive in the order. Replace each single vertex v with a cycle of length $\deg(v)$, with each vertex of the new cycle inheriting a corresponding neighbour of v. Note that for a vertex of

degree 2, this creates a pair of parallel edges, so the result may be a multigraph. Since edges on the same trail are neighbours, we can make the cycle-edge between the two corresponding vertices belong to the same trail. The rest of the cycle edges form new length 1 trails. This graph has exactly $2m$ vertices and $3m$ edges, and any one-way trail orientation on this graph translates to a one-way trail orientation of the original graph. The graph is constructed in $\mathcal{O}(m+n)$ time.

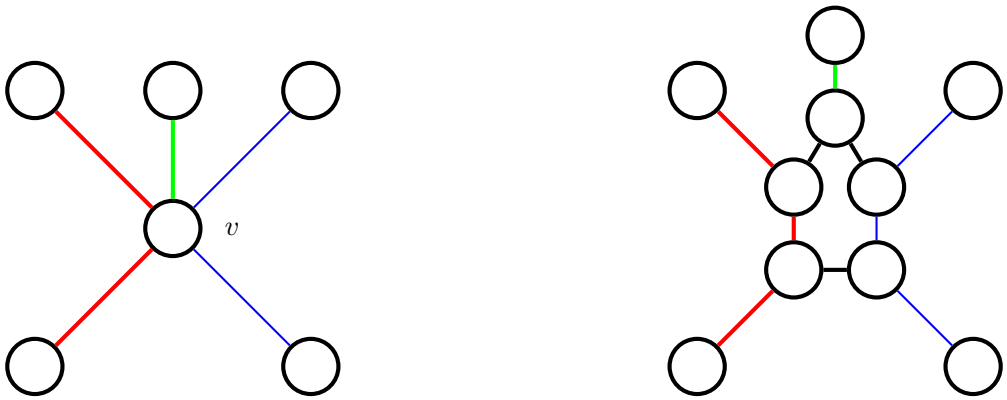


Figure 4: A node of degree 5 turns into a cycle of length 5

□

Recall now that a graph C is called a *cactus* if it is connected and each edge is contained in at most one cycle. If G is any connected graph we let $C_1, \ldots, C_k$ be its 3-edge connected components. It is well known that if we contract each of these we obtain a cactus graph. For a proof of this result see section 2.3.5 of [7]. As the cuts in a contracted graph are also cuts in the original graph we have that if G is 2-edge connected then the cactus graph is 2-edge connected. The edges of the cactus are exactly the edges of G which are part of a 2-edge cut. We will call these edges *2-edge critical*.

It is easy to check that if a cactus has m edges and n vertices then $m \leq 2(n-1)$. We will be using this result in the proof of the following lemma.

Lemma 7. *Let $G = (V, E)$ be a cubic 2-edge connected multigraph, let $X \subseteq E$, and let $F \supseteq E \setminus X$ be minimal such that $H = (V, F)$ is 2-edge connected. Then H has at least $\frac{2}{5}|X|$ distinct 3-edge connected components.*

Proof. Let $X_{\text{del}} \subseteq X$, be the set of edges deleted from G to obtain H, and let $X_{\text{keep}} = X \setminus X_{\text{del}}$ be the remaining edges in X.

If $|X_{\text{keep}}| \geq \frac{4}{5}|X|$, then by minimality of H there are at least $|X_{\text{keep}}|$ 2-edge-critical edges in H i.e. edges of the corresponding cactus, and thus at least $\frac{1}{2}|X_{\text{keep}}| + 1 \geq \frac{2}{5}|X| + 1$ distinct 3-edge connected components.

If $|X_{\text{keep}}| \leq \frac{4}{5}|X|$ then $|X_{\text{del}}| \geq \frac{1}{5}|X|$, and since G is cubic and the removal of each edge creates two vertices of degree 2 we must have that H has at least $2|X_{\text{del}}| \geq \frac{2}{5}|X|$ distinct 3-edge connected components. □

Lemma 8. *Let $G = (V, E)$ be a connected cubic multigraph with E partitioned into trails. Then G has a spanning tree that contains all edges that are not at the end of their trail.*

Proof. Let F be the set of edges that are not at the end of their trail. Since G is cubic, the graph (V, F) is a collection of vertex-disjoint paths, and in particular it is acyclic. Since G is connected F can be extended to a spanning tree. □

Note that we can find this spanning tree in linear time. Indeed, we may assign weight 0 to edges in F and 1 to the remaining edges and use the so-called[5] Prim's minimal spanning tree algorithm with a suitable priority queue to find the tree.

Lemma 9. *Let $G = (V, E)$ be a cubic 2-edge connected multigraph with E partitioned into trails. Let T be a spanning tree of G containing all edges that are not at the end of their trail. Let H be a minimal subgraph of G that contains T and is 2-edge connected. Then for any $k \geq 5$, less than $\frac{4}{5}\frac{k}{k-1}|V|$ of the vertices in H are in a 3-edge connected component with at least k vertices.*

Proof. Let X be the set of edges that are not in T. Since G is cubic, $|X| = \frac{1}{2}|V| + 1$. By Lemma 7 H has at least $\frac{2}{5}|X| > \frac{1}{5}|V|$ 3-edge connected components. Each such component contains at least one vertex, so the total number of vertices in components of size at least k is less than $\frac{k}{k-1}\left(|V| - \frac{1}{5}|V|\right) = \frac{4}{5}\frac{k}{k-1}|V|$. □

Definition 10. Let C be a 3-edge connected component in some graph H, whose edges is partitioned into trails. Define $\Gamma_H(C)$ to be the 3-edge connected graph obtained by replacing each min-cut $\{e, f\}$ where $e = (e_1, e_2)$ and $f = (f_1, f_2)$ and $e_1, f_1 \in C$ with a single new edge (e_1, f_1). Define the corresponding partition of the edges of $\Gamma_H(C)$ into trails by taking every trail that is completely contained in C, together with new trails combined from the fragments of the trails that were broken by the min-cuts together with the new edges that replaced them. See figure 5.

At this point the idea of the algorithm can be explained. We remove as many of the edges, at the end of their trails, as we can still maintaining that the graph is 2-edge connected. Lemma 9 guarantees that we obtain a graph H with $\Omega(|V|)$ many 3-edge connected components of size $O(1)$. We solve the problem for each $\Gamma_H(C)$ for every 3-edge connected component. Finally, we combine the solutions for the different components like in the proof of theorem 2.

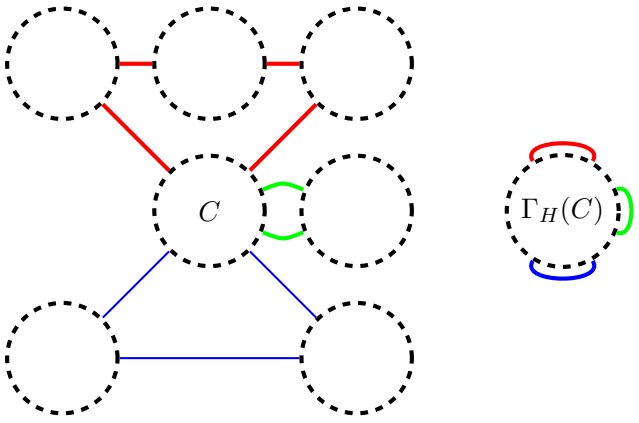


Figure 5: 3-edge connected components, notice how every edge out from the centre is part of a cycle. This right hand shows $\Gamma_H(C)$ where C is the component in the middle.

[5]Originally discovered by Jarník [4], later rediscovered by Prim [8]

Theorem 11. *The one-way trail orientation problem can be solved in $\mathcal{O}(m+n)$ time on any 2-edge connected graph or multigraph with n vertices and m edges.*

Proof. By Lemma 6, we can assume the graph is cubic. For the algorithm we will use two subroutines. First of all when we have found the minimum spanning tree T containing the edges that are not on the end of their trail we can use the algorithm of Kelsen [5] to, in linear time, find a minimal (w.r.t. inclusion) subgraph H of G that contains T and is 2-edge connected. Secondly we will use the algorithm of Melhorn [6] to, in linear time, build the cactus graph of 3-edge connected components. The algorithm runs as follows:

1. Construct a spanning tree T of G that contains all edges that are not at the end of their trail.
2. Construct a minimal subgraph H of G that contains T and is 2-edge connected[6].
3. Find the cactus of 3-edge connected components of[7] H.
4. For each 3-edge connected component C_i, construct $\Gamma_H(C_i)$.
5. Recursively compute an orientation for each[8] $\Gamma_H(C_i)$.
6. Combine the orientations from each component.

First we will show correctness and then we will determine the running time.

Recall that we can flip the orientation in each $\Gamma_H(C_i)$ and still obtain a strongly connected graph respecting the trails in $\Gamma_H(C_i)$. The way we construct the orientation of the edges of G is by flipping the orientation of each $\Gamma_H(C_i)$ in such a way that each cycle in the cactus graph becomes a directed cycle[9]. This can be done exactly because no edge of the cactus is contained in two cycles. By construction this orientation respects the trails so we need to argue that it gives a strongly connected graph.

For showing that the resulting graph is strongly connected, first let every 3-edge connected component be contracted, then the graph is strongly connected since the cycles of the cactus graph have become directed cycles. Now assume inductively that we have uncontracted some of the components and call this graph G_1. Now we uncontract another component C (see figure 6) and obtain a new graph G_2 which we will show is also strongly connected. If $u, v \in C$, then since $\Gamma_H(C)$ is strongly connected there is a path from u to v in $\Gamma_H(C)$. If this path only contains edges which are edges in C clearly this path also exists in G_2 so we are done. If the path uses one of the added edges (e_1, f_1) (without loss of generality oriented from e_1 to f_1), it is because there are edges (e_1, e_2) and (f_1, f_2) forming a cut and thus being part of a cycle in the cactus. In this case we use edge (e_1, e_2) to leave component C and then go from e_2 back to component C which is possible since G_1 was strongly connected. When we get back to the component C we must arrive at f_1 since otherwise there would be two cycles in the cactus containing the edge (e_1, e_2). Hence the edge (e_1, f_1) was not needed. This argument can be used for any of the edges of $\Gamma_H(C)$ that are not in C and thus we can move between any two nodes in C. Since G_1 was strongly connected this

[6] See Kelsen [5]

[7] See Melhorn [6]

[8] Note that $\Gamma_H(C_i)$ is cubic unless it consists of exactly one node. In this case however we don't need to do anything.

[9] In practise this is done by making a DFS (or any other search tree one likes) of the cactus and repeatedly orienting each component in a way consistent with the previous ones.

suffices to show that G_2 is strongly connected. By induction this implies that after uncontracting all components the resulting graph is strongly connected.

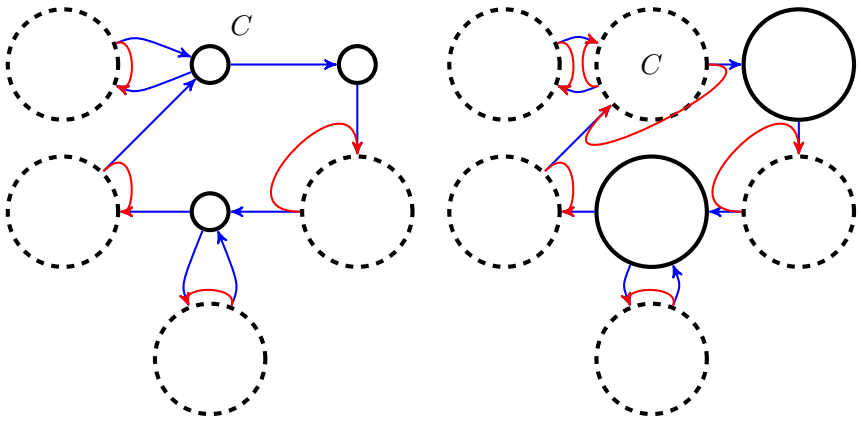


Figure 6: Before and after uncontracting component C

Now for the running time. By Lemma 9 each level of recursion reduces the number of vertices in "large" components by a constant fraction, for instance for $k = 10$ we reduce the number of vertices in large components by a factor of $\frac{1}{9}$. Let $f(n)$ be the worst case running time with n nodes for a cubic graph, and pick c large enough such that cn is larger than the time it takes to go through steps 1-4 and 6 as well as computing the orientations in the "small" components. Let $a_1, \ldots, a_k$ be the number of vertices in the "large" 3-edge connected components. Then $\sum_i a_i \leq \frac{8n}{9}$ and

$$f(n) \leq cn + \sum_i f(a_i)$$

Inductively we may assume that $f(a_i) \leq 9cn$ and thus obtain

$$f(n) \leq cn + \sum_i f(a_i) \leq cn + \sum_i 9ca_i = cn + 8cn = 9cn$$

proving that $f(n) \leq 9cn$ for all n.

□

6 Open problems

We here mention two problems concerning trail orientations which remain open.

First of all, our linear time algorithm for finding trail orientations only works for undirected graphs and it doesn't seem to generalise to the trail orientation problem for mixed graphs. It would be interesting to know whether there also exists a linear time algorithm working for mixed graphs. If so it would complete the picture of how fast an algorithm we can obtain for any variant of the trail orientation problem.

Secondly, our sufficient condition for when it is possible to solve the trail orientation problem for mixed multigraphs is clearly not necessary. It would be interesting to know whether there is a

Algorithm 2: Linear time algorithm for cubic graphs.

Input: An undirected multigraph G and a partition $\mathcal{P}$ of the edges of G into trails.

Output: True if $(G, \mathcal{P})$ has a strong trail orientation, otherwise false. G is modified in place, either to have such a strong trail orientation, or to a forced graph that is not strongly connected.

```
1  Construct a spanning tree T of G that contains all edges that are not at the end of their trail.
2  Construct a minimal subgraph H of G that contains T and is 2-edge connected.
3  Find the cactus C of 3-edge connected components of H.
4  for each 3-edge connected component C_i in C in DFS preorder do
5  |  Construct G_i = Γ_H(C_i).
6  |  Recursively compute an orientation for G_i.
7  |  if the orientation of G_i is not compatible with its DFS parent then
8  |  |  Flip orientation of G_i
9  |  end
10 end
11 for each edge e deleted from G to create H do
12 |  if no edge on the trail of e has been oriented yet then
13 |  |  Pick an arbitrary orientation for e.
14 |  else
15 |  |  Set the orientation of e to follow the trail.
16 |  end
17 end
```

simple necessary and sufficient condition like there is in the undirected case. Since in the mixed case the answer to the problem actually depends on the given trail decomposition and not just on the connectivity of the graph it is harder to provide such a condition. One can give the following condition. It is possible to orient the trails making the resulting graph strongly connected if and only if when we repeatedly direct the forced trails end up with a graph satisfying our condition in theorem 4. This condition is not simple and is not easy to check directly. Is there a more natural condition?

References

[1] Frank Boesch and Ralph Tindell. Robbins's theorem for mixed multigraphs. *The American Mathematical Monthly*, 87(9):716–719, 1980.

[2] Fan R. K. Chung, Michael R. Garey, and Robert E. Tarjan. Strongly connected orientations of mixed multigraphs. *Networks*, 15(4):477–484, 1985.

[3] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[4] V. Jarník. *O jistém problému minimálním: (Z dopisu panu O. Borůskovi)*. Práce Moravské přírodovědecké společnosti. Mor. přírodovědecká společnost, 1930.

[5] Pierre Kelsen and Vijaya Ramachandran. On finding minimal 2-connected subgraphs. In *Proceedings of the Second Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 28-30 January 1991, San Francisco, California.*, pages 178–187, 1991.

[6] Kurt Mehlhorn, Adrian Neumann, and Jens M. Schmidt. Certifying 3-edge-connectivity. *Algorithmica*, 77(2):309–335, 2017.

[7] Hiroshi Nagamochi and Toshihide Ibaraki. *Algorithmic Aspects of Graph Connectivity*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[8] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[9] H. E. Robbins. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly*, 46(5):281–283, 1939.