UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PhD Thesis

Algorithms and Data Structures for Graphs

Eva Rotenberg

Supervised by Mikkel Thorup
Co-supervised by Christian Wulff-Nilsen

This thesis has been submitted on August 25, 2017 to the PhD School of The Faculty of Science, University of Copenhagen

Abstract

This PhD-dissertation within theoretical computer science concerns algorithms and data structures for graphs. The contribution consists of the following parts:

Reachability for Planar Directed Graphs We show how to represent a planar digraph in linear space so that reachability queries can be answered in constant 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].

A Hamilton Cycle in the Square of a 2-connected Graph in Linear Time Fleischner's theorem says that the square of every 2-connected graph contains a Hamiltonian cycle. We present a proof resulting in an $O(|E|)$ algorithm for producing a Hamiltonian cycle in the square G^2 of a 2-connected graph $G = (V, E)$. More generally, we get an $O(|E|)$ algorithm for producing a Hamiltonian path between any two prescribed vertices, and we get an $O(|V|^2)$ algorithm for producing cycles $C_3, C_4, \ldots, C_{|V|}$ in G^2 of lengths $3, 4, \ldots, |V|$, respectively.

Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time We present a deterministic fully-dynamic data structure for maintaining information about the bridges in a graph. We support updates in $\widetilde{O}((\log n)^2)$ 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 $\mathcal{O}(m + n \log n \log\log n)$. Finally, we show how to obtain $\mathcal{O}(\log n / \log\log n)$ query time, matching the optimal trade-off between update and query time.

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

Decremental SPQR-trees for planar graphs We present a decremental data structure for maintaining the SPQR-tree of a planar graph subject to contractions and deletions of edges. The update time, amortized over $\Omega(n)$ operations, is $O(\log^2 n)$.

Via SPQR-trees, we show a decremental algorithm for maintaining 2- and 3-vertex connectivity in planar graphs. It answers queries in $O(1)$ time and processes edge deletions and contractions in $O(\log^2 n)$ amortized time. For 3-vertex connectivity in a planar graph subject to deletions, this is an exponential improvement over the previous best bound of $O(\sqrt{n})$ that has stood for over 20 years. In addition, the previous data structures only supported edge deletions.

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 known strategy 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.

Danish Abstract

Denne PhD-afhandling i teoretisk datalogi omhandler algoritmer og datastrukturer for grafer. Bidraget består af følgende dele:

Tilgængelighed i plane orienterede grafer Vi viser, hvordan en plan orienteret graf kan repræsenteres på lineær plads, således at tilgængelighedsforespørgsler kan svares i konstant tid. Denne repræsentation af tilgængelighed er derfor optimal i både tid og plads. Den tidligere bedste løsning havde et pladsforbrug på $O(n \log n)$ ord og havde konstant forespørgselstid [Thorup FOCS'01].

En Hamiltonkreds i kvadratet på en tosammenhængende graf i lineær tid Fleischners sætning siger, at kvadratet på enhver tosammenhængende graf indeholder en Hamiltonkreds. Vi giver et bevis som resulterer i en lineærtidsalgoritme til at beskrive en Hamiltonkreds i kvadratet G^2 af en tosammenhængende graf G. Mere generelt opnår vi en lineærtidsalgoritme til at beskrive en Hamiltonsti, der forbinder vilkårlige to på forhånd angivne knuder, og vi opnår en $O(n^2)$ algoritme til at beskrive kredse $C_3, C_4, \ldots C_n$ i G^2 af længde $3, 4, \ldots n$, hvor n er antallet af knuder i G.

Dynamisk Brodetektion in $\tilde{O}(\log^2 n)$ amortiseret tid Vi giver en fuldt-dynamisk datastruktur, som vedligeholder information om broer i en graf. Datastrukturen understøtter indsættelser og sletninger af kanter i $\widetilde{O}((\log n)^2)$ amortiseret opdateringstid, og kan finde en bro i samme komponent som en specificeret knude i $O(\log n / \log \log n)$ tid i værste tilfælde. Disse køretider afviger kun med $\log \log n$-faktorer fra de bedste køretider for deterministisk fuldt-dynamisk grafsammenhæng.

Den hidtil bedste datastruktur for dynamisk brodetektion af Thorup [STOC2000] havde en amortiseret opdaterings tid på $\widetilde{O}((\log n)^3)$, og var en bittrickbaseret forbedring af datastrukturen af Holm et al. [STOC98, JACM2001], som havde $\mathcal{O}((\log n)^4)$ amortiseret opdateringstid.

Vores resultat baserer sig på en anden og rent kombinatorisk forbedring af datastrukturen af Holm et al., som i sig selv giver en ny kombinatorisk datastruktur med en amortiseret opdateringstid på $\widetilde{O}((\log n)^3)$. Kombineret med Thorups bittrick opnås $\widetilde{O}((\log n)^2)$ amortiseret tid.

Vi forbedrer også datastrukturens pladsforbrug. Vi beskriver et generelt redskab, som kan anvendes på begge vore nye algoritmer såvel som på de gamle, som reduceder pladsforbuget til $O(m+n)$. Dette er en forbedring over de tidligere datastrukturers pladsforbrug på $O(m+n \log n \log \log n)$. Endelig viser vi hvordan man opnår en forespørgselstid på $O(\log n / \log \log n)$, hvilket er bedst muligt for den angivne opdateringstid.

Vore resultater forbedrer køretiden for at afgøre hvorvidt en graf indeholder en unik perfekt parring.

Vedligeholdelse af SPQR-træer for plane grafer under sletning og sammentrækning af kanter Vi giver en datastruktur, som vedligeholder et SPQR-træ for en plan graf under sletning og sammentrækning af kanter. Opdateringstiden, amortiseret over $\Omega(n)$ operationer, er $O(\log^2 n)$.

Via SPQR-træer viser vi, hvordan man kan vedligeholde information om to- og tresammenhæng af knuder i en plan graf under sletning og sammentrækning af kanter. Forespørgsler kan besvares i konstant tid, og opdateringer tager $O(\log^2 n)$ amortiseret tid.

Dette er en eksponentiel forbedring af den tidligere bedste datastruktur til at håndtere forespøgsler om tresammenhæng af knuder i en plan graf under sletning af kanter, da den tidligere bedste datastruktur havde en opdateringstid på $O(\sqrt{n})$, hvilket stod uforbedret i over 20 år. Ydermere understøtter vi som noget nyt ikke kun kantsletninger men også sammentrækning af kanter.

Online bipartit parring med $O(\log^2 n)$ udskiftninger I problemet ved navn online bipartit parring med udskiftninger er den ene side af en bipartit graf givet på forhånd, og knuderne på den anden side ankommer en ad gangen sammen med alle deres kanter. Målet er at vedligeholde en parring, som til alle tider har højest mulig kardinalitet, men at foretage færrest muligt ændringer (udskiftninger) i parringen per knudeankomst. Vi viser at den grådige algoritme, som altid augmenterer en korteste vej fra den nyest ankomne knude (SAPprotokollen) foretager højest amortiseret $O(\log^2 n)$ udskiftninger per ankommen knude, hvor n er det samlede antal ankomne knuder. Dette er den første analyse til at opnå polylogaritmisk udskiftningstal for *nogen* udskiftningsstrategi, og opnår næsten den nedre

grænse på $\Omega(\log n)$. Den hidtil bedste strategi opnåede amortiseret $O(\sqrt{n})$ udskiftninger [Bosek, Leniowski, Sankowski, Zych, FOCS 2014]. Specifikt for SAPprotokollen var der ikke kendt nogen bedre analyse end den trivielle øvre grænse på $O(n)$. Det følger umiddelbart af vores analyse at den samme øvre grænse på $O(\log^2 n)$ genplaceringer gælder for versionen hvor hver af de statiske knuder har en på forhånd angivet kapacitet.

Vi analyserer også problemet om til alle tider at minimere maksimum belastning af den statiske side med færrest muligt genplaceringer. Vi viser, at hvis den endelige graf har belastning L, så udfører SAPprotokollen amortiseret $O(\min\{L \log^2 n, \sqrt{n} \log n\})$ genplaceringer. Vi viser en næsten tilsvarende nedre grænse, idet $\Omega(\min\{L, \sqrt{n}\})$ genplaceringer kan være nødvendige.

Preface

This dissertation was prepared at the Department of Computer Science, University of Copenhagen, Denmark. The format is that of a *"synopsis with manuscripts of papers or already published papers attached"*, as specified in the *General rules and guidelines for the PhD programme* in the Faculty of Science, University of Copenhagen. All included results were obtained during my enrolment as a Ph.D.-student from December 2013 to August 2017.

The dissertation presents only a selection of my work. For coherency, I have chosen to focus on algorithms and data structures for graphs. Amongst my results on algorithms and data structures for graphs, this dissertation includes a subset, aiming to highlight the results that are strongest and most elegant. For completeness, here follows a list of all the papers and manuscripts written during my enrolment as a PhD student. A total of 9 papers have been published at peer-reviewed venues, and 6 are still in submission.

Full list of articles and manuscripts written during my Ph.D.

[1] Aamand, A., Hjuler, N., Holm, J., and Rotenberg, E. One-way trail orientations. *ArXiv e-prints arXiv:1708.07389* (Aug. 2017). In submission

[5] Abrahamsen, M., Bodwin, G., Rotenberg, E., and Stöckel, M. Graph reconstruction with a betweenness oracle. In *33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)* (2016), N. Ollinger and H. Vollmer, Eds., Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 5:1–5:14

[6] Abrahamsen, M., Holm, J., Rotenberg, E., and Wulff-Nilsen, C. Best Laid Plans of Lions and Men. In *33rd International Symposium on Computational Geometry (SoCG 2017)* (2017), B. Aronov and M. J.

Katz, Eds., vol. 77 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16

[8] Alstrup, S., Georgakopoulos, A., Rotenberg, E., and Thomassen, C. A Hamiltonian cycle in the square of a 2-connected graph in linear time. In submission

[18] Bernstein, A., Holm, J., and Rotenberg, E. Online bipartite matching with amortized $O(\log^2 n)$ replacements. *ArXiv e-prints arXiv:1707.06063* (July 2017). In submission

[31] Cohen-Addad, V., de Mesmay, A., Rotenberg, E., and Roytman, A. The bane of low-dimensionality clustering. In submission., 2017

[33] Dahlgaard, S., Knudsen, M. B. T., Rotenberg, E., and Thorup, M. The power of two choices with simple tabulation. In *SODA'16* (2016), SIAM, pp. 1631–1642

[32] Dahlgaard, S., Knudsen, M. B. T., Rotenberg, E., and Thorup, M. Hashing for statistics over k-partitions. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS'15* (2015), IEEE, pp. 1292–1310

[86] Holm, J., Italiano, G. F., Karczmarz, A., Łącki, J., Rotenberg, E., and Sankowski, P. Contracting a Planar Graph Efficiently. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria* (Sept. 2017)

[85] Holm, J., Italiano, G. F., Karczmarz, A., Łącki, J., and Rotenberg, E. Decremental SPQR-trees for planar graphs. In submission, 2017

[87] Holm, J., and Rotenberg, E. Dynamic planar embeddings of dynamic graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany* (2015), E. W. Mayr and N. Ollinger, Eds., vol. 30 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 434–446

[88] Holm, J., and Rotenberg, E. Dynamic planar embeddings of dynamic graphs. *Theory of Computing Systems* (2017), 1–30

[89] Holm, J., Rotenberg, E., and Thorup, M. Planar reachability in linear space and constant time. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS'15* (Oct 2015), pp. 370–389

[90] Holm, J., Rotenberg, E., and Thorup, M. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. *arXiv preprint arXiv:1707.06311* (2017). In submission

[91] Holm, J., Rotenberg, E., and Wulff-Nilsen, C. Faster fully-dynamic minimum spanning forest. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, Sept. 14-16, 2015, Proceedings* (2015), pp. 742–753

Acknowledgements I am grateful to my advisors Mikkel Thorup and Christian Wulff-Nilsen for their guidance and support. I thank Jacob Holm for countless scientific discussions and eminent team work. I thank Carsten Thomassen and DTU Compute for their hospitality during my 6 months exchange visit, and, in particular, I thank Carsten Thomassen for many interesting discussions. I thank Guiseppe Italiano at Universitá Roma Tor Vergata for his hospitality during my visit and for many interesting discussions.

I am grateful to the Department of Computer Science, University of Copenhagen, for the support and encouragement, and to my colleagues for their interest, for their support, and for good company.

I wish to extend my gratitude to my many co-authors with whom I collaborated during my PhD-studies: Jacob Holm, Mikkel Thorup, Christian Wulff-Nilsen, Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Mikkel Abrahamsen, Jakub Łącki, Giuseppe Italiano, Adam Karczmarz, Gregory Bodwin, Morten Stöckel, Piotr Sankowski, Carsten Thomassen, Agelos Georgakopoulos, Stephen Alstrup, Aaron Bernstein, Vincent Viallat Cohen-Addad, Arnaud de Mesmay, Alan Roytman, Anders Aamand, and Niklas Hjuler.

Contents

Chapter 1

Introduction

This work is within theoretical computer science, and concerns algorithms and data structures for graphs. A graph consists of a set of vertices and a set of edges between vertices. Graphs are a popular mathematical model for road maps, communication networks, electrical circuits, social networks, transmission of diseases, job assignments, resource allocation, and more. Although the problems studied in this paper are motivated and inspired by a large collection of real-world problems, and though they may form a basis for improving existing methods for calculating solutions to these problems, we have not taken the final step in the form of implementation and comparison with existing methods. Instead, we focus on the purely theoretical task of designing and analysing algorithms and data structures in popular computational models.

Common threads This dissertation includes work on several very different but interesting graph problems. There are however a few common themes, that each occur in more than one chapter. Planar graphs, data structures, and higher connectivity. I will now give a short introduction and motivation of these.

Planar graphs are graphs which can be embedded in the plane, that is, drawn without edge-crossings. Planar graphs can be used as models for road maps, but e.g. also for a layer of a microchip. Planar graphs have more structure than general graphs, which can be used to make algorithms and data structures for them that are more elegant and even more efficient than what is possible for a general graph. Planar graphs have been studied by mathematicians for centuries.

Data structures for static graphs are representations of the graph that

quickly reply to queries about properties of the graph, or about properties of vertices of the graph. The ideal data structure takes up little space, responds quickly to queries, and has a fast construction time.

Data structures for dynamic graphs. Furthermore, we study the situation in which there are changes to the graph. E.g. edges are deleted, inserted, or contracted, and the data structure needs to be updated such that it still replies quickly to queries. This is indeed motivated by the real world which is not a static thing: Links in the communication network fail, a road becomes inaccessible or slow, or — social networks change all the time. There are also theoretical motivations for studying dynamic graphs: various algorithms for static graphs use a dynamic data structure as a subroutine. In the case of data structures for dynamic graphs, the ideal data structure has quick construction time, query time, and update time, and takes up little space.

Data structures for online problems with replacements. We also study the setting in which vertices of a graph arrive one by one with all their edges. In particular, we study the setting in which we have a set of "server" vertices that are given in advance, and a set of "client" vertices that arrive one by one with all their edges, and where all edges go between a client and a server. Motivations for this problem include resource allocation, hashing, data streaming, job scheduling, and data storage [28]. We maintain an assignment of clients to servers in this setting. In this case, we do not only wish to calculate the new assignment as fast as possible, but we also want to minimise the number of changes to the assignment (replacements) done through the course of the algorithm, while adhering to constraints such as not exceeding the capacity of the servers, or minimising maximum server load.

Higher connectivity. An undirected graph is connected if for any two vertices of the graph, there is a path connecting them. However, for many practical purposes, e.g. in communication networks, it is important for the graph to be highly connected: If the removal of few vertices or edges would disconnect the graph, this points to a vulnerability in the network; an indication that the network connectivity may be about to break down. Higher connectivity also has theoretical implications, indeed, we give a linear time algorithm for outputting a Hamilton Cycle in the square of a highly connected graph (to be precise, a 2-vertex connected graph), but in general, the problem of outputting a Hamilton cycle in the square of a graph is NP-complete. This dissertation also contains results on maintaining information about higher connectivity in a dynamic graph subject to insertions and deletions, or deletions and contractions of edges.

1.1 Outline

In addition to this general introduction, the dissertation consists of the following papers and manuscripts, which were written during my PhD studies.

Chapter 2 *Planar Reachability in Linear Space and Constant Time.* Jacob Holm, Eva Rotenberg, and Mikkel Thorup. This paper is published in the proceedings of the IEEE 56th Annual Symposium on Foundations of Computer Science 2015.

Chapter 3 *A Hamiltonian Cycle in the Square of a 2-connected Graph in Linear Time* Stephen Alstrup, Agelos Georgakopoulos, Eva Rotenberg, Carsten Thomassen. In submission.

Chapter 4 *Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time* Jacob Holm, Eva Rotenberg, Mikkel Thorup. In submission.

Chapter 5 *Decremental SPQR-trees for planar graphs* Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki and Eva Rotenberg. In submission.

Chapter 6 *Online bipartite matching with amortized $O(\log^2 n)$ replacements* Aaron Bernstein, Jacob Holm, Eva Rotenberg. In submission.

With minor exceptions, these papers and manuscripts appear in their original published or submitted form. For this reason, notation and terminology are not always consistent throughout the dissertation.

In the the remaining part of this introductory chapter, these results, as well as selected other results obtained during my PhD studies, are set in context, some of the technical contributions are sketched, and some potential future directions of research are mentioned. The chapter contains, first, some preliminary notation and terminology, and brief surveys of dynamic higher connectivity and dynamic data structures for planar graphs, and of online algorithms with replacements, setting my contributions to these areas in context. Then, a section concerning each chapter of the dissertation, and short sections on other related results obtained during my PhD studies.

1.2 Preliminaries

In this section, some definitions and terminology that is used across the introduction to the dissertation are introduced. The intention is that these

preliminaries equip the reader to read the introduction, have some understanding of the results presented in the thesis, their relations to previous work, and a high-level understanding of the techniques. This section is based on papers and manuscripts [8, 18, 85, 88–91], and text books [21, 38, 114].

Computational model Unless otherwise specified, the computational model for all upper bounds is the word-RAM model with standard operations. We assume that a machine word is at least the logarithm of the input size, and that standard operations (in AC^0) on words take constant time. We measure space in number of words stored, and time in number of operations performed. This model, also called practical RAM, is a much used model for programmes in languages such as C running on problem instances that fit into the computer's working memory. We will explicitly state whenever an analysis relies on constant time multiplication of two variables.

1.2.1 Higher connectivity

The connected components (also called components) of an undirected graph are the equivalence classes of vertices that can reach each other. That is, the equivalence classes induced by the relation $u \sim v$ for for every edge uv of the graph.

In this section, we generalise this notion to higher connectivity for for undirected graphs.

k-vertex connectivity For a positive integer k, a graph is *k-vertex connected* if and only if it is connected, has at least k vertices, and stays connected after removing any set of at most $k-1$ vertices. A pair of vertices are said to be *locally k-vertex connected* if there are k internally vertex-disjoint paths connecting them.

In Chapter 3, we refer to 2-vertex connectivity as simply 2-*connectivity*. In Chapter 5, we refer to 2-vertex connectivity and to local 2-vertex connectivity as *biconnectivity*, and to 3-vertex connectivity as well as local 3-vertex connectivity as *triconnectivity*.

Proper ear decompositions Given a graph, an ear is a trail that may start and end in the same point, but which otherwise has no repetition of edges or vertices, that is, each internal vertex has degree 2, and the endpoints have degree $\leqslant 2$. An *ear decomposition* of a non-trivial graph (i.e. a graph with at least one edge) is a partitioning of its edges into a sequence of ears, such that the first is a cycle, the endpoints of each ear belong to previous

ears, and each vertex of the graph is internal in exactly one ear. An ear decomposition is *proper* (also referred to as an open ear decomposition in the literature), if for all but the first ear, the end-vertices are distinct. It is a result by Robbins [143] that a graph admits a proper ear decomposition if and only if it is 2-vertex connected.

Cutpoints and block-cutpoint trees If a connected graph is not 2-vertex connected, there is one graph whose removal causes the graph to fall apart. This vertex is called a *cut-vertex*, *cutpoint* or an *articulation point*. The maximal (with respect to inclusion) 2-vertex connected components of a graph are called *blocks*. For a connected graph G, the *block-cutpoint tree* is a tree whose vertices represent blocks of G or articulation points of G, and where there is an edge between an articulation a point and a block B when $a \in B$. The block-cutpoint tree is also denoted the *block-cut tree* or the *BC-tree* in the literature.

Separation pairs and SPQR-trees Similarly, if a graph is not 3-vertex connected, there is a pair of vertices whose removal cause the graph to "fall apart" (see Definition 1.1). Such a pair is called a *separation pair* throughout this dissertation. A generalisation of block-cutpoint trees representing how locally 3-vertex connected components relate to each other in a 2-vertex connected graph is the SPQR-tree (see Definition 1.2 and Figure 1.1).

Definition 1.1 (Hopcroft and Tarjan [93, p. 6])**.** Let $\{a, b\}$ be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1, E_2, \ldots, E_k$, such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an end-point are in the same class. The classes E_i are called the *separation classes* of G with respect to $\{a, b\}$. If there are at least two separation classes, then $\{a, b\}$ is a *separation pair* of G unless (i) there are exactly two separation classes, and one class consists of a single edge, or (ii) there are exactly three classes, each consisting of a single edge.

Definition 1.2 (Chapter 5 Definition 5.3)**.** The SPQR-tree for a biconnected multigraph $G = (V, E)$ with at least 3 edges is a tree with nodes labelled S, P, or R, where each node x has an associated *skeleton graph* $\Gamma(x)$ with the following properties:

- For every node x in the SPQR tree, $V(\Gamma(x)) \subseteq V$.

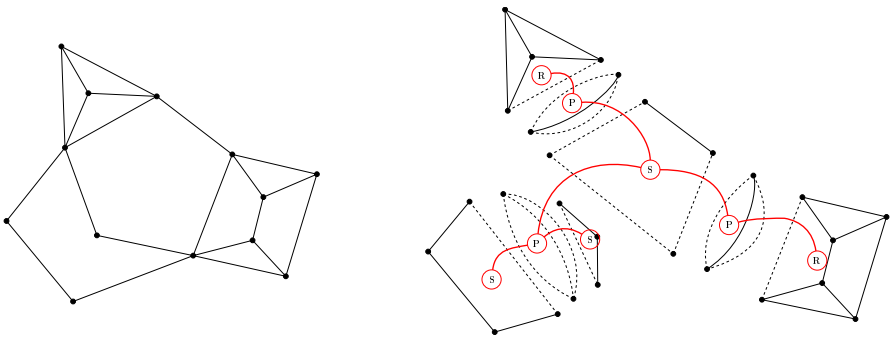

Figure 1.1: A biconnected graph and its SPQR tree. See Definition 1.2. This figure is a citation of Chapter 5 Figure 5.2.

- For every node x in the SPQR tree, every edge in $\Gamma(x)$ is either in E or a *virtual edge* corresponding to an edge (x, y) in the SPQR-tree.
- For every edge $e \in E$ there is a unique node x in the SPQR-tree such that $e \in E(\Gamma(x))$.
- For every edge (x, y) in the SPQR tree, $V(\Gamma(x)) \cap V(\Gamma(y))$ is a separation pair $\{a, b\}$ in G, and there is a virtual edge ab in each of $\Gamma(x)$ and $\Gamma(y)$.
- If x is an S-node, $\Gamma(x)$ is a simple cycle with at least 3 edges.
- If x is a P-node, $\Gamma(x)$ consists of a pair of vertices with at least 3 parallel edges.
- If x is an R-node, $\Gamma(x)$ is a simple triconnected graph.
- No two S-nodes are neighbors, and no two P-nodes are neighbors.

The SPQR-tree for a biconnected graph is unique (see e.g. [36] for a proof of this theorem). The nodes of the SPQR-tree, as well as the skeleton graphs associated with these, are referred to as the triconnected components of G, the 3-vertex connected components of G, or the locally 3-vertex connected components of G.

k-edge connectivity For a positive integer k, a graph is *k-edge connected* if and only if it stays connected after removing any set of at most $k-1$ edges. A pair of vertices are said to be *k-edge connected* if they are still connected

after the removal of any set of $k-1$ edges. By Menger's Theorem [131], this is equivalent to saying that a pair of vertices are k-edge connected if there exist k edge-disjoint paths between them. By edge-disjoint is meant that no edge appears in both paths.

A connected graph is not 2-edge connected if there exists an edge whose removal would disconnect the graph. Throughout this dissertation we call such an edge a *bridge*, but it is also referred to as a *cut-edge* or an *ithmus* in the literature.

1.2.2 Hamilton cycles, Euler tours, matchings, and bipartiteness

Euler tours and paths In a graph, an Euler tour is a trail that uses each edge exactly once. Similarly, an Euler path from the vertex u to the vertex v is a trail using each edge exactly once whose endpoints are u and v. An Euler tour exists if and only if it is connected and all vertices have even degree (such a graph is for this reason called *Eulerian*), and can be calculated in linear time [81].

Hamilton cycles and paths A Hamilton cycle, on the other hand, is a cycle which visits each vertex exactly once. Similarly, given a pair of vertices, a simple path between them is a *Hamilton path* if it visits each vertex of the graph exactly once. In general, determining whether a graph contains a Hamilton cycle (or path) is NP-hard [105], and no polynomial time algorithm is known. In general, determining whether the square of a graph contains a Hamilton cycle (or path) is also NP-hard. Here, the square of a graph G is a graph on the same vertex set which has an edge for every path of length 1 or 2 in G. Fleischner's theorem says that the square of a 2-vertex connected graph has a Hamilton cycle.

Bipartite graphs A graph is said to be *bipartite* if the vertex set can be partitioned into sets A and B, and all edges are of the form ab with $a \in A$ and $b \in B$.

Matchings A *matching* of a graph is a subset of edges such that no edges share an end-point. Such a matching is called *maximum* if it has maximal cardinality, and *perfect* if every vertex of the graph is the endpoint of exactly one edge.

A theorem by Kotzig [118] states that if a graph has a unique perfect matching, then it contains a bridge that belongs to this unique perfect matching. This is used in an algorithm by Gabow, Kaplan, and Tarjan [58] for determining whether a given graph has a unique matching, using decremental 2-edge connectivity as a subroutine (see Section 1.2.1).

1.2.3 Planar graphs

For a connected planar graph G embedded in the plane[1], its dual G^* is a graph with a vertex for each face of the embedding, and an edge $e^* = f^*g^*$ for each edge e of G incident to faces f and g. For a general graph G embedded in the plane, we define its dual G^* to be the (disjoint) union of the duals of its components. The face-degree ofs a face f in the embedded graph G is the degree of f^* in G^*. Equivalently, the dual can be defined combinatorially, see Klein and Mozes [114].

The vertex-face graph For a connected planar embedded non-trivial multigraph G, a *corner*[2] of G an ordered pair of (not necessarily distinct) edges (e_1, e_2) such that e_1 immediately precedes e_2 in the clockwise order around some vertex. Define the *vertex-face graph* $G^\diamond$ as the graph whose vertex-set is union of the the vertex-sets of G and G^*, and whose edges are the corners of the embedded graph G. Note that $G^\diamond$ is isomorphic to $(G^*)^\diamond$. Clearly, $G^\diamond$ is bipartite and planar, with a natural embedding given by the embedding of G. Furthermore, each face of $G^\diamond$ has degree 4 and corresponds exactly to an edge of G. Less trivially, $G^\diamond$ is simple if and only if G is loopless and biconnected [26, Theorem 5(i)]).

We will later make use of the obserservation (see Brinkmann et al. [26]) that G is simple and 3-vertex connected if and only if $G^\diamond$ is simple, triconnected and has no non-facebounding 4-cycles.

The dynamic operations on G correspond to dynamic operations on G^* and $G^\diamond$. Deleting a non-bridge edge e of G corresponds to contracting e^* in G^*, and vice versa; contracting an edge e corresponds to deleting the corresponding edge from the dual. Finally, deleting a non-bridge edge or contracting an edge corresponds to adding and then immediately contracting an edge across a face of $G^\diamond$ (and removing two duplicate edges).

[1] A *planar embedding* or an *embedding in the plane* is an isomorphism class of topological embeddings of the graph into $\mathbb{R}^2$, or, equivalently, an ordering of the edges around each vertex such that the Euler Characteristic (as defined in [114]) equals 2.

[2] See [88] and [145] for alternative definitions.

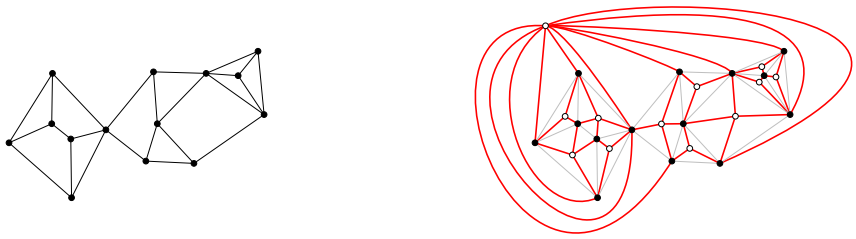

Figure 1.2: Left: a plane embedded graph. Right: the corresponding vertex-face graph (red) and the underlying graph (gray). This figure is Figure 5.1 from Chapter 5.

Separators A property of planar graphs is that they admit small balanced *separators*: Given a planar graph on n vertices, there exists a set of size $O(\sqrt{n})$ whose removal would cause the graph to fall apart into components of size $\leqslant \frac{2}{3}n$. Such a separator can be found in linear time [127].

1.2.4 Top trees

Top trees are a data structure for maintaining information about a dynamic forest subject to deletion and insertion of edges. For each tree in the forest, its top tree has the entire tree as root, and each edge of the tree as leaves. The internal nodes are *clusters*; a generalisation of edges in the sense that they are connected subgraphs with at most 2 boundary vertices. Here, the *boundary vertices* are those incident to something outside the subgraph. Each internal node of the top-tree contains the union of its children.

The top trees support insertions and deletions, but also the expose operation which is particularly useful in order to obtain information about a single vertex or about a tree-path between a pair of vertices. All operations are implemented via splits and merges of clusters, and, as base case, destroys and creates of leaves containing a single edge. A split consists of destroying a cluster and replacing it by its children, and merge creates a parent that contains the union of its children. Thus, when using top-trees as a subroutine in any algorithm or data structure, it is especially necessary to specify two things: what information is propagated to the children when a cluster is split, and how is the information of children combined when clusters are merged.

When each internal node of the top tree has at most 2 children, we say the top tree is binary. Alstrup, Holm, de Lichtenberg, and Thorup [10] show that for a dynamic forest on n vertices, it is possible to maintain binary

top trees of height $O(\log n)$, and support dynamic updates to the forest, as well as expose operations, with a sequence of $O(\log n)$ calls to merge, split, create, and destroy.

Top trees can be generalised to have larger clusters as leaves.

1.3 Dynamic higher connectivity and related results

We call the problem of maintaining connected components in a dynamic graph subject to insertion and deletion of edges *dynamic connectivity*, or, to emphasise that both insertions and deletions are allowed, *fully dynamic* connectivity. When only insertions are allowed, it is called *incremental connectivity*, and when only deletions are allowed, *decremental connectivity*. A dynamic connectivity data structure facilitates queries to whether a pair of vertices are connected. Similarly, the problem of building a data structure that facilitates k-edge connected queries between vertices is called dynamic (or incremental, or decremental) k-edge connectivity. The problem of building a data structure that facilitates locally k-vertex connected queries about pairs of vertices is called dynamic (or incremental, or decremental) k-vertex connectivity. In general for fully dynamic connectivity problems, we let n denote the (fixed) number of vertices, and let m denote the current number of edges in the graph.

There has been a chain of work on dynamic graphs dating back to Fredrickson [56], who invented a data structure for maintaining dynamic minimum spanning forests with deterministic $O(\sqrt{m})$ worst-case update time, and thus, also solving dynamic connectivity in $O(\sqrt{m})$ time. Later, Frederickson gave a data structure for 2-edge connectivity with the same time bounds [57]. Both structures have constant query-time. These were improved to $O(\sqrt{n})$ by Eppstein et al. [48] using their sparsification technique. This was further improved by Henzinger and King [78], who gave a data structure for dynamic minimum spanning forest with amortized $O(n^{1/3} \log n)$ time per operation, and constant worst-case query time.

Since the 1990s, a lot of work has gone into obtaining data structures with *polylogarithmic* update- and query times, that is, times of the form $O(\log^c n)$, for a constant value of c (which we would like to be as small as possible). The first result in this direction was an expected amortized update time of $O(\log^3 n)$ and a query time of $O(\log n / \log \log n)$ for dynamic connectivity by Henzinger and King [79]. Soon after, Henzinger and King gave a data structure with expected amortized $O(\log^5 n)$ update time and $O(\log n)$

query time for fully dynamic two-edge connectivity [78]. The first deterministic data structure with polylogarithmic update- and query time was given by Holm, de Lichtenberg, and Thorup [83, 84], who support fully dynamic connectivity in $O(\log^2 n)$ amortized time, decremental minimum spanning forest in $O(\log^2 n)$ amortized time, fully dynamic minimum spanning forest in $O(\log^4 n)$ amortized time, fully dynamic 2-edge connectivity in $O(\log^4 n)$ amortized time, and fully dynamic 2-vertex connectivity in $O(\log^5 n)$ amortized time per update, and $O(\log n)$ time per query.

These deterministic results have been further improved: Wulff-Nilsen presented a data structure with deterministic $O(\log^2 n/\log\log n)$ amortized update time and $O(\log n/\log\log n)$ query time [164], and Thorup gave improvements to the update times of 2-edge and 2-vertex connectivity [157], achieving update times of $O(\log^3 n \log\log n)$ for 2-edge and $O(\log^4 n \log\log n)$ for 2-vertex connectivity. In [90] we improve this further to $O(\log^2 n \log\log^2 n)$ amortized update time and $O(\log n/\log\log n)$ worst-case query time for 2-edge connectivity, and improve the bounds to $O(\log^3 n \log\log^2 n)$ amortized update time and $O(\log n)$ worst-case query time for 2-vertex connectivity. In [91], we improve the data structure for fully-dynamic minimum spanning forest to obtain an amortized update time of $O(\log^4 n/\log\log n)$.

Allowing randomization yields even better bounds, achieving $\widetilde{O}(\log n)$ expected amortized update- and querytimes for connectivity, where the $\widetilde{O}$-notation hides log log-factors. Indeed, Thorup [157] shows a randomized data structure with amortized $O(\log n \log\log^3 n)$ expected update time and $O(\log n/\log\log\log n)$ worst-case query time. This expected amortized update time has since been improved to $O(\log n \log\log^2 n)$ by Huang et al. [97].

A Monte Carlo-randomized connectivity data structure with polylogarithmic update time was made by Kapron, King and Mountjoy, who gave a data structure with $O(\log^5 n)$ worst-case update time and $O(\log n/\log\log n)$ worst-case query time. This was since improved to an update time of $O(\log^4 n)$ by Gibb, Kapron, King, and Thorn [64].

Recently, a Las Vegas-randomized data structure for fully-dynamic minimum spanning forest with expected subpolynomial worst-case update time was announced by Nanongkai, Saranurak, and Wulff-Nilsen [135].

Dynamic planar graphs For planar embedded graphs, it is less clear what is expected of a fully dynamic data structure. While deletions and contractions of edges preserve planarity, the insertion of an edge may cause the graph to cease to be embeddable in the plane. On the other hand, the insertion of an edge may just be incompatible with this particular embedding,

while another embedding exists that can accommodate the new edge.

Italiano, La Poutré, and Rauch gave a data structure that supports deletions of edges and insertion of an edge across a face in $O(\log^2 n)$ time per operation, and their data structure also answers queries to whether a given pair of vertices lie on the same face [100]. In [88] (see also Section 1.10.1), we give a different data structure which supports these operations, but which also supports changes in the embedding in the form of reversing the embedding in a subgraph that is not 3-vertex connected to the rest of the graph. Such a change to the embedding is called a *flip* in [88]. If the insertion of an edge is possible by changing the embedding by at most one flip, we are able to point to such a flip and perform it, all in $O(\log^2 n)$ time.

The current algorithm known for maintaining whether or not a fully dynamic graph is planar runs in $O(\sqrt{n})$ worst-case update time, and is an improvement by Eppstein [49] upon a data structure with $O(n^{2/3})$ update time by Galil, Sarnak, and Italiano [59]. The first linear time algorithm for determining whether a graph is planar is by Hopcroft and Tarjan [95]. Incremental planarity testing of a graph subject to insertion of edges can be done in $O(\alpha(q, n))$ total time, where q is the number of insertions and n is the number of edges, as shown by La Poutré [120].

Deleting or contracting an edge of an embedded planar graph does not violate the embedding. An interesting question is thus to use information about the embedding to maintain decremental connectivity. A data structure with amortized constant edge-deletion and query time for decremental connectivity in a planar graph is described by Łącki and Sankowski [125]. In [86] (see also Section 1.10.4), we extend the ideas by Łącki and Sankowski [125] to show how to support 2-edge connectivity, 2-vertex connectivity, and maximal 3-vertex connected subgraphs of a planar graph subject to deletions of edges in constant time per operation. The result on 2-edge connectivity immediately implies a linear time algorithm for determining whether a planar graph has a unique perfect matching. Furthermore, in [86], we improve the update time for decremental 2-vertex connectivity and 3-edge connectivity to amortized $O(\log n)$ per edge-deletion. In [85] (see also Chapter 5), we show how to maintain the SPQR-tree and facilitate locally 3-vertex connected queries in a planar graph subject to deletions and contractions of edges: our update time is amortized $O(\log^2 n)$, and the query time is $O(1)$.

Lower bounds For fully dynamic connectivity, and for fully dynamic higher connectivity, there is a cell-probe lower bound by Pătraşcu et al. [141], showing a trade-off between the update time u and query time q, namely,

that $q \log\left(\frac{u}{q}\right) \in \Omega(\log n)$, and, vice versa, $u \log\left(\frac{q}{u}\right) \in \Omega(\log n)$. Pătraşcu also shows a lower bound of $\Omega(\log n)$ for fully dynamic planarity testing.

For other dynamic graph problems, including fully dynamic shortest paths, fully dynamic single-source reachability and fully dynamic strong connectivity, Abboud and Vassilevska-Williams [3] have given conditional polynomial time lower bounds, based on popular conjectures. Even for planar graphs, Abboud and Dahlgaard [2] gave conditional polynomial time lower bounds for dynamic shortest paths, again, based on popular conjectures.

1.4 Online graph algorithms with replacements

In the model of online algorithms, the input to an algorithm is given little at the time, e.g. vertices of a graph are revealed one at the time, along with all edges to previously revealed vertices. The task for the algorithm is to make decisions based on what has been revealed so far. Future information may prove these decisions to be sub-optimal, and thus, one often analyses the competitive ratio between the optimal choices and those made by an online algorithm.

In the model of online algorithms with replacements — alternatively referred to as online algorithms with recourse, or online algorithms with rearrangements — the goal is to maintain an optimal solution while making as few changes to the solution as possible. This model is relevant in settings where changes to the solution are possible, but expensive. The model originally goes back to online Steiner trees, where Imase and Waxman [98] study the setting in which a graph is given, but the set of terminals changes by the promotion and demotion of ordinary vertices of the graph, and the task is to maintain a minimal tree connecting the terminals, that is, a Steiner tree. They show how to maintain an constant factor approximate minimum Steiner tree with amortized $O(\sqrt{K})$ replacements over a sequence of length K. Recently, there have been several improvements for this problem [67,69,123,130].

In the problem of exact online bipartite matchings with replacements, the vertices on one side of a bipartite graph 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. Throughout the section, we let n denote the number of clients. The goal is to at all times maintain a maximum matching, but to minimise the number of changes. The problem was introduced in 1995 by Grove, Kao, Krishnan, and Vitter [66], who showed matching upper and lower bounds of $\Theta(n \log n)$ replacements for the case

where each client has degree two.

When a client arrives, the cardinality of the matching can either increase or stay constant. The cardinality increases if and only if there exists an augmenting path from the new client to an available server, that is, a path alternating between edges in the matching and edges not in the matching. We use the *SAP-protocol* as a term for online algorithms that augment along a shortest path, and say that an algorithm is *not SAP* otherwise.

Chadhuri, Daskalakis, Kleinberg, and Lin [28] showed that SAP augments an expected number of $\Theta(n \log n)$ replacements for a bipartite graph with vertices arriving in random order. 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 Zyck later analyzed the SAP protocol for forests, giving an upper bound of $\mathcal{O}(n \log^2 n)$ replacements [23], later improved to the optimal $\mathcal{O}(n \log n)$ total replacements [24].

For general bipartite graphs, previous to our work, no analysis of SAP showed a better upper bound than the trivial $O(n^2)$ total replacements. Bosek, Leniowski, Sankowski and Zyck [22] gave a different algorithm (not SAP) that achieves a total of $\mathcal{O}(n\sqrt{n})$ replacements. We improve upon this by showing that the SAP-protocol makes a total of $O(n \log^2 n)$ replacements.

Bosek et al. [22] also give an implementation of their non-SAP 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 by Hopcroft and Karp [92]. We give an implementation of the SAP-protocol which takes total time $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$. Our implementation is thus an $O(\sqrt{\log n})$ factor slower, but has the interesting theoretical property that it uses shortest augmenting paths.

We also study the problem of minimizing maximum load: each client is assigned to an adjacent server, and the load of a server is the number of clients assigned to it. The ideal would be to make few reassignments (reassigning a client to a different adjacent server) while at all times minimising the maximum load. We show that this ideal is not possible: For any $L \leqslant \sqrt{n/2}$ divisible by 4, there exists an instance where the final graph requires a load of L and where amortized $\Omega(L)$ reassignments are necessary. This is in stark contrast to the world of approximation algorithms. Here, Gupta, Kumar, and Stein [70] and Bernstein, Kopelowitz, Pettie, Porat, and Stein [19] showed that using only $\mathcal{O}(1)$ amortized changes per client insertion, on can maintain an assignment where the maximum load is at all times within a factor of 8 of optimum.

1.5 On Chapter 2: Planar reachability in Linear Space and Constant Time.

In Chapter 2, we[3] give a linear space reachability oracle for planar graphs. Given as input a planar graph with n vertices, we build a data structure that uses $O(n)$ words and which answers reachable uv in $O(1)$ time for any vertices u, v in the graph. We use the word *oracle* to denote a data structure with constant query time, and we call a data structure for reachable-queries a *reachability* structure. We will refer to vertices of in-degree 0 as *sources*, and vertices of out-degree 0 as *targets*.

1.5.1 Reductions of the problem

A result by Tamassia says that if the graph is an *s-t-graph*, that is, there exists a source s that can reach all vertices, and a target t reachable from all vertices, then there exists an $O(\log n)$ bit labelling scheme for reachability. That is, one may assign each vertex a label of $O(\log n)$ bits, and then any pair of vertices can deduce from comparing their labels whether one is reachable from the other. As a trivial corollary, one may construct a reachability oracle for s-t-graphs by simply making a table of the vertices in the graph and their labels. Their result extends to *truncated* planar s-t-graphs, that is, graphs of the form $G \backslash \{s, t\}$ for some planar s-t-graph G with source s and target t. We thus want to make a reduction from general planar graphs to s-t-graphs.

First, we may note vertices that belong to the same strongly connected component may reach the exact same vertices, and are reachable from the exact same vertices. Thus, we may reduce to the problem of constructing a reachability oracle for an acyclic planar graph. This can be done in linear time using the depth-first search algorithm by Tarjan [154].

Then, we use the reduction from Thorup [158], which is a reduction from planar acyclic graphs to planar *in-out graphs*: graphs which contain a source s that can reach all targets of the graph (see Figure 1.3). The reduction goes via partitioning the graph into alternating in- and out-layers, such that any path is contained in at most two consecutive layers. The construction time is linear.

We show how to reduce further to the case where the acyclic planar graph has a single source that can reach all vertices, and, finally, how to use the solution for truncated s-t-graphs to solve the case in which the graph has a single source.

[3] The chapter is co-authored by Jacob Holm and Mikkel Thorup.

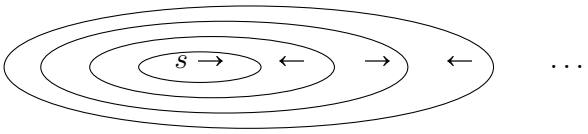

Figure 1.3: Thorup's reduction to in-out graphs via layers [158].

1.5.2 Contributions

The solution for single source graphs using truncated s-t-graphs is most involved. Denote the unique source by s. The idea is to carefully choose a face f of the graph, and consider the backwards closure $\mathrm{bc}(f)$ of f, which is the union of all paths from s to vertices of f. This partitions the plane into regions corresponding to the faces of $\mathrm{bc}(f)$. We now observe that these regions are completely independent. Any path from a vertex v residing in one region to a vertex residing in another would cross $\mathrm{bc}(f)$, and thus, since v is reachable from s, v would itself belong to $\mathrm{bc}(f)$. This can be used to build an *s-t-decomposition*; a tree where each node (denoted s-t-node) contains a subgraph of the backwards closure of some face, and which has one child for each face of that subgraph.

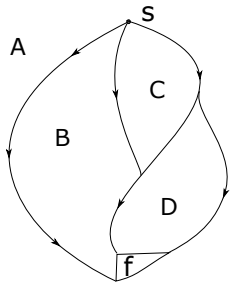

Figure 1.4: All vertices that can reach a carefully chosen face belong to this s-t-node. This partitions the graph into subgraphs: A, B, C and D. This s-t-node thus has 4 children.

We now observe that each region is separated from the rest of the graph by a *frame* consisting of two directed paths, or in other words, that it has *alteration number* 2. Recursively, we can ensure that all frames consist of 2 or 4 paths, and that the s-t-decomposition has logarithmic height. Each vertex of the graph will belong to exactly one s-t-node, and each s-t-node will constitute a truncated planar s-t-graph. For each s-t-node, we build the data structure by Tamassia and Tollis [152] in order to answer reachability

queries within the s-t-node in constant time.

Now, given the s-t-decomposition, u can reach v if and only if the are either in the same s-t-node, or if u's s-t-node $C(u)$ is an ancestor of v's s-t-node. If they are in the same s-t-node, the data structure by Tamassia and Tollis [152] answers the query. Otherwise, we consider the frame in $C(u)$ separating v from the rest of the graph (see Figure 1.6). This frame has a alteration number 2 or 4, and thus, there are at most 4 "latest vertices" on the frame that can reach v—I like to think of them as projections of v onto the s-t-graph in $C(u)$. Thus, u can reach v if and only if it can reach any of v's projections, each of which can be computed in constant time.

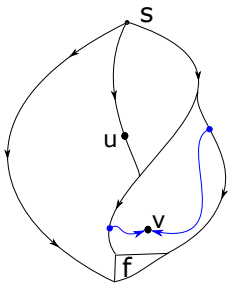

Figure 1.5: u can reach v if and only if it can reach one of its projections (blue vertices).

Computing v's projections in $C(u)$ is easy to do in $O(\log n)$ space, by simply storing all projections of each vertex. To save the final log-factor, we need to observe how frames nest inside each other, and how v's projections to its ancestral frames relate to each other. I will sketch the idea for frames of alteration number 2, called 2-*frames*, but it extends to frames of alteration number 4. For a 2-frame, v has exactly two projections, $l(v)$ (for left) and $r(v)$ (for right). Now consider the frame of v in an s-t-node x_i and its parent x_{i-1}, and denote the projections $l_i(v), r_i(v)$ in x_i and $l_{i-1}(v), r_{i-1}(v)$ in x_{i-1}. Either, the the projections of l_i and r_i in x_{i-1} are l_{i-1} and r_{i-1}, respectively (see Figure 1.6 left). Or, there is a crossing (see Figure 1.6 right), where both l_{i-1} and r_{i-1} are the projections of the same vertex in $\{l_i(v), r_i(v)\}$. Because of planarity, they cannot both "cross". Thus, to calculate v's projection in $C(u)$, we just have to find the latest "crossing" on the path to $C(u)$. To get from the crossing to the frame in $C(u)$ is easy; we note that the left and right projections each form a tree, and we use level ancestors as described by Alstrup and Holm in [9] to instantly look up the ancestors in $C(u)$.

Finally, we show how to derive a solution for planar in-out graphs from

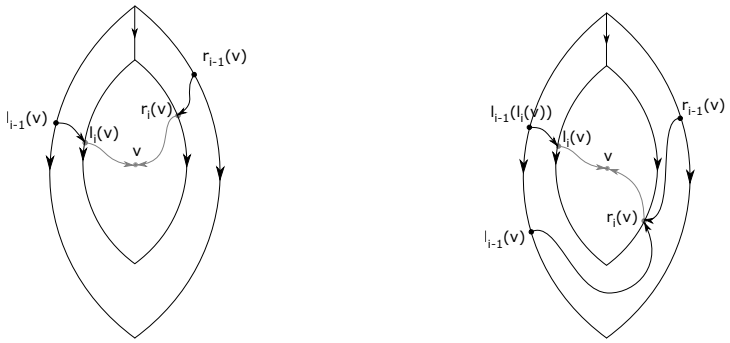

Figure 1.6: The relation between frames: either there is no crossing (left), and the projections behave like trees, or there is a crossing (right).

our structure for planar single source graphs. Given an in-out graph with a source s that can reach all targets, the basic idea is to change the direction of all edges whose tail is not reachable from s to obtain a single-source graph G', and build the reachability data structure for G'. We use the notion that vertices not reachable from s are *red vertices*, and that edges between them are red edges. We call vertices reachable from s *green vertices*, and edges between them green edges. Now, reachability between a pair of green vertices or a pair of red vertices can be answered directly by the data structure. By definition, a green vertex cannot reach a red vertex. And finally, if a red vertex can reach a green vertex, the path has to consist of first a segment of red edges, a *blue* edge, that is, one going from a red to a green vertex, and then, a segment of green edges. The red segment and the blue edge can only go towards the root in the s-t-tree (or stay in an s-t-node), the green segment can only go away from the root. We use this to extend our structure to answer all reachability queries.

1.5.3 Future directions

An obvious open problem concerns labelling schemes. For planar s-t-graphs, $O(\log n)$ bit labelling scheme exists. There is a matching lower bound of $\Omega(\log n)$, even when we restrict ourselves to graphs consisting on one long path. Thorup's planar reachability oracle also gives an $O(\log^2 n)$ bit labelling scheme. The question is whether a better labelling scheme exists, or there is an $\Omega(\log^2 n)$ lower bound?

1.6 On Chapter 3

In Chapter 3, we[4] give efficient algorithms Hamiltonicity-related results in the square G^2 of a 2-vertex connected graph G, or a graph whose block-cutpoint decomposition is a path.

Recall the definitions of 2-vertex connected, squared graph, Hamilton cycle, and block-cutpoint decomposition from Section 1.2. We give the following results for a 2-vertex connected graph G:

- A linear time algorithm for outputting a Hamilton cycle in G^2,
- Given vertices u, v, a linear time algorithm for outputting a Hamilton path in G^2 from u to v,

Furthermore, for any graph G whose block-cutpoint tree is a path,

- A linear time algorithm for outputting a Hamilton cycle in G^2,
- Given any vertex x, a quadratic time algorithm for outputting cycles in G^2 of lengths 3 to n containing x. (This property of a graph, that it contains cycles of all lengths, is also called *pancyclicity*.)
- Given a vertex x as above, a near-linear time algorithm for outputting a description of the nested vertex sets. That is, there is an $O(m + n \log^3 n \log\log^2 n)$ algorithm which outputs an ordering of the vertices, such that each suffix forms the vertex set of a cycle in G^2.

In general, deciding whether a the square of a graph contains a Hamilton cycle is NP-complete. However, for a 2-vertex connected graph, it was shown by Fleischner [54] that the square always contains a Hamilton cycle.

Our algorithm for outputting a Hamilton cycle in G^2 improves the previous best, which is an $O(n^2)$ algorithm by Lau [121], and expands on ideas in Georgakopoulos' short proof of Fleischner's theorem [62].

1.6.1 Techniques

The main result is the linear time algorithm for outputting a Hamilton cycle in G^2 for a biconnected graph G. From this, the second, third and fourth result follow from known reductions.

The proof that a Hamilton cycle can be found in linear time goes via an ear-decomposition of a minimally 2-vertex connected spanning subgraph

[4] The chapter is co-authored with Stephen Alstrup, Agelos Georgakopoulos, and Carsten Thomassen

(which can be found in linear time [74, 148]). We then prove that since the graph was minimally 2-vertex connected, that is, the removal of any edge would violate 2-vertex connectivity, each ear contains a vertex of degree 2. We then go on to double or delete carefully chosen edges to construct a connected graph where all vertices have even degree (called an *Eulerian graph*). Such a graph has an Euler tour T which can be found in linear time [81]. Finally, we replace carefully chosen subpaths of T of length two by edges of G^2 (we call this *lifting*), in a manner that achieves a Hamilton cycle in G^2. In order to perform these careful choices, we inflict an orientation upon the edges, whose sole purpose is to guide our search for a useful Euler tour and for useful 2-paths to lift.

The proof that we can in near-linear time order the vertices such that every suffix of size $\geqslant 3$ is the vertex set of a cycle in G^2 goes via cut-vertex detection in a decremental graph. We expand on the fully dynamic 2-vertex connectivity structure by Holm, de Lichtenberg, and Thorup [84], Thorup [157], and Holm, Rotenberg, and Thorup [90], which has an update time of $O(\log^3 n \log\log^2 n)$ per edge deletion, and an even faster query time. This structure is used as a subroutine in an algorithm that deletes all edges of the graph and makes a constant number of queries per deleted edge. Since we can start by finding a sparse 2-vertex connected subgraph in linear time, the total running time is $O(m + n\log^3 n \log\log^2 n)$.

1.6.2 Future directions

Our algorithm for outputting cycles of all lengths appears to be optimal, since the sum of the lengths of the cycles is $O(n^2)$. However, the vertex sets can be chosen such that they are nested, and thus, it is not unthinkable that the difference between two consecutive cycles has a short description, or at least that the cycles can be chosen in a clever way such that they have a linear description which possibly could be found in linear or near-linear time.

1.7 On Chapter 4: Dynamic Bridge-Finding in $\tilde{O}(\log^2 n)$ Amortized Time

In Chapter 4, we[5] give a deterministic data structure that supports bridge detection in a fully dynamic graph. The data structure supports insertions and deletions of edges in $O(\log^2 n \log\log^2 n)$ amortized time, and uses $O(m+n)$ space. Given vertices u, v, it detects whether a bridge separating u from

[5] The chapter is co-authored with Jacob Holm and Mikkel Thorup

v exists, and in the affirmative case, outputs it, in $O(\log n/\log\log n)$ worst-case time. Similarly, given a vertex v, it outputs a bridge in the connected component of v in $O(\log n/\log\log n)$ worst-case time if one exists.

This time bound is particularly interesting because it only differs by $\log\log n$ factors from the current best deterministic data structure for fully dynamic connectivity by Wulff-Nilsen [164], which has the same query time, and an update time of $O(\log^2 n/\log\log n)$.

The techniques can be applied to give the same improvement to the 2-vertex connectivity structure from [84, 157] to achieve an update time of $O(\log^3\log\log^2 n)$, and a query time of $O(\log n/\log\log n)$.

1.7.1 Techniques

The original data structure by Holm, de Lichtenberg, and Thorup [83] had an update time of $O(\log^4 n)$. This was since improved by Thorup [157] using approximate counting to obtain an update time of $O(\log^3 n\log\log n)$. We describe a different combinatorial improvement to get down to $O(\log^3\log\log n)$ update time, which again can be combined with approximate counting to get down to $O(\log^2 n\log\log^2 n)$ update time.

Our combinatorial improvement basically consists of a tweak in what and how information is maintained in the clusters of the top trees in the original data structure by Holm et al. Originally, each cluster keeps track of $O(\log^2 n)$ sized table, where each entry is the size of some subgraph. We show that maintaining the differences between sizes and calculating the size on demand saves an $O(\log n/\log\log n)$ factor in both time and space.

To obtain a query time of $O(\log n/\log\log n)$, the data structure uses an auxiliary top tree of degree $O(\log n/\log\log n)$. That is, a top tree where each non-leaf cluster is the union of the up to $O(\log n/\log\log n)$ clusters that are its children.

We further show that space consumption can be brought down to linear: Each cluster maintains enough information to allow us to calculate the sizes of certain subgraphs efficiently, but when a cluster is small enough, within the given time bounds, we can afford to compute that information from scratch.

1.7.2 Future directions

We have shown that the current best deterministic fully dynamic data structures for connectivity and two-edge connectivity differ only by $O(\log\log^3 n)$ in the amortized update time. This can be seen as an indication that the data

structure for deterministic fully dynamic connectivity might be improvable. Another indication is that using randomization one can in fact get down to $\widetilde{O}(\log n)$ amortized update time for dynamic connectivity. On the other hand, it could also be taken as an indication that deterministic fully dynamic 2-vertex connectivity should also have an update time of $\widetilde{O}(\log^2 n)$.

Another interesting open question is whether the techniques from Kapron, Mountjoy and King [104] can be used to build a data structure for two-edge connectivity and bridge detection with polylogarithmic worst-case update and query time using randomisation.

Finally, a very interesting question is that of fully dynamic minimum spanning forest. The bridges of the graph, which all belong to any spanning forest, are exactly those edges we can delete without having to look for a replacement edge. Can any of the ideas and techniques presented in our paper be used to improve significantly on the fully dynamic minimum spanning forest problem?

1.8 On Chapter 5: Decremental SPQR-trees for planar graphs

In this chapter, we[6] give a data structure that can maintain an SPQR-tree of each 2-vertex connected component of a planar graph subject to deletions and contractions of edges. Utilizing the SPQR-trees, we maintain local 2- and 3-vertex connectivity (see section 1.2 for a formal definition). It has an update time of amortized $O(\log^2 n)$ and constant query time.

The previous best data structure for 3-vertex connectivity has an update time of $O(\sqrt{n})$. None of the previous decremental data structures for maintaining 2-vertex connectivity or 3-vertex connectivity simultaneously support both the deletion and the contraction of edges.

1.8.1 Techniques

Crucial to maintaining SPQR-trees is the task of detecting separation pairs.

We show that for a planar embedded graph G, there exists a graph $G^\diamond$ (the face-vertex graph of G) with bounded face-degree, such that detecting separation pairs in G subject to deletions and contractions of edges corresponds to detecting non-facebounding 4-cycles in $G^\diamond$ subject to insertions and contractions of edges. The size of $G^\diamond$ is linear in the size of G, and $G^\diamond$

[6] The chapter is co-authored with Jacob Holm, Guiseppe F. Italiano, Adam Karczmarz, and Jakub Łącki

can be embedded in the plane. In fact, we only need to find the edges that participate in any non-facebounding 4-cycle, not the 4-cycles themselves.

Based on recursive use of separators of planar graphs, we build a tree of $O(\log n)$ height, and in each non-leaf node, we detect non-facebounding 4-cycles that cross the separator, whereas the leaves have a constant size and we detect 4-cycles by brute-force. In an internal node, the detection of 4-cycles that cross the separator boils down to detecting length-two paths with endpoints on the separator. Here, we use planarity to see that there cannot be many vertices that each have many neighbours on the separator, an insight which facilitates an amortised analysis in which we classify vertices according to how well connected they are to the separator.

If the deletion or contraction of an edge in an R-node (see Definition 1.2) of G gives rise to at least one non-facebounding 4-cycle in $G^\diamond$, we use the edges that participate in any non-facebounding 4-cycle to guide a search that singles out each new R-nodes of G, and finds the separation pairs that separate them. We use that the deletion of one edge can only split the R-node into a graph whose SPQR-tree is a path (see Figure 1.7).

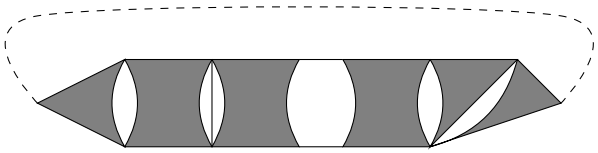

Figure 1.7: The deletion of an edge (dashed) causes an R-node to fall apart into a subgraph whose SPQR-tree is a path.

It is a well-known fact that contractions in G correspond to deletions in the dual G^*. We use this to show that we can support contractions in the almost exact same manner as deletions.

Finally, whether a pair of vertices are locally 3-vertex connected can be determined from the SPQR-tree. One can associate a constant number of words with each vertex, and with each SPQR-node, and check a constant number of cases in order to answer whether a pair of vertices are locally 3-vertex connected. Through the course of deletions and contractions, for each vertex or SPQR-node, the associated information changes at most $O(\log n)$ times.

1.8.2 Future directions

While fully dynamic planarity testing and fully dynamic connectivity and higher connectivity have $\Omega(\log n)$ cell-probe lower bounds by Pătraşcu, no

super-constant lower bound is known for decremental 3-vertex connectivity in planar graphs. On the contrary, for connectivity [125], and for 2-edge connectivity [86], amortized $O(1)$ upper bounds are known. Thus, we possibly have not only one but two unnecessary log-factors. Indeed, we sloppily build several data structures from scratch upon the detection of a separation pair. If we could in any way reuse most of the data structure already built, this could potentially save us a log-factor. However, this would be highly non-trivial and would require new theoretical insights.

Another future direction relates to higher edge-connectivity. We present a data structure that detects 4-cycles, and we use it to detect 4-cycles in the face-vertex graphs corresponding to separation pairs in the original graph. If instead we detect 4-cycles in the dual graph, those correspond to 4-edge cuts in the original graph, and can be used to maintain the cactus graph of the 5-edge connected components of a 4-edge connected graph. Similar to the SPQR-tree for 3-vertex connectivity, this cactus graph could possibly be used to implement a data structure for 5-edge connectivity in a decremental 4-edge connected planar graph.

While SPQR-trees and 3-vertex connectivity are particularly motivated for planar graphs, they are defined for general graphs as well. Curiously, there exist fully dynamic data structures for connectivity and 2-vertex connectivity with poly-logarithmic time per operation, but no non-trivial data structure for 3-vertex connectivity exists. The only known lower bound for 3-vertex connectivity is $\Omega(\log n)$ by Pătraşcu. This represents a huge gap, and any step towards closing it would be interesting.

Finally, SPQR-trees can be seen as a compact representation of all planar embeddings of a 2-connected planar graph [128]. Namely, whenever we have a separation pair, we may "flip" or "reflect" a subgraph which is isolated from the rest of the graph by that separation pair. We have shown a data structure that maintains an embedding of a planar graph subject to such flips. Is it possible to use those two results as building blocks in a data structure for maintaining a planar embedding of a fully dynamic graph if and only if such an embedding exists? If such a data structure with poly-logarithmic update time exists, it would be a major result, and it would require non-trivial new theoretical insights.

1.9 On Chapter 6: Online bipartite matching with amortized $O(\log^2 n)$ replacements

In Chapter 6, we[7] study the problems relating to minimising replacements and reassignments in on-line bipartite graphs, while at all times maintaining a maximum matching or an optimal assignment. All results are analyses of the greedy approach of simply augmenting a shortest path from the newly arrived vertex, denoted the *SAP-protocol*.

We give the following results, in the case where n clients arrive with all their incident edges:

- For bipartite matching, the SAP-protocol makes a total of $O(n \log^2 n)$ replacements.
- The SAP-protocol can be implemented in total time $O(m\sqrt{n}\sqrt{\log n})$.
- For the capacitated assignment problem, in which each server is initiated with a fixed capacity, SAP makes a total of $O(n \log^2 n)$ reassignments.

For the problem minimising maximum load, let $\textsc{opt}(G)$ denote the minimum possible maximum load of the final graph G. We show:

- SAP makes a total of $O(n \cdot \textsc{opt}(G) \log^2 n) \cap O(n\sqrt{n} \log n)$ reassignments, while maintaining an optimal assignment,
- For $L \leqslant \sqrt{n/2}$, and L divisible by 4, there exists an instance with maximum load L which requires $\Omega(nL)$ reassignments. This lower bound holds even when the algorithm knows the instance in advance.

Our main result, that SAP makes a total of $O(n \log^2 n)$ replacements, is a huge improvement over previous work. The previous best guaranteed total number of replacements was $O(n\sqrt{n})$, and, specifically for SAP, nothing better than $O(n^2)$ was known for general bipartite graphs.

Note also that our lower bound for the exact capacitated assignment problem stands in stark contrast to the approximate version, in which only $O(1)$ amortized reassignments per clients is needed to obtain a constant factor approximation to the optimally balanced load [19, 70].

[7] The chapter is co-authored with Aaron Bernstein and Jacob Holm

1.9.1 Techniques

Our main idea for showing our main result is to keep track of the necessity $\alpha(s)$ of each server s. The necessity is independent of the matching, and, in fact, all of our results hold even if we allow an adversary to make arbitrary changes to the matching between each update. We start out with $\alpha(s) = 0$, and if at some point a subset of clients $|C|$ have a neighbourhood $N(C)$ of size $|C|$, then $\alpha(s) = 1$ for all servers $s \in N(C)$, reflecting that all these servers are necessary in a maximum matching (see figure 1.8). We give a

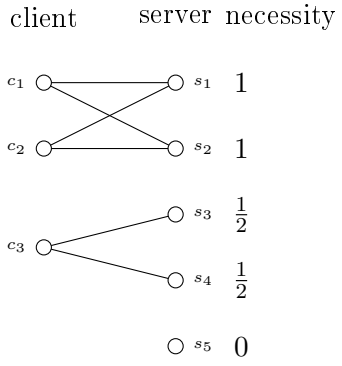

Figure 1.8: Servers s_1 and s_2 are completely necessary, because they are crucial for matching c_1 and c_2. Servers s_3 and s_4 each have a necessity of a half because of c_3, and server s_5 is not necessary at all.

definition of necessities α such that:

- Server necessity is only increasing, and once a server has necessity 1, we never can disregard it for the rest of the algorithm,
- If a client arrives and the shortest augmenting path is long, then augmenting along a shortest path will increase the necessity of many servers which already had high necessities. It follows that this can only happen a limited number of times before the involved servers have necessity 1 and can be disregarded.

Following the above indicated line of reasoning, our main lemma is, that for any h, the SAP protocol augments down a total of at most $4n \ln n/h$ paths of length $> h$. It is easy to see how the main theorem follows from the lemma: Consider the statement when h has a value of the form 2^i. For each $i = 0, 1, \ldots \lg_2(n)$, an augmenting of length between 2^i and 2^{i+1} is accounted for by $h = 2^i$, but contains at most 2^{i+1} edges. But then, summing up, we

get $8n \ln n(\lg_2 n + 1)$ which is $O(n \log^2 n)$, as an upper bound for the total length of all augmenting paths through the course of the algorithm.

The capacitated assignment problem where each server has a fixed integral capacity for how many clients it can serve is easily reduced to bipartite matchings, by simply adding multiple copies of each server—as many copies as is the capacity of the server. We thus still get $O(n \log^2 n)$ reassignments for inserting n vertices in the capacitated assignment problem.

Minimizing maximum load The upper bound of $O(L \log^2 n)$ is very intuitive and easy to prove: Let the i'th *epoch* be the time where the maximum load is i. Then it follows from our result on capacitated assignments for capacity i that during the i'th epoch, at most $O(n \log^2 n)$ reassignments are made. Thus, we get a total of $O(nL \log^2 n)$ reassignments. To show the upper bound of $O(n\sqrt{n} \log n)$, notice that we make $O(n\sqrt{n} \log n)$ reassignments during the first $\sqrt{n}/\log n$ epochs (we call these epochs *early*). If $L \leqslant \sqrt{n}/\log n$, we are done. Otherwise, all future epochs, maximally loaded servers have $\sqrt{n}/\log n$ clients assigned. But then, there are at most $\sqrt{n} \log n$ such servers, and shortest augmenting paths involving them will have length $O(\sqrt{n} \log n)$. Thus, each vertex in the late epochs causes $O(\sqrt{n} \log n)$ reassignments, and since the early epoch vertices cause amortized $O(\sqrt{n} \log n)$ reassignments, we have shown the upper bound.

The lower bound is more involved, and is proven by explicitly constructing an instance with $n = L^2$ vertices and final load L, that requires $\Omega(L^3)$ changes. The instance consists of $O(L)$ epochs, each inserting $O(L)$ clients, and each requiring $\Omega(L^2)$ reassignments in order to ensure that the maximum load is minimised by the end of the epoch.

1.9.2 Future directions

We show that the strategy of augmenting along a shortest path (the SAP-protocol) causes at most amortized $O(\log^2 n)$ replacements, while the lower bound is amortized $\Omega(\log n)$ replacements, even for graphs consisting of disjoint paths, by Grove, Kao, Krishnan, and Vitter [66]. Recently, Bosek, Leniowski, Zych, and Sankowski [24] have shown an upper bound of amortized $O(\log n)$ replacements for trees. Is it possible to show that SAP causes amortized $\Theta(\log n)$ replacements for general graphs? Or is there a lower bound separating general bipartite graphs from trees?

We give an implementation of SAP that has $O(m\sqrt{n}\sqrt{\log n})$ total running time. This almost matches the $O(m\sqrt{n})$ algorithm by Hopcroft and

Karp [92], except for a superfluous $O(\sqrt{\log n})$ factor. It would be interesting to improve our implementation by an $O(\sqrt{\log n})$ factor.

Finally, we show that amortized $\Omega(\sqrt{n})$ replacements can be necessary to maintain an exact minimal maximum load. In almost any practical setting, this would be a decisive argument for only maintaining an approximation of the minimal maximum load, such as the 8-approximation that is obtainable with $O(1)$ reassignments as described in [70] and [19]. Thus, a fundamental question is: How well can we approximate the minimal maximum load, while allowing only a constant or polylogarithmic number of reassignments per insertion?

1.10 On other related results obtained during my PhD-studies

This section contains a high level overview of selected results that have not been included in the dissertation. The results all relate to algorithms and data structures for graphs, and have been published in journals or conference proceedings.

1.10.1 On Dynamic Planar Embeddings of Dynamic Graphs [88]

We[8] give a data structure that maintains a planar embedding of a graph. The data structure supports edge-deletions, and the insertion of an edge across a face. The data structure supports changes to the embedding by flipping a subcomponent that is connected to the rest of the graph by at most 2 vertices (see Figure 1.9). Given a pair of vertices, it facilitates queries to their linkability, that is, whether they lie on the same face of the embedding. For non-linkable vertices, it facilitates the one-flip-linkable query, which points to a flip that makes them linkable if such a flip exists. All updates and queries are supported in $O(\log^2 n)$ time, where n is the number of vertices in the graph.

Techniques We maintain a tree/cotree decomposition for the planar embedded graph, which is a well known technique of partitioning the edges into a spanning tree for the graph and a spanning tree for its dual. We maintain top trees over both the tree (the *primal top-tree*) and the cotree (the *dual*

[8] The article is co-authored with Jacob Holm

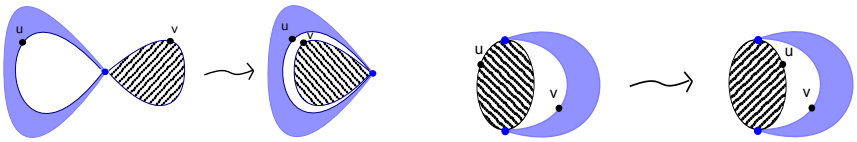

Figure 1.9: We support the flip of a subgraph which is connected to the rest of the graph in an articulation point, or by a separation pair.

top-tree), and they interact via their *extended Euler tour*, which corresponds to a carefully chosen Euler tour in the medial graph $(G^\diamond)^*$.

To facilitate linkability queries, we use an expose operation in the primal top-tree to mark all corners incident to either vertex, and then, use the dual top-tree to search for a common face. In order to have the information in the dual top-tree reflect that some vertex is "marked", we perform a constant number of top-tree operations in the dual tree for each merge or split in the primal top tree, leading to an $O(\log^2 n)$ running time.

To facilitate one-flip-linkable for a pair of not-linkable vertices u and v, we use that a flip exists exactly when there are two faces f_u, f_v in G^* such that u is incident to f_u and v is incident to f_v and such that u^* and v^* are internal faces in different separation classes in G^* (see Definition 1.1). Once we have found such faces f_u and f_v, we can use linkable query in G^* to obtain the vertices f_u and f_v have in common in which we can flip.

Future directions We show how to facilitate one-flip-linkable, but as soon as two flips are needed to accommodate an edge, we are at a loss. If the embedding is not carefully chosen, up to $\Theta(n)$ flips can be needed to accommodate an edge. Is it possible to efficiently find such a sequence of flips, if it exists? Is it possible to carefully choose the embedding such that only $O(\log^c n)$ flips are needed, if the edge can be added without violating planarity? These questions all relate to the major open problem of whether it is possible to maintain whether a dynamic graph is planar in polylogarithmic time per operation.

1.10.2 On Faster Minimum Spanning Forests [91]

We[9] give a data structure for fully-dynamic minimum spanning forests (MSF). For a dynamic graph on n vertices, we support updates in expected amortized $O(\log^4 n/\log\log n)$ in the word-RAM model with constant

[9]The article is co-authored with Jacob Holm and Christian Wulff-Nilsen

time multiplication. This is an $O(\log \log n)$ improvement over the previous $O(\log^4 n)$ algorithm by Holm, de Lichtenberg, and Thorup [84].

Our time bound relies on linear time sorting [12] of a poly-logarithmic (in n) sized set of numbers in a quadraticly (in n) bounded range, and this is the only place in the analysis where randomisation and constant time multiplication are used. In fact, even deterministically and without constant time multiplication, we give a data structure with a running time of $O(\log^4 \sqrt{\log \log \log n}/\sqrt{\log \log n})$, by simply using a different sorting algorithm as a subroutine [142].

Techniques.

We give a new analysis of the reduction by Henzinger and King [78] which was also used by Holm et al. [84], which we combine with a new decremental MSF data structure with slightly faster construction time but slightly slower update time, to obtain the result.

Namely, given a data structure for decremental MSF with construction time $O(t_c m)$ and amortized deletion time $O(t_d)$ per edge deletion, the reduction gives a data structure for fully-dynamic MSF with $O(\log^3 n + t_c \log^2 n + t_r \log n)$ update time amortized over $\Omega(n)$ updates.

Our decremental MSF with these properties is based on the dynamic connectivity structure by Wulff-Nilsen [164], which can easily be modified to give a decremental MSF with $O(\log^2 n)$ update time. After an edge deletion, when searching for a replacement edge, the algorithm makes several downwards searches but only $O(1)$ upwards searches in a tree of height $O(\log n)$. The main idea is to improve the downwards-searching time to $O(\log n / \log \log n)$ at the cost of increasing the upwards-searching time by an $O(\log^\varepsilon)$-factor. We implement this idea by making a short-cutting system for downwards searches that involves fast priority queues, which explains why the running time for sorting algorithms appear in the time bounds.

Future directions

We improve the update time for MSF from $O(\log^4 n)$ to $O(\log^4 n / \log \log n)$, but no lower bound beyond $\Omega(\log n)$ by Pătraşcu [141] is known. It is likely that some entirely new ideas and insights would lead to a data structure with $O(\log^3 n)$ update time, or even faster.

On the other hand, giving bounds of the form $\Omega(\log^2 n)$ for any connectivity-related dynamic graph problem would be very interesting and require new techniques and insights.

1.10.3 On One-way Trail Orientations [1]

We[10] prove the following extension of Robbin's theorem on strong orientations of graphs: Given a 2-edge connected planar graph, and given a partitioning of its edges into trails, we show that each trail can be oriented consistently such that the resulting graph is strongly connected. We show that such an orientation can be found in linear time.

Here, a trail is any walk that does not use the same edge twice. A consistent orientation of a trail is one where each vertex that is not an end-vertex has the same number of in- and out-edges on the trail.

We extend our result to mixed graphs in which some trails are already oriented. We show that as long as no cut has exactly one undirected edge, there exists a strong orientation that orients the remaining trails consistently.

Techniques

We give a constructive induction proof which runs in polynomial time: Take an arbitrary end-edge of an arbitrary trail. If its removal does not violate 2-edge connectivity, this edge can be disregarded, since any orientation on the rest of its trail can be extended to involve it. Otherwise, it were part of some two-edge cut, and we modify either side of the cut by inserting an edge which we concatenate with at most 2 trails, obtaining two smaller instances of the problem, which by induction can be assumed to have a solution. We show how to piece solutions to those instances together, to obtain a solution to the original problem.

This algorithm has an $O(m \log^2 n \log\log^2 n)$ implementation using the bridge detection structure in Chapter 4, [90].

To get down to linear time, we need to handle many end-edges in one swoop. The idea is to carefully choose a subgraph that contains as few end-edges as necessary (using [74]), show that this graph has many 3-edge connected components, meaning most of them have at most constant size, and show that the cactus graph over the 3-edge connected components can be used to give a strong, trail-consistent orientation of the entire graph. We thus show how to spend linear time on reducing away a constant fraction of the edges, which by a geometric argument leads to a linear algorithm.

[10] The paper is co-authored with Anders Aamand, Niklas Hjuler, and Jacob Holm

Future directions

In the mixed case, where a some edges are already oriented, and the remaining are partitioned into paths, we have not yet found a linear time algorithm. Is it possible to give a linear time algorithm, or is there a lower bound?

1.10.4 On Contracting Planar Graphs Efficiently [86]

We[11] present a data structure for maintaining a planar graph subject to edge contractions. It supports adjacency queries and facilitates access to neighbour lists in $O(1)$ time, has linear construction time, and has amortized update time $O(1)$ per edge contraction.

Using this data structure for the dual graph, we obtain a several decremental data structures for a planar graph on n vertices. In the following, update times are amortized, and query times are worst-case.

- For local 2-edge connectivity and bridge detection, we present a data structure with $O(1)$ time per operation. As a consequence, determining whether a planar graph has a maximal matching can be done in $O(n)$ time.
- For local 2-vertex connectivity and 3-edge connectivity, we show that they can be supported in $O(\log n)$ time per deletion and $O(1)$ query time.

Techniques

To obtain an efficient structure for edge contractions, we use a known result for planar graphs called r-divisions: The graph can in linear time be decomposed into the union of regions, which are subgraphs of size $\leqslant r$, such that only $O(n/r)$ vertices, called boundary vertices, participate in more than one region [65] (see Figure 1.10). The idea is to divide the responsibility of the edges. Edges between boundary vertices (red edges in Figure 1.10) are handled in the top level, and edges with at least one non-boundary endpoint (blue edges in Figure 1.10) are handled within a region. Thus, we can afford to spend polylogarithmic time per update on the top-level. To handle edges within a region, we perform the trick of subdivision again, obtaining regions of size $O((\log \log n)^4)$. These regions are so small that any possible sequence of contractions can in linear time be precomputed and stored. Finally, we

[11] The paper is co-authored with Jacob Holm, Guiseppe F. Italiano, Adam Karczmarz, Jakub Łącki, and Piotr Sankowski

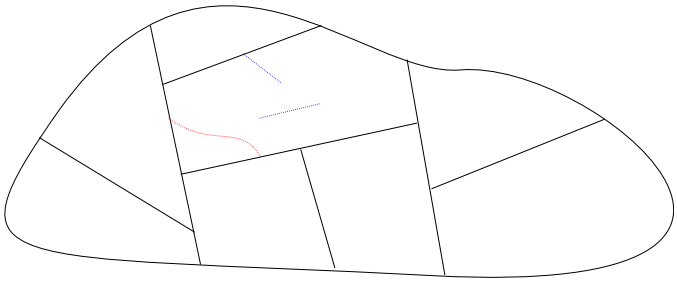

Figure 1.10: An r-division of a planar graph. The red edge goes between boundary vertices, while the blue edges are internal in their region.

note that because the graph is planar, the total number of times a contraction within a region leads to the insertion of an edge between boundary vertices in the layer above, is only linear in the number of boundary vertices.

Future directions

We have shown how to support deletions in amortized $O(1)$ time for 2-edge connectivity, and amortized $O(\log n)$ time for 2-vertex connectivity and 3-edge connectivity. One obvious direction is to get the update time for the latter down from $O(\log n)$ to $O(1)$. Another regards contractions: Is it possible to support both deletions and contractions in the planar graph in constant time? We have seen that for 2- and 3-vertex connectivity, an update time of $O(\log^2 n)$ is possible, but no lower bound beyond $\Omega(1)$ is known.

Chapter 2

Planar Reachability in Linear Space and Constant Time

Jacob Holm, Eva Rotenberg, Mikkel Thorup[1]

Abstract

We show how to represent a planar digraph in linear space so that reachability queries can be answered in constant 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].

2.1 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 v 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 [160] proved that there are graphs classes such that any representation of reachability needs $\Omega(m)$ bits. Also, Pătraşcu [138] has proved that there are directed graphs with $O(n)$

[1]The content of this chapter is published at FoCS'15 [89]

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 [158] 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 [110]. A word is a unit of space big enough to fit any vertex identifier, so a word has $w \geqslant \lg n$ bits, and word operations take constant time. Here $\lg = \log_2$. In our upper bounds, we limit ourselves to the *practical RAM* model [132], which is a restriction of the word RAM to the standard operations on words available in C that are AC^0. 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 [138] for general graphs is in the cell-probe model subsuming the word RAM with an arbitrary instruction set.

Other related work Before [158], 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 [16, 30, 43]. For such planar distance oracles, the best current time-space trade-off is $\tilde{O}(n/\sqrt{s})$

time for any $s \in [n, n^2]$ [133].

The construction of [158] also yields approximate distance oracles for planar digraphs. With edge weights from $[N]$, $N \leqslant 2^w$, 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, [112, 158] 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 [107] 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 [108] 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 $\bar{O}$ 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 [153] have shown that we can represent reachability in linear space, answering reachability queries in constant time. Also, [30,44,45] present 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 can reach t. [153]

2. Acyclic planar single-source graph; $\exists s$, such that all vertices are reachable from s. See Section 2.3.

3. Acyclic planar In-Out graph; $\exists s$ such that all vertices with out-degree 0 are reachable from s. See Section 2.4

4. Any acyclic planar graph. The reduction to acyclic planar In-Out graphs from general acyclic planar graphs is known. [158]

5. Any planar graph. The reduction to acyclic planar graphs is well-known. Using the depth first search algorithm by Tarjan [154], 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 [158], we use separators to form a tree over a partitioning of the vertices of the graph. However, in [158], the *alternation number*; the number of directed segments in the frame that separates a child from its parent (see Section 2.2), 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 [158]. These two features let us use a level ancestor -like algorithm to quickly calculate the best $\leqslant 4$ 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 $\leqslant 4$ vertices.

2.2 Preliminaries

For a vertex v at depth d in a rooted forest T and an integer $0 \leqslant i \leqslant d$, 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 $G^* \backslash H$, where G^* is the dual graph of G. 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 S the total ordering stemming from reachability of vertices on the path via the path, and by convention we set $\text{succ}(t, S) = \perp$ for a target vertex t on the disegment. Given a set of edges $S \subseteq E$, we denote by $\text{init}(S)$ the set of inital vertices, $\text{init}(S) = \{u | (u, v) \in S\}$. Given a connected planar graph with a spanning tree T, the edges $T^* := E \backslash T$ 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 acyclic planar s-t-graph, all faces has alternation number 2 (see [152, Lemma 1]).

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 rooted tree with truncated s-t-graphs as nodes is obtained by recursively choosing a face f wisely, 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 2.3.1, 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 2.3.3. Frames with alternation number 4 are similar but more involved, and the details are found in Section 2.3.4.

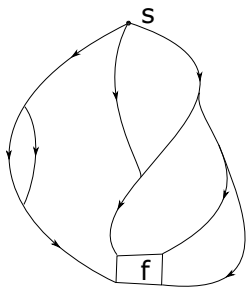

Figure 2.1: In a single source graph, the backwards closure of the face f is the union of all paths from the source, s, to $V[F]$.

Definition 2.1. 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 2.2. 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. (See Figure 2.1.)

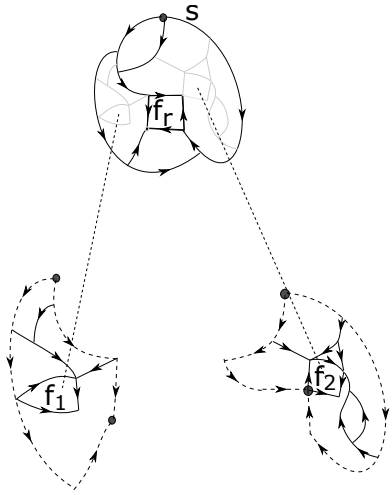

Figure 2.2: A tree of truncated s-t-graphs, each child contained in a face-cycle of its parent.

Definition 2.3. 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 $\mathrm{bc}(f_x) \backslash S_y$ if x is a child of y.
- G_x^* is the subgraph of G^* induced by $\{f_z |\ z \text{ is a descendant of } x\}$. Furthermore, if x is a child of y we require that G_x^* is the connected component of $G^* \backslash E^*[S_y]$ 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.

That is, G_x^* is the set of faces contained in the parent frame F_x.
The name s-t-decomposition is chosen based on the following

Lemma 2.1. *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 target t' in f_x and edges (v, t') for $v \in V[f_x]$ 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 $C_x = \mathrm{bc}(f_x) \backslash S_y = S_x \backslash S_y = (S_x/S_y) \backslash \{s'\}$ 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 $v \in S_N \backslash S_y = C_N$. 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 2.1. *Any acyclic single-source plane digraph has a good s-t-decomposition.*

We defer the proof to section 2.3.1. The reason for studying s-t-decompositions in the context of reachability is the following

Lemma 2.2. *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 of y such that any $u \rightsquigarrow v$ path contains a vertex in F_z.*

Proof. Note that in general, whenever $w \rightsquigarrow w'$ with $w' \in C_a$, w must belong to an ancestor of a, since $w \in bc(f_a)$. Thus, in our case, x is an ancestor of y, which means that either $x = y$ or x has a child z that is an ancestor of 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 2.1) 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 2.3.3 we will show how to do this efficiently when F_z is *a 2-frame*, that is, has alternation number 2, and in section 2.3.4 we will extend this to the case when F_z is *a 4-frame*, that is, has alternation number 4.

Theorem 2.2. *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 2.1) and can be built in linear time (Lemma 2.9). Adding DFS pre- and postorder numbers to each node in the tree lets us discover the ancestry relationship between any two vertices in the same node, in constant time.

To answer reachable(u, v), there are the following cases. Let $u \in C_x$ and $v \in C_y$ be the st-nodes of the st-decomposition where u and v belong, and let $\preceq$ denote ancestry of st-nodes in the st-decomposition.

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 [153].

3. If $x \prec y$ and there are no 2-frames separating u and v, but, since $x \prec y$, there are 4-frames. Then, by Theorem 2.5 we can in constant time compute the at most 4 best vertices in C_x that can reach v. If u can reach any of them, them u can reach v, otherwise no.

4. Otherwise, $x \prec y$ and there is a 2-frame F_z separating u and v such that there are no 2-frames between x and z. Then, by Theorem 2.4 we can in constant time compute the at most 2 best vertices in C_z that can reach 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 2, and similarly, the recursive calls in step 4 only leads to questions of type 3 or 2. Thus, we use only constant time per query. □

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

Theorem 2.3. *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.

2.3.1 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 each vertex of $G \backslash H$ lies in a unique face of H. We may choose a tree-cotree decomposition 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 2.4).

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 2.3. *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 $T^* \backslash E^*[S]$.

Proof. Let S^* be the dual of S, then $S^* = G^*/(G^* \backslash E^*[S])$ and the claim is equivalent to saying that the components of $G^* \backslash E^*[S]$ correspond to the components of $T^* \backslash E^*[S]$. Consider a pair of faces $f_1, f_2 \in V^*$. Clearly, if they are in separate components of $G^* \backslash E^*[S]$, they are also in separate components in $T^* \backslash E^*[S]$. On the other hand, suppose f_1 and f_2 are in different components in $T^* \backslash E^*[S]$. 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 2.4. *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 2.3. □

Lemma 2.5. *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:*

$$A^* \subseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 4.$$
$$A^* \nsubseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 2.$$

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 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_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. □

The following lemma will help us ensure that we can choose all frames to be 2- or 4-frames.

Lemma 2.6. *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:*

$$A^{0^*} \nsubseteq V^*[T_y^*] \wedge A^{1^*} \nsubseteq V^*[T_y^*] \implies F_y \text{ has alternation number } 2.$$
$$A^{0^*} \nsubseteq V^*[T_y^*] \vee A^{1^*} \nsubseteq V^*[T_y^*] \implies F_y \text{ has alternation number } \leqslant 4.$$

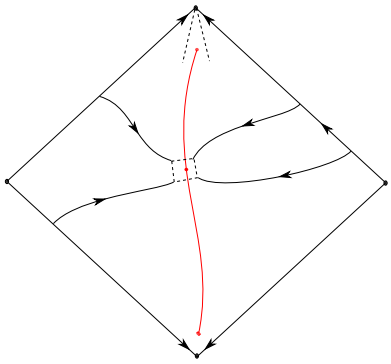

Figure 2.3: Choosing a splitting-face on the co-path between faces near either target of a 4-frames yields only 4-frames and 2-frames on the next level.

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 $A^{i*} \nsubseteq V^*[T_y^*]$ 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. □

We can now prove that a good s-t-decomposition exists.

proof of theorem 2.1. 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 2.4 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), $|T_z^*| \leqslant \frac{1}{2}|T_x^*|$.

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, $|T_y^*| \leqslant \frac{1}{2}|T_x^*|$, 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, $|T_y^*| \leqslant \frac{1}{2}|T_x^*|$, and, by Lemma 2.5, 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^* (see Figure 2.3). By Lemma 2.6 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 $|T_y^*| \leqslant \frac{1}{2}|T_x^*|$.
- If $f_x \neq m$ and T_y^* is not the component of m in $T_x^* \backslash E^*[\mathrm{bc}(f_x)]$, then $|T_y^*| \leqslant \frac{1}{2}|T_x^*|$.
- If $f_x \neq m$, and T_y^* is the component of m in $T_x^* \backslash E^*[\mathrm{bc}(f_x)]$, then T_y^* contains neither f_0 nor f_1, so by Lemma 2.6 the parent frame F_y has alternation number at most 2 and we have just shown this means any child z of y has $|T_z^*| \leqslant \frac{1}{2}|T_y^*| \leqslant \frac{1}{2}|T_x^*|$. □

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 2.4 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 [11] 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 $\varnothing$ if x is the root), and a face in T_x^* incident to each target in the set.

Build a top tree (see [10]) 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 [76] 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 2.7. *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 2.5, since we chose a splitting face on the cotree path between faces near the two targets. □

Lemma 2.8. *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 $f_x \neq v_i^*$ which is the projection of v_i^* on $f_1 \dots f_2$. As noted in Lemma 2.5, 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 2.9. *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 2.7 and 2.8, the resulting s-t-decomposition is good. □

2.3.3 2-frames

Given a graph G with vertices V and edges E, we have shown how to construct an st-decomposition $\mathcal{T}$ in linear time. An st-decomposition is a tree with st-nodes and st-edges. Each non-root st-node, x, has a frame F_x, which will either be a 2-frame or a 4-frame. We say that the st-node has a 2- or a 4-frame.

Since we want to reason about the relationship between 2-frames, disregarding 4-frames, we construct a 2-*frame-decomposition* from the st-decomposition. The 2-frame-decomposition is a tree whose nodes are contracted subtrees of the st-decomposition. Specifically, each st-node with a 4-frame is contracted with its nearest ancestor with a 2-frame.

Definition 2.4. 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 st-edge (z, y) in $\mathcal{T}$ where the child z of y has 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 $E_x := E_{\text{lca}(Y)}$.

Recall, E_x are the edges with their tail on the frame F_x. If the frame is a 2-frame, it consists of a clockwise and a counterclockwise disegment. The embedding of G gives a natural cyclic order to E_x, and we can partition the edges of E_x into two contiguous subsets in that cyclic order, such that all tails in one subset are on the clockwise disegment of F_x and all tails in the other subset are on the counterclockwise disegment of F_x. We notice that there exists a partitioning of all edges whose tail is on a 2-frame, into sets $\mathcal{R}$ and $\mathcal{L}$ such that for *any* E_x, the sets $E_x \cap \mathcal{L}$ and $E_x \cap \mathcal{R}$ form such a partition.

Definition 2.5. 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 $\mathcal{L}$.

Lemma 2.10. *Let $(\mathcal{L}, \mathcal{R})$ be the partition from Definition 2.5. Then for any $x \in \mathcal{T}_2$, $(E_x \cap \mathcal{L}, E_x \cap \mathcal{R})$ is a partition of E_x, such that all tails in $E_x \cap \mathcal{L}$ are on the clockwise disegment of F_x, and all tails in $E_x \cap \mathcal{R}$ are on the counterclockwise disegment of F_x.*

Proof. Let $x \in \mathcal{T}_2$, let $(u, v) \in E_x$. If u is the source or the target of F_x, it is on both the disegments, and we are done. Suppose therefore that u is

neither, and thus, lies on exactly one disegment of F_x. Let $y \in \mathcal{T}_2$ be the st-node closest to the root such that $(u, v) \in E_y$, and such that u is not the target vertex of E_y, as in the definition. Since u is not the source vertex of F_x, u cannot be the source vertex of F_y. That means, u lies on exactly one disegment of F_y.

Assume for contradiction that u lies on the clockwise disegment of F_x but the counterclockwise disegment of F_y. Then, there must be some st-node x' on the path between y and x in $\mathcal{T}_2$ such that u lies on the clockwise disegment of x' and on the counterclockwise disegment of x''s parent, z.

First note that u cannot be the target vertex of F_z, since then $x = x' = y$. Also, u cannot be the source of neither F_z or $F_{x'}$, as then u would be the source of F_x. So u lies only on the counterclockwise disegment of F_z. Then, the source vertex $s_{x'}$ of $F_{x'}$ belongs to $bc(f_z)$, and any non-trivial path from $s_{x'}$ to u would belong to C_z, and thus to u's frame at that level, causing (u, v) to belong to the clockwise disegment. That means if u belongs to the counterclockwise disegment of F_z, we must have $s_{x'} = u$, a contradiction. □

Each vertex belongs to some node of $\mathcal{T}_2$, and thus, we can define the depth of the vertex in the $\mathcal{T}_2$-tree:

Definition 2.6. Let $\mathcal{T}_2$ be a 2-frame-decomposition of $G = (V, E)$. For any vertex $v \in V$ define:

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

Given a vertex, v, and given a frame F that is ancestral to v in the st-decomposition, each disegment on F contains a last vertex that can reach v via an ingoing edge. Thus, all other vertices on the frame can reach v via an ingoing edge if and only if they can reach v via one of those last vertices (see Figure 2.4). For 2-frames, such vertices will be called $l_i(v)$ and $r_i(v)$ and are defined as follows:

Definition 2.7. For any $0 \leqslant i < d_2[v]$, let x be the ancestor of $c_2[v]$ at

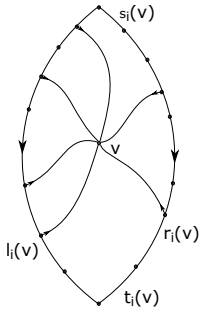

Figure 2.4: The best two vertices that can reach v on level i.

depth $i+1$ and define:

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

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 $0 \leqslant i < d_2[v]$. The main idea that *almost* works is to represent each function with a suitable rooted forest and use a level ancestor structure on that forest to answer queries.

Definition 2.8. For any vertex $v \in V$, define the left parent $p_l[v]$ and right parent $p_r[v]$ as

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

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

Using these trees we can define functions $l'_i(v)$ and $r'_i(v)$ (related but not always equal to $l_i(v)$ and $r_i(v)$), as the nearest ancestor to v in T_l or T_r with depth $\leqslant i$. We will later show how to use a level ancestor structure to compute these efficiently, but for our proofs it is more convenient to define them recursively as follows.

Definition 2.9. For any $v \in V \cup \{\perp\}$, and $i \geqslant 0$ let

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

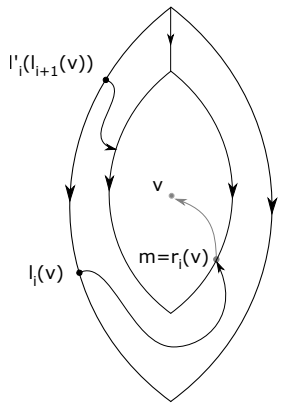

Figure 2.5: The best path from $L_i(v)$ to v goes via $r_{i+1}(v)$.

As mentioned, we do not always have $l_i(v) = l'_i(v)$ and $r_i(v) = r'_i(v)$. When F_y is the parent frame of F_x in $\mathcal{T}_2$, there are two cases for the best

vertices on F_y in relation to the best vertices of F_x, call them l, r. Either the left parent of l and the right parent of r are the best vertices on F_y, or there is a *crossing*, to one or the other side, e.g. when the left parent of r is later than the left parent of l on their disegment of F_y (see Figure 2.5).

Lemma 2.11 (Crossing lemma)**.** *Let $v \in V$, and $0 \leqslant i < d_2[v] - 1$.*

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

The proof needs som additional technical lemmas, and is deferred to the end of this section. What the lemma says is that when there is a crossing at level i, then there exists some *crossing vertex* $m \in \{r_{i+1}(v), l_{i+1}(v)\}$ such that the left- and right-parents of m are the best vertices that can reach v on level i. We define $m_i(v)$ as either the crossing vertex at level i, if there is a crossing, or the nearest crossing vertex on a descendent level, otherwise.

Definition 2.10. Let $v \in V$ and $0 \leqslant i < d_2[v]$.

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

Corollary 2.1. *Let $v \in V$ and $0 \leqslant i < d_2[v] - 1$. If $l_i(v) \neq l'_i(l_{i+1}(v))$ or $r_i(v) \neq r'_i(r_{i+1}(v))$ then*

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

Proof. This is just a reformulation of Lemma 2.11 in terms of $m_i(v)$. □

Given this definition of $m_i(v)$, one may always find the best vertices that can reach v, $l_i(v)$ and $r_i(v)$ as the left- and right-ancestors of $m_i(v)$.

Lemma 2.12. *For any vertex $v \in V$ and $0 \leqslant i < d_2[v]$*

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

Proof at the end of this section.

To calculate $l_i(v)$ and $r_i(v)$ quickly for some given i, the idea is to store for each vertex a bit array of whether there is a crossing at a given level, then to calculate the crossing vertex for that level, and then find the left- and right-ancestors for that crossing vertex. To represent the function $m_i(v)$, we again use a suitable rooted forest, and use a level ancestor structure on that forest to answer queries.

Definition 2.11. For any vertex $v \in V$, let

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

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

Theorem 2.4. *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 level ancestor 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 $d_2[m_i(v)] = \operatorname{succ}(M[v] \cup \{d_2[v]\}, i)$ in constant time, as well as computing the depth in T_m of $m_i(v)$. Then, using the level ancestor 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 level ancestor structure, this 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 2.12 says we can compute $l_i(v)$ and $r_i(v)$ in constant time given constant-time functions for l', r', and m. □

Technical lemmas and proofs

To prove Lemmas 2.11 and 2.12, we use some technical lemmas.

Lemma 2.13. *For any $u, v \in V$ and $0 \leqslant i < d_2[u]$: 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 $\widehat{R}_i(u) \subseteq \widehat{R}_i(v)$. □

When an edge (w, w') that lies on a path from s to v skips several levels, that is, w' lies on a level possibly much higher than that of w, then (w, w') belongs to either $\hat{L}$ or $\hat{R}$ of v for all those levels:

Lemma 2.14. *Given any vertex $v \in V$, $0 \leqslant i < d_2[v]$, and $(w, w') \in E_i(v)$. Then:*

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

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

The vertices that are left and right level ancestors $l'_i(v)$ and $r'_i(v)$ from Definition 2.9 are tails of edges of $\hat{L}_i(v)$ and $\hat{R}_i(v)$:

Lemma 2.15. *Let $v \in V$, and $i \geqslant 0$ be given, then*

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

Proof. We will show this for l' only, as r' is completely symmetrical. If $i > d_2[v] - 1$ then $d_2[v] \leqslant i$ and we get $l'_i(v) = v$ directly from the definition of l'. Similarly if $i = d_2[v] - 1$ then $l'_i(v) = l'_i(p_l[v]) = l'_i(l_{d_2[v]-1}(v)) = l'_i(l_i(v)) = l_i(v) \in \operatorname{init}(\widehat{L}_i(v)) \cup \{\perp\}$. 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 $d_2[w] \leqslant i < d_2[w'] \leqslant d_2[u]$ and by setting $(v, i, (w, w')) = (u, d_2[u] - 1, (w, w'))$ in Lemma 2.14 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 2.13 and we are done. □

We prove the following two intuitive lemmas: The level ancestor of a level ancestor of v is again the level ancestor of v, and if $l_i(v) = \perp$, then the i'th level ancestor of $l_{i+1}(v)$ is also $\perp$ (similar for $r_i(v)$).

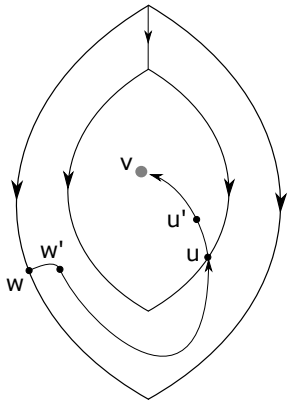

Figure 2.6: If $l'_i(l_{i+1}) \neq l_i$, then any path from l_i to v must go through $R_{i+1}(v)$.

Lemma 2.16. *Let $v \in V$ and $0 \leqslant i \leqslant j$ 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 recursion. The case for r' is symmetric. □

Lemma 2.17. *Let $v \in V$, and $0 \leqslant i < d_2[v] - 1$, then*

$$\begin{aligned} l_i(v) = \perp \quad &\Longrightarrow \quad l'_i(l_{i+1}(v)) = \perp \\ r_i(v) = \perp \quad &\Longrightarrow \quad r'_i(r_{i+1}(v)) = \perp \end{aligned}$$

Proof. If $l_i(v) = \perp$ then $\widehat{L}_i(v) = \varnothing$, 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 2.15 and Lemma 2.13 $l'_i(l_{i+1}(v)) \in \text{init}(\widehat{L}_i(l_{i+1}(v))) \cup \{\bot\} \subseteq \text{init}(\widehat{L}_i(v)) \cup \{\bot\} = \{\bot\}$ so again $l'_i(l_{i+1}(v)) = \bot$. The case for r is symmetric. □

We may now prove Lemma 2.11 (Crossing Lemma).

Proof of Lemma 2.11 (Crossing Lemma). 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). Let $m = r_{i+1}(v)$. We then aim to show, first, $d_2[m] = i+1$, and secondly, $l_i(v) = l'_i(m)$ and $r_i(v) = r'_i(m)$. If $l_i(v) \neq l'_i(l_{i+1}(v))$, then $l_i(v) \neq \bot$ by lemma 2.17. Thus, there is a last edge $(w, w') \in \widehat{L}_i(v)$ with $w = l_i(v)$ and $d_2[w] \leqslant i < d_2[w']$ and a path $P = w' \rightsquigarrow v$.

Now, $(w, w') \notin E_{i+1}(v)$, since otherwise by Definition 2.7 $(w, w') \in L_{i+1}(v)$ 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 2.15, contradicting our assumption.

Since $(w, w') \notin E_{i+1}(v)$, the path P must cross $\widehat{F}_{i+1}(v)$. (See Figure 2.6.) Let (u, u') be the unique edge in $P \cap \widehat{F}_{i+1}(v)$. Then, $w' \rightsquigarrow u$ so $d_2[u] \geqslant d_2[w'] \geqslant i+1$ and $(u, u') \notin L_{i+1}(v)$ since otherwise $d_2[l_{i+1}(v)] = i+1$ and hence by Lemma 2.15 $l_i(v) = l'_i(l_{i+1}(v))$, again contradicting our assumption. Since $\widehat{F}_{i+1}(v) \neq \varnothing$, we therefore have $(u, u') \in \widehat{R}_{i+1}(v)$. But then we can choose P so it goes through (m, m') where $m = r_{i+1}(v) \neq \bot$. Now $i+1 \leqslant d_2[w'] \leqslant d_2[r_{i+1}(v)] \leqslant i+1$ so $d_2[m] = i+1$.

Finally, 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 2.15. □

We may now prove the essential Lemma stating that $l_i(v) = l'_i(m_i(v))$ (and similar for r_i):

Proof of Lemma 2.12. 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 2.15, 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 2.1 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 2.16, $l'_i(l_{i+1}(v)) = l'_i(l'_{i+1}(m_{i+1}(v))) = l'_i(m_{i+1}(v)) = l'_i(m_i(v))$, showing that $l_i(v) = l'_i(m_i(v))$ as desired. The case for r is symmetric. □

2.3.4 4-frames

Given a vertex and an ancestral frame, F, we have now seen how to find the at most two best vertices on F that can reach v when F a 2-frame. A similar statement is true for 4-frames, only now we have four disegments, and thus, up to four best vertices. To find the best vertices, we may use Theorem 2.4 as a subroutine, and thus we only need to get from the nearest descendent 2-frame, if it exists, to a given level. That is, we may disregard edges that cross a 2-frame.

In our construction, we exploit that whenever a 4-frame occurs, it shares at least one target with its parent frame. In particular, if its parent is again a 4-frame, they share a target vertex.

Definition 2.12. 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_x^0 and s_x^1 be the source corners on F_x and let t_x^0 and t_x^1 be the target corners on F_x, numbered such that their clockwise cyclic order on F_x is $s_x^0, t_x^0, s_x^1, t_x^1$, and such that if F_y is a 4-frame there is an $\alpha \in \{0,1\}$ so $t_x^\alpha = t_y^\alpha$.

Recall from 2-frames that we found a global partition of all the down-edges whose tail is on a 2-frame into two sets $\mathcal{L}$ and $\mathcal{R}$. It turns out we can partition the remaining down-edges (those that are only in 4-frames) into four sets $\mathcal{L}^0$, $\mathcal{R}^0$, $\mathcal{L}^1$, and $\mathcal{R}^1$, such that for edges that do not cross a 2-frame, the partitioning corresponds to the four disegments.

Definition 2.13. Let $\mathcal{E}_4$ be the set of down-edges (u,v) such that for all st-nodes x where $(u,v) \in E_x$, 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 $(u,v) \in E_y$ and 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 $\mathcal{L}^\alpha$.

Lemma 2.18. *Let $(\mathcal{L}^0, \mathcal{R}^0, \mathcal{L}^1, \mathcal{R}^1)$ be the partition from Definition 2.13. Then, for any $x \in \mathcal{T}$, $(E_x \cap \mathcal{L}^0, E_x \cap \mathcal{R}^0, E_x \cap \mathcal{L}^1, E_x \cap \mathcal{R}^1)$ is a partition of $E_x \cap \mathcal{E}_4$, such that for $\alpha \in \{0,1\}$ all tails in $E_x \cap \mathcal{R}^\alpha$ are on the clockwise disegment of F_x between s^α and t^α, and all tails in $E_x \cap \mathcal{L}^\alpha$ are on the counterclockwise disegment of F_x between $s^{1-\alpha}$ and t^α.*

Proof. Consider an edge $(u, v) \in \mathcal{E}_4$. If u is a target vertex in all frames F_x where $(u, v) \in E_x$, then by Definitions 2.12 and 2.13 there is an $\alpha \in \{0, 1\}$ such that (u, v) in $\mathcal{L}^\alpha$ and $t_x^\alpha = u$ for all such st-nodes x. Thus, for each such x, the tail of (u, v) is on the counterclockwise disegment of F_x between $s_x^{1-\alpha}$ and t_x^α as required.

Otherwise, there is an st-node y and an $\alpha \in \{0, 1\}$, such that u is not a target of F_y, and for any ancestor z to y with $(u, v) \in E_z$, $u = t_z^\alpha$. By Definition 2.13, we then have (u, v) in either $\mathcal{L}^\alpha$ or $\mathcal{R}^\alpha$, and t_z^α is on both the required segments. Thus, the statement holds for edges incident to a target of F_x.

Finally, let z and x be st-nodes such that z is the parent of x and $(u, v) \in E_z \cap E_x$, and u is not a target of F_z. Then, u is not a target of F_x, either, and:

- If u is on the clockwise disegment between s_z^α and t_z^α, then u is on the clockwise disegment between s_x^α and t_x^α.
- If u is on the counterclockwise disegment between $s_z^{1-\alpha}$ and t_z^α, then u is on the counterclockwise disegment between $s_x^{1-\alpha}$ and t_x^α.

Thus, since u is on the correct disegment of F_y, it will also be on the correct disegment of all descendants of y, and we are done. □

Similar to the definition of $d_2[v]$; v's depth in $\mathcal{T}_2$, we may define the depth of v in the entire st-decomposition, $\mathcal{T}$. Given a vertex v belonging to some st-node $c[v]$, we let $j_2[v]$ denote the nearest ancestor to $c[v]$ whose frame is a 2-frame.

Definition 2.14. Let $\mathcal{T}$ be an st-decomposition of $G = (V, E)$. For any vertex $v \in V$ define:

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

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 $j_2[v] \leqslant i < d[v]$, we will have 4-frames nested in 4-frames, which gives a lot of useful structure.

Given a vertex v and a 4-frame F ancestral to v, we may now define the at most four best vertices that can reach v; one for each disegment of F. They will be called $l_i^0(v), r_i^0(v), l_i^1(v)$, and $r_i^1(v)$.

Definition 2.15. For any $j_2[v] \leqslant i < d[v]$ and $\alpha \in \{0,1\}$, let x be the ancestor of $c[v]$ at depth $i+1$ and define:

$$\begin{aligned}
F_i(v) &:= F_x & E_i(v) &:= E_x \cap \mathcal{E}_4 \\
L_i^\alpha(v) &:= E_x \cap \mathcal{L}^\alpha & R_i^\alpha(v) &:= E_x \cap \mathcal{R}^\alpha \\
\widehat{L}_i^\alpha(v) &:= \left\{(w,w') \in L_i^\alpha(v) \middle|\, w' \rightsquigarrow v\right\} \\
\widehat{R}_i^\alpha(v) &:= \left\{(w,w') \in R_i^\alpha(v) \middle|\, w' \rightsquigarrow v\right\} \\
\widehat{F}_i(v) &:= \widehat{L}_i^0(v) \cup \widehat{R}_i^0(v) \cup \widehat{L}_i^1(v) \cup \widehat{R}_i^1(v)
\end{aligned}$$

$$l_i^\alpha(v) := \begin{cases} \bot & \text{if } \widehat{L}_i^\alpha(v) = \varnothing \\ \text{the last vertex in } \operatorname{init}(\widehat{L}_i^\alpha(v)) \text{ on the counterclockwise dipath of } F_x & \text{otherwise} \end{cases}$$

$$r_i^\alpha(v) := \begin{cases} \bot & \text{if } \widehat{R}_i^\alpha(v) = \varnothing \\ \text{the last vertex in } \operatorname{init}(\widehat{R}_i^\alpha(v)) \text{ on the clockwise dipath of } F_x & \text{otherwise} \end{cases}$$

$$\begin{aligned}
s_i^\alpha(v) &:= \text{The vertex associated with } s_x^\alpha \\
t_i^\alpha(v) &:= \text{The vertex associated with } t_x^\alpha
\end{aligned}$$

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 2.3.3 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 $j_2[v] \leqslant i < d[v]$.

As for 2-frames, the idea is to define some suitable trees that allow us to compute some related functions, and a crossing lemma that lets us use these to compute the functions we actually want.

Definition 2.16. 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 2.17. For any $v \in V \cup \{\bot\}$, $\alpha \in \{0,1\}$, and $i \geqslant j_2[v]$ let

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

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

Lemma 2.19 (Crossing lemma)**.** *Let $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leqslant i < d[v] - 1$.*

$$l_i^\alpha(v) \neq l_i'^\alpha(l_{i+1}^\alpha(v)) \implies l_i^\alpha(v) = l_i'^\alpha(m) \wedge r_i^\alpha(v) = r_i'^\alpha(m) \wedge d[m] = i+1$$
$$\text{where } m = r_{i+1}^\alpha(v) \neq \bot$$
$$r_i^\alpha(v) \neq r_i'^\alpha(r_{i+1}^\alpha(v)) \implies l_i^\alpha(v) = l_i'^\alpha(m) \wedge r_i^\alpha(v) = r_i'^\alpha(m) \wedge d[m] = i+1$$
$$\text{where } m = l_{i+1}^\alpha(v) \neq \bot$$

This Crossing Lemma (see Figure 2.7) is the 4-frame version of Lemma 2.11. Again, the proof needs som additional technical lemmas, and is deferred to the end of this section.

Definition 2.18. Let $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leqslant i < d[v]$.

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

Corollary 2.2. *Let $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leqslant i < d[v] - 1$. 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*

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

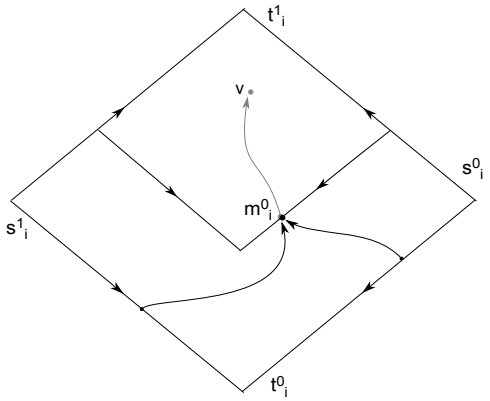

Figure 2.7: Sometimes, the best path from $L_i^0(v)$ to v must go through $R_{i+1}^0(v)$.

Proof. This is just a reformulation of Lemma 2.19 in terms of $m_i^\alpha(v)$. □

Lemma 2.20. *For any vertex $v \in V$, $\alpha \in \{0,1\}$, and $j_2[v] \leqslant i < d[v]$*

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

Definition 2.19. For any vertex $v \in V$, and $\alpha \in \{0,1\}$ let

$$M^\alpha[v] := \left\{ i \middle|\, j_2[v] < i < d[v] \wedge m_{i-1}^\alpha(v) \neq m_i^\alpha(v) \right\}$$
$$p_m^\alpha[v] := \begin{cases} \perp & \text{if } M^\alpha[v] = \varnothing \\ m^\alpha_{\max M^\alpha[v]-1}(v) & \text{otherwise} \end{cases}$$

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

Theorem 2.5. *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 $l_i^\alpha(v)$ and $r_i^\alpha(v)$ queries in constant time.*

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

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

Now, store level ancestor structures for each of T_l^α, T_r^α, and T_m^α, together with $d[v]$, $j_2[v]$, $J_2[v]$, $D_l^\alpha[v]$, $D_r^\alpha[v]$, and $M^\alpha[v]$ for each vertex. Since the

height of the st-decomposition is $\mathcal{O}(\log n)$ each of $J_2[v]$, $D_l^\alpha[v]$, $D_r^\alpha[v]$, and $M^\alpha[v]$ can be represented in a single $\mathcal{O}(\log n)$-bit word.

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

Similarly, this representation of the $D_l^\alpha[v]$ set lets us compute the depth in T_l^α of $l_i'^\alpha(v)$ in constant time, and, with the level ancestor structure, this lets us compute $l_i'^\alpha(v)$ in constant time. A symmetric argument shows that we can compute $r_i'^\alpha(v)$ in constant time.

Finally, lemma 2.20 says we can compute $l_i^\alpha(v)$ and $r_i^\alpha(v)$ in constant time given constant-time functions for l', r', and m. □

Technical lemmas and proofs (4-frames)

To prove the Lemmas 2.19 and 2.20, we prove some technical lemmas.

Lemma 2.21. *For any vertex $v \in V$ and $j_2[v] \leqslant i < d[v]$: $\widehat{F}_i(v) \neq \varnothing$*

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

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

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

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

Proof. Let $j = \max\{d[w], j_2[v]\}$ and $k = \min\{d[w'], d[v]\}$. Clearly, $(w,w') \in E_{i'}(v)$ for all $j \leqslant i' < k$. Suppose $(w,w') \in \widehat{L}_i^\alpha(v) \subseteq L_i^\alpha(v)$, then, since $j \leqslant i < k$, the definition gives us $(w,w') \in L_{i'}^\alpha(v)$ for all $j \leqslant i' < k$.

And since $w' \rightsquigarrow v$, this implies $(w, w') \in \widehat{L}^{\alpha}_{i'}(v)$ for all $j \leqslant i' < k$ and the result follows. The case for R is symmetric. □

Lemma 2.24. *Let $v \in V$, $\alpha \in \{0, 1\}$, and $i \geqslant j_2[v]$ be given, then*

$$\begin{array}{lll} i = d[v] - 1 & \Longrightarrow & l'^{\alpha}_i(v) = l^{\alpha}_i(v) \quad \wedge \quad r'^{\alpha}_i(v) = r^{\alpha}_i(v) \\ i \leqslant d[v] - 1 & \Longrightarrow & l'^{\alpha}_i(v) \in \mathrm{init}(\widehat{L}^{\alpha}_i(v)) \cup \{\bot\} \quad \wedge \quad r'^{\alpha}_i(v) \in \mathrm{init}(\widehat{R}^{\alpha}_i(v)) \cup \{\bot\} \\ i > d[v] - 1 & \Longrightarrow & l'^{\alpha}_i(v) = v \quad \wedge \quad r'^{\alpha}_i(v) = v \end{array}$$

Proof. We will show this for l' only, as r' is completely symmetrical. If $i > d[v] - 1$ then $d[v] \leqslant i$ and we get $l'^{\alpha}_i(v) = v$ directly from the definition of l'. Similarly, if $i = d[v] - 1$, then $l'^{\alpha}_i(v) = l'^{\alpha}_i(p^{\alpha}_l[v]) = l'^{\alpha}_i(l^{\alpha}_{d[v]-1}(v)) = l'^{\alpha}_i(l^{\alpha}_i(v)) = l^{\alpha}_i(v) \in \mathrm{init}(\widehat{L}^{\alpha}_i(v)) \cup \{\bot\}$. Finally, suppose $i < d[v] - 1$. If $l'^{\alpha}_i(v) = \bot$ 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 $d[w] \leqslant i < d[w'] \leqslant d[u]$, and by setting $(v, i, (w, w')) = (u, d[u] - 1, (w, w'))$ in Lemma 2.23, we get $(w, w') \in \widehat{L}^{\alpha}_i(u)$, and therefore $l'^{\alpha}_i(v) \in \mathrm{init}(\widehat{L}^{\alpha}_i(u))$. But since $u \rightsquigarrow v$ we have $\widehat{L}^{\alpha}_i(u) \subseteq \widehat{L}^{\alpha}_i(v)$ by Lemma 2.22 and we are done. □

Lemma 2.25. *Let $v \in V$, $\alpha \in \{0, 1\}$, and $j_2[v] \leqslant i \leqslant j$ 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 2.26. *Let $v \in V$, $\alpha \in \{0, 1\}$, and $j_2[v] \leqslant i < d[v] - 1$, then*

$$\begin{array}{rcl} l^{\alpha}_i(v) = \bot & \Longrightarrow & l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \bot \\ r^{\alpha}_i(v) = \bot & \Longrightarrow & r'^{\alpha}_i(r^{\alpha}_{i+1}(v)) = \bot \end{array}$$

Proof. If $l^{\alpha}_i(v) = \bot$ then $\widehat{L}^{\alpha}_i(v) = \varnothing$, so either $l^{\alpha}_{i+1}(v) = \bot$ implying $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \bot$ by the definition of l', or $l^{\alpha}_{i+1}(v) \notin \mathrm{init}(\widehat{L}^{\alpha}_i(v))$ so $d[l^{\alpha}_{i+1}(v)] = i+1$ and by Lemma 2.24 $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) \in \mathrm{init}(\widehat{L}^{\alpha}_i(l^{\alpha}_{i+1}(v))) \cup \{\bot\} \subseteq \mathrm{init}(\widehat{L}^{\alpha}_i(v)) \cup \{\bot\} = \{\bot\}$ so again $l'^{\alpha}_i(l^{\alpha}_{i+1}(v)) = \bot$. The case for r is symmetric. □

We may now prove Lemma 2.19 (Crossing Lemma).

Proof of Lemma 2.19 (Crossing Lemma). Suppose $l_i^\alpha(v) \neq l_i'^\alpha(l_{i+1}^\alpha(v))$ (the case $r_i^\alpha(v) \neq r_i'^\alpha(r_{i+1}^\alpha(v))$ is symmetrical). Then, $l_i^\alpha(v) \neq \perp$ by lemma 2.26. Thus, there is a last edge $(w, w') \in \widehat{L}_i^\alpha(v)$ with $w = l_i^\alpha(v)$ and $d[w] \leqslant i < d[w']$ and a path $P = w' \rightsquigarrow v$.

Now, $(w, w') \notin E_{i+1}(v)$, since otherwise by Definition 2.15 $(w, w') \in L_{i+1}^\alpha(v)$, and since $w' \rightsquigarrow v$, even $(w, w') \in \widehat{L}_{i+1}^\alpha(v)$, implying $l_i^\alpha(v) = l_{i+1}^\alpha(v)$, and thus, $l_i^\alpha(v) = l_i'^\alpha(l_{i+1}^\alpha(v))$ by Lemma 2.24, 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 unique edge in $P \cap \widehat{F}_{i+1}(v)$. Then, $w' \rightsquigarrow u$ so $d[u] \geqslant i+1$ and $(u, u') \notin L_{i+1}^\alpha(v)$ since otherwise $d[l_{i+1}^\alpha(v)] = i+1$ and hence by Lemma 2.24 $l_i^\alpha(v) = l_i'^\alpha(l_{i+1}^\alpha(v))$, again contradicting our assumption.

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

Since $t_i^\alpha(v) \neq t_{i+1}^\alpha(v)$, then, by definition, $t_i^{1-\alpha}(v) = t_{i+1}^{1-\alpha}(v)$ and hence $L_{i+1}^{1-\alpha}(v) \subseteq L_i^{1-\alpha}(v)$ and $R_{i+1}^{1-\alpha}(v) \subseteq R_i^{1-\alpha}(v)$, implying $d[w''] \leqslant i$ for all $w'' \in L_{i+1}^{1-\alpha}(v) \cup R_{i+1}^{1-\alpha}(v)$. Thus, $(u, u') \notin L_{i+1}^{1-\alpha}(v) \cup R_{i+1}^{1-\alpha}(v)$ since $d[u] > i$, and we can conclude that $(u, u') \in \widehat{R}_{i+1}^\alpha(v)$.

But then we can choose P so it goes through (m, m') where $m = r_{i+1}^\alpha(v) \neq \perp$. Now $i+1 \leqslant d[w'] \leqslant d[r_{i+1}^\alpha(v)] \leqslant i+1$ 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 2.24. □

We may now prove the essential Lemma stating that $l_i^\alpha(v) = l_i'^\alpha(m_i^\alpha(v))$ (and similar for r_i^α):

Proof of Lemma 2.20. 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 2.24, 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 2.2, that $l_i^\alpha(v) = l_i'^\alpha(m_i^\alpha(v))$ and $r_i^\alpha(v) = r_i'^\alpha(m_i^\alpha(v))$.

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 $l_{i+1}^\alpha(v) = l_{i+1}'^\alpha(m_{i+1}^\alpha(v))$ and $r_{i+1}^\alpha(v) = r_{i+1}'^\alpha(m_{i+1}^\alpha(v))$. Then, by Lemma 2.25, $l_i'^\alpha(l_{i+1}^\alpha(v)) = l_i'^\alpha(l_{i+1}'^\alpha(m_{i+1}^\alpha(v))) = l_i'^\alpha(m_{i+1}^\alpha(v)) = l_i'^\alpha(m_i^\alpha(v))$, showing that $l_i^\alpha(v) = l_i'^\alpha(m_i^\alpha(v))$ as desired. The case for r is symmetric. □

2.4 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 2.4.1.

M3 Information about paths from a red vertex in some component C to a green vertex in an ancestor component of C. See Section 2.4.2.

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".

Now, $\text{reach}_G(u, v)$ has four cases, based on the colours of u and v:

- For green u and red v, $\text{reach}_G(u, v) = \text{No}$.
- 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.

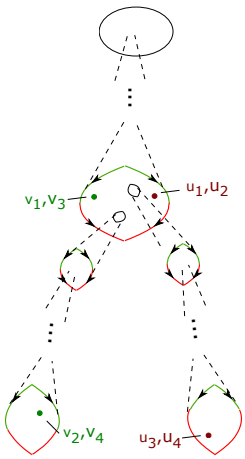

Figure 2.8: Cases 1 through 4.

When u is red and v is green, a path from u to v must consist of a (possibly trivial) red path, a blue edge, and a (possibly trivial) green path. In the st-decomposition of H, red and blue edges can only either stay in an st-node, or go towards the root. Green edges, on the contrary, stay in an st-node, or go to a descendant. There are the following cases (see Figure 2.8) for $\text{reach}_G(u, v)$, based on where in the heirarchy of components u and v are.

1. $c[u] = c[v]$. There may be a path from u to v:
   - Staying within the st-node, that is, $\text{reach}_{c[u]}(u, v)$. To handle this case we need to store more information, see Section 2.4.1.
   - Via a green vertex w in the parent frame of u. For each candidate $w \in \mathit{BestGreen}(u)$, try $\text{reach}_H(w, v)$. (See M1).
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 can calculate the at most 4 such ws from the single source structure, namely $l(v)$ and $r(v)$, or $l^\alpha(v)$ and $r^\alpha(v)$.
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 2.4.2.

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.

2.4.1 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 2.27. *We may assign to each vertex $\leqslant 2$ 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 $i \leqslant l \leqslant j$ or $i \leqslant r \leqslant j$ or $\min\{l, r\} \leqslant j < i$ or $j < i \leqslant \max\{l, r\}$.*

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. □

2.4.2 Intercomponental blue edges

For any red vertex u, for a blue edge (u', v), where u' is reachable from u and separated from u by the frame $F_i(u)$, then one of u's "best" vertices, $l_i(u), r_i(u)$ or $l_i^\alpha(u), r_i^\alpha(u)$, is a red vertex, and can reach u'. For any red vertex w, let $E_{c[w]}$ denote the edges of w's parent frame, $F_{c[w]}$. Consider the set $B(w)$ of those blue edges in $E_{c[w]}$ that are reachable from w in G (or, equvalently, can reach w in H). For each disegment of $F_{c[w]}$, there is at most one "best" edge of $B(w)$, that is, whose green head is closest to the source. Let each red vertex remember the best $\leqslant 4$ blue edges it can reach on its own frame. Then we can define 4 bitmasks $\{B^\beta(u)\}_{0\leqslant\beta\leqslant 3}$ such that for any i finding the highest 1-bit $\leqslant i$ in each, gives at most 4 levels such that u's best vertices on those levels together know the best blue edges for u.

Chapter 3

A Hamiltonian Cycle in the Square of a 2-connected Graph in Linear Time

Stephen Alstrup, Agelos Georgakopoulos, Eva Rotenberg, Carsten Thomassen

Abstract

Fleischner's theorem says that the square of every 2-connected graph contains a Hamiltonian cycle. We present a proof resulting in an $O(|E|)$ algorithm for producing a Hamiltonian cycle in the square G^2 of a 2-connected graph $G = (V, E)$. The previous best was $O(|V|^2)$ by Lau in 1980. More generally, we get an $O(|E|)$ algorithm for producing a Hamiltonian path between any two prescribed vertices, and we get an $O(|V|^2)$ algorithm for producing cycles $C_3, C_4, \ldots, C_{|V|}$ in G^2 of lengths $3, 4, \ldots, |V|$, respectively.

3.1 Introduction

Fleischner [54] proved in 1974 that the square of every 2-connected graph is Hamiltonian, solving a conjecture from 1966 by Nash-Williams. This remarkable result has stimulated much work on paths and cycles in the square of a finite graph, e.g [27], [55], [82] and [155]. Fleischner's theorem has also been extended to infinite locally finite graphs with at most two ends by

Thomassen [156], and to (compactifications of) all locally finite graphs by Georgakopoulos [61].

A short proof of Fleischner's theorem was obtained by Říha (1991) [161]. The technique in that proof has some resemblance with the technique in [156]. More recently, a simpler proof was presented by Georgakopoulos (2009) [62], see also [38].

Lau (1980) [121] was the first to give an efficient constructive algorithm, more precisely an $O(|V|^2)$ algorithm, for finding a Hamiltonian cycle in the square of a 2-connected graph.

In this paper, we present a simple proof of Fleischner's theorem based on the ideas of [62], which results in a linear time algorithm for finding a Hamiltonian cycle in the square G^2 of a 2-connected graph G.

Finding a Hamiltonian cycle in a graph is fundamental and used in many graph algorithms, but is also known to be NP-complete (as shown by Karp [105], see also Garey and Johnson [60]). Finding Hamiltonian cycle in the square of 2-connected is used explicit in the Bottleneck Travelling Salesman Problem, see Parker and Rardin [137], but can also be used to compact implicit representations of distances in general graphs (labelling schemes), see Alstrup et al. [7].

Combining this algorithm with the proof in [27] (which we reproduce) that G^2 is *Hamiltonian connected*, that is, it has a Hamiltonian path between any two prescribed vertices, we get an $O(|E|)$ algorithm for producing such a path. We also apply the algorithm to the result in [82, 155] that G^2 is *pancyclic*, that is, it has a collection of cycles $C_3, C_4, \ldots, C_{|V|}$ of lengths $3, 4, \ldots, |V|$, respectively. More precisely, we combine the algorithm with the proof in [155] (which we reproduce in the present paper as well) and obtain thereby an $O(|V|^2)$ algorithm for producing cycles of all lengths in the square of a 2-connected graph. In fact, we may, in $O(|V|^2)$ time, produce such cycles C_i with nested vertex sets, that is, $V(C_i) \subseteq V(C_{i+1})$, such that $x \in V(C_3)$ for any prescribed vertex x of a graph whose block-cutvertex tree is a path.

These results follow from our main theorem:

Theorem 3.1. *There is a linear time algorithm finding a Hamiltonian cycle in the square of any 2-connected graph.*

Proof. The algorithm comprises the following steps:

1. Find a minimally 2-connected spanning subgraph G of the 2-connected graph under consideration.

2. Find a proper ear decomposition of G.

3. Pick a vertex of degree 2 in each ear, which exists by Lemma 3.2.

4. Modify the proof of [62] such that it results in a linear time algorithm.

A linear time algorithm for step 1 above was found by Han et. al. [74, Theorem 12]. A linear time algorithm for step 2 was found by Schmidt [148]. Given an ear in our decomposition, one can check the degrees of all vertices and choose one of degree 2 for each ear, ensuring linear running time of step 3 above. Finally, we will show in Section 3.4 why step 4 runs in linear time. □

3.2 Preliminaries

A *graph* has no loops or multiple edges. In the proofs we shall double some edges resulting in a *multigraph*, and we shall also introduce orientations of edges. An edge between the vertices x, y in an undirected graph is denoted xy.

A *proper ear decomposition* of an undirected graph G is a partition of its set of edges into a sequence $C^0, \ldots, C^k$ where C^0 is a cycle and C^i is a path for every $i \geqslant 1$, such that for every $i \geqslant 1$, $C^i \cap \bigcup_{j<i} C^j$ consists of the two endvertices of C^i.

A graph is *k-connected* if no deletion of $k-1$ vertices disconnects the graph. An edge e of a graph G is *k-essential*, if $G - e$ is not k-connected. A minimally k-connected graph is one where every edge is k-essential.

An *Euler tour* (respectively *Euler walk*) of a multigraph is a walk which uses every edge exactly once, and has a last vertex which is the same (respectively not the same) as the first vertex. A multigraph has an Euler tour if and only if all components except one are isolated vertices, and every vertex of the exceptional component has even degree. An Euler tour may be found in linear time using Hierholzer's algorithm [81]. A multigraph is *Eulerian* if and only if it has an Euler tour.

The *square* of a graph $G = (V, E)$ is the graph $G^2 = (V, E')$ where $uv \in E'$ if and only if u and v are connected by a path of length at most 2 in G. The *G-degree*, or just *degree* of a vertex v in a graph G is the number of edges in G incident with v. A *chord* of a path or cycle is an edge which not in the path or cycle and which joins two vertices of the path or cycle. Note that Dirac [42, Definition 5] uses the term chord with a different meaning.

A *Hamiltonian cycle* is a cycle that contains all vertices of the graph. A graph G is *pancyclic* if it contains a cycle of length i for every $i \in$

$\{3, 4, \ldots, |V(G)|\}$. A graph is *vertex-pancyclic* if, for every vertex x, these cycles can be chosen to pass through x.

3.3 Every ear has a vertex of degree 2

Dirac [42] gave a detailed investigation of the minimally 2-connected graphs. His work inspired deep results on minimally k-connected graphs, see e.g. [119, 129]. We use the definition introduced by Dirac [42, Definition 6]: Given two vertices of a minimally 2-connected graph, they are *compatible* if no path between them has a chord. We also use Dirac's observation that every 2-connected subgraph of a minimally 2-connected graph is minimally 2-connected.

Lemma 3.1. *Let G be a minimally 2-connected graph, and let u and v be vertices of G. Suppose there are three internally disjoint paths P_1, P_2, P_3 between u and v in G. Then each of the three paths contains at least one vertex of G-degree 2.*

Proof. We claim that u and v must be compatible. To prove this claim, let P_4 be any path between u and v, and consider the union $G' = P_1 \cup P_2 \cup P_3 \cup P_4$ which is a 2-connected subgraph of G, and hence minimally 2-connected by the above observation. Assume for contradiction that e is a chord of P_4. Then e lies on at most one of P_1, P_2, P_3. But then, $G' - e$ is still 2-connected, contradicting the minimality. Now [42, Corollary 2 to Theorem 6] says that every path from u to v has a vertex of degree 2 in G. □

Lemma 3.2. *Let $C^0, \ldots, C^k$ be a proper ear decomposition of a minimally 2-connected graph G. Then every C^i contains a vertex of G-degree 2, and C^0 contains at least two vertices of G-degree 2.*

Proof. Consider a proper ear decomposition of a minimally 2-connected graph as stated. Then the union of the first i ears $C^0 \cup \ldots \cup C^{i-1}$ forms a 2-connected graph. If u, v are the endvertices of C^i, there exist two internally disjoint paths between them in $C^0 \cup \ldots \cup C^{i-1}$. Together with C^i they form three internally disjoint paths, each of which contains a vertex of G-degree 2 by Lemma 3.1. Thus, C^i contains a vertex of G-degree 2.

For C^0 we have a stronger statement. If $k = 0$, then all vertices of $C^0 = G$ are of G-degree 2. Otherwise, let u, v be the endvertices of C^1. Then there are two internally disjoint paths in C^0 between u, v. Together with C^1 we have three internally disjoint paths, as before, and each must contain a vertex of G-degree 2. Two of these are in C^0. □

3.4 A Hamiltonian cycle in linear time

Let G be a minimally 2-connected graph. In this section, we use the ear decomposition found above in order to construct a Hamiltonian cycle in the square of G. This part of our algorithm draws heavily from the proof of [62].

Let $C^0, C^1, \ldots, C^k$ be a proper ear decomposition of G, where C^0 is a cycle and each other C^i has both its endvertices in ears with smaller indices. By Lemma 3.2, every C^i contains an interior vertex y^i of G-degree 2, and it is easy to pick such a vertex for each i in linear time. Furthermore, by Lemma 3.2, C^0 contains two vertices of G-degree 2, say x and y^0.

We enumerate the vertices of each C^i as $x_0^i, x_1^i, \ldots, x_{\ell_i}^i$ in the order they appear on C^i, starting with the endvertex lying in the ear with the smallest index. For C^0 we just start the enumeration at $x_0^0 = x = x_{\ell_0}^0$.

We start our procedure by turning G into an Eulerian multigraph $G_{\Q}$ by adding parallel edges to some existing edges of G and deleting some edges of G as follows.

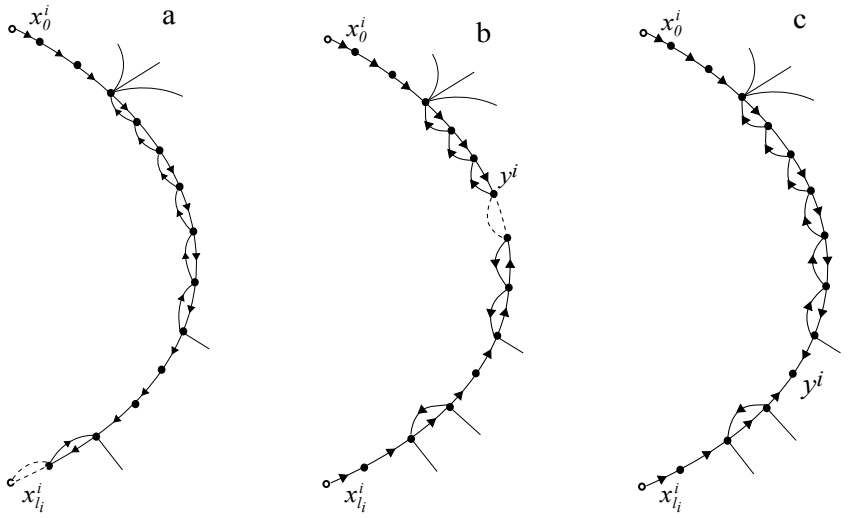

Figure 3.1: We go through the ears in decreasing order, double some edges and delete at most one (dotted line), thus, turning the graph into an Eulerian graph.

For each $i = k, k-1, \ldots, 0$, we define the graph $G_i = C^k \cup C^{k-1} \cup \ldots C^i$. For $i = k, k-1, \ldots, 0$, we define the multigraph G_i^+ as follows. First we put $G_k^+ = G_k = C^k$. Suppose we have defined G_{i+1}^+. We now traverse the vertices and edges of $C^i - x_0^i$, starting with x_1^i. When we traverse the

edge $e = x^i_j x^i_{j+1}$ we either delete e or add an edge e^+ parallel with e, or we leave e unchanged. Suppose we have traversed $x^i_1 x^i_2 \dots x^i_j$. If the degree $d(x^i_j)$ is odd in the current graph, we introduce a new edge e^+ parallel to $e = x^i_j x^i_{j+1}$ so that $d(x^i_j)$ becomes even. If the last edge $e = x^i_{\ell_{i-1}} x^i_{\ell_i}$ of C^i is doubled by this procedure, we delete both of e, e^+ (see Figure 3.1a). If the last edge is not doubled, but one (and hence both) of the edges incident with y^i is doubled, then we delete the pair of parallel edges incident with y^i which succeedes y^i as we move along C^i from x^i_0 (see Figure 3.1b). Figure 3.1c shows the situation where possibly some edges of $C^i - x^i_0$ are doubled but none are deleted. The procedure terminates when we have defined G^+_0. Clearly, this procedure for defining G^+_0 has a linear running time.

We claim that every vertex v distinct from x has even G^+_0-degree. To see this we first observe that there is a smallest i such that v is in C_i. Then v is interior in C_i (as v is distinct from x). The degree of v is made even when we form G^+_i, and the degree of v remains even after that because v is not contained in any C_j with $j < i$. More precisely, the G^+_0-degree of v equals the G^+_i-degree of v. The only vertex we never consider in this procedure is x. But x, too, has even G^+_0-degree by the handshaking lemma. Moreover, G^+_0 is connected because every vertex of C^i is still connected to $C^0 \cup C^1 \cup \dots \cup C^{i-1}$ after any edge deletion, because at most one edge of C^i is deleted. Thus G^+_0 is Eulerian. We denote G^+_0 by $G_\emptyset$.

Next, we orient the edges of $G_\emptyset$ as follows. We orient any pair e, e^+ of parallel edges in $G_\emptyset$ in opposite directions. We go through the ears C^i of G again (in fact, this step of our algorithm can be combined with the previous step). If the last edge of C^i has been deleted, we orient all edges of $C^i \cap G_\emptyset$ (that is, the edges that have not been doubled) from x^i_0 towards $x^i_{\ell_i - 1}$ (see Figure 3.1a). Otherwise, we orient all edges of $C^i \cap G_\emptyset$ (that have not been doubled) towards y^i (see Figure 3.1b and c).

Note that after we are done, though vertices may have arbitrarily high out-degree, every vertex v has at most 2 incoming edges, since only edges of the ear C^i containing v as an interior vertex can be directed towards v by construction; here we used the fact that the first edge of each C^i is never doubled, and if the last one is doubled it is immediately deleted. Moreover, if $v \neq x$, then v has at least 1 incoming edge.

We now describe an Euler tour J of the underlying undirected graph $G_\emptyset$ such that for every vertex v having two incoming edges vw, vz, these edges are consecutive in J. This can easily be achieved by first replacing these two edges vw, vz by a single wz edge for every vertex v having two incoming edges, then finding an Euler tour in the resulting auxiliary graph

G^-, and finally replacing the new edge wz by the pair wv, vz for every v as above. Note that v becomes an isolated vertex if v has $G_\emptyset$ -degree 2. The fact that G^- has only vertices of even degree follows immediately from the construction of $G_\emptyset$. It only remains to be proved that G^- consists of a connected graph and possibly a set of isolated vertices. When the ear is as Figure 3.1c, its special vertex y^i may become an isolated vertex. But, all other vertices in the ear are still part of the same connected component. More precisely, for each vertex $v \in C^i$ $(v \neq y^i)$, if it has indegree 2, it has at least one outneighbour on C^i, and thus, the deletion of its two incoming edges does not disconnect v from C^i.

Finally, we transform the Euler tour J into a Hamiltonian cycle , which we call H, by replacing some pairs of edges of $G_\emptyset$ by single edges of G^2. More precisely, when we traverse the Euler tour J, we replace every 2-edge subwalk $u \leftarrow w \rightarrow v$ in J by the single edge uv. We make a single exception in that we keep the unique subwalk having x as its middle vertex as it is.

Note that this operation is well-defined, as whenever we have the subwalk $u \leftarrow w \rightarrow v$, the edges in question are incoming to both u and v. So an edge cannot be part of two such subwalks.

We claim that H is indeed a Hamiltonian cycle of G^2. Clearly, H traverses x precisely once, because J traverses x once, and we keep the two edges incident with x when we form H. For every vertex $w \neq x$, the number of times that H traverses w equals the number of subwalks uwv in J containing an incoming edge of w. We claim that there is exactly one such subwalk. This is clear if w has indegree 1. As each vertex $w \neq x$ has indegree either 1 or 2 in $G_\emptyset$, we consider the case where w has indegree 2. By the construction of J, the two edges entering w form a subwalk of J, and those two edges are part of H. All other pairs of edges incident with w are replaced by single edges when we transform J into H.

3.5 A Hamiltonian path in linear time

Given vertices u, v of a 2-connected graph G, it was shown by Chartrand, Hobbs, Jung, Kapoor, and Nash-Williams [27] that G^2 contains a Hamiltonian path from u to v. We shall now describe an efficient algorithm to find it.

Theorem 3.2. *There exists a linear time algorithm for finding a Hamiltonian path between any two prescribed vertices u, v in the square of a 2-connected graph G.*

Proof. We use the following trick from [27]. Take the union of five disjoint copies of G. Add two new vertices x, y. Let x be joined to the five copies of u, and let y be joined to the five copies of v. The resulting graph H is 2-connected, and therefore our algorithm can produce, in linear time, a Hamiltonian cycle C in H^2. One of the five copies of G does not contain a neighbor of x, y in C. The intersection of that copy with C is a Hamiltonian path from u to v in G^2. □

3.6 Cycles of all lengths in quadratic time

Hobbs [82] proved that the square of a 2-connected graph is pancyclic. Thomassen [155] proved the same under the weaker assumption that the block-cutvertex tree is a path. If G is a graph, then its *block-cutvertex tree* is the tree whose vertices are the blocks and cutvertices of G. There is an edge between a block and a cutvertex if and only if the cutvertex is contained in the block. The block-cutvertex tree can be found in linear time [154].

The first part of the following result was first proven in [155].

Lemma 3.3. *If G is a graph whose block-cutvertex tree is a path, then G^2 has a Hamiltonian cycle. Moreover, there exists a linear time algorithm for finding a Hamiltonian cycle in G^2.*

Proof. If G is 2-connected we use Theorem 3.1. So assume that G is not 2-connected. We let u, v be two non-cutvertices in distinct end-blocks of G. We now use the following trick from [155]. Take the union of four disjoint copies G_1, G_2, G_3, G_4 of G. Add two new vertices x, y. Let x be joined to the two copies of u in G_1, G_2, and let y be joined to the two copies of v in G_3, G_4. Let the copy of v in G_1 (respectively G_2) be joined to the copy of u in G_3 (respectively G_4). The resulting graph H is 2-connected, and therefore our algorithm can produce, in linear time, a Hamiltonian path P between x, y in H^2. As proved in [14], the intersection of P with one of the four copies of G gives rise to a Hamiltonian cycle in G^2. □

Theorem 3.3. *There exists an $O(n^2)$ algorithm for producing cycles $C_3, C_4, \ldots, C_n$ of lengths $3, 4, \ldots n$, respectively in the square of a graph G on n vertices whose block-cutvertex tree is a path. Moreover, if x_0 is any vertex in G, then the cycles can be chosen such that $x_0 \in V(C_3) \subseteq V(C_4) \subseteq ... \subseteq V(C_n)$.*

Proof. Again, we use the idea in [155]. First, we may find the block-cutvertex graph in linear time using [154], and use the linear time algorithm of [74] on

each block to obtain a spanning subgraph such that every block is minimally 2-connected. Dirac [42] proved that such a graph has at most $2n-4$ edges. Then, it follows from Lemma 3.3 that we may find a Hamiltonian cycle C_n in in G^2 in linear time. If G has only one block, we delete any edge of G and use the linear algorithm in [154] to find the block-cutvertex tree which is a path. As pointed out by Dirac [42], each block is minimally 2-connected. If G is not 2-connected, we let x be a cutvertex contained in an end-block B of G. If x has degree at least 2 in B we delete any edge in B incident with x, and use the linear algorithm in [154] to find the block-cutvertex tree which is a path. Finally, if B has only two vertices x, y, then we delete y. (If $y = x_0$, we consider the other end-block of the current graph instead of B.) Then we use the algorithm in Lemma 3.3 to find a Hamiltonian cycle C_{n-1} in $(G-y)^2$. We repeat the argument.

We now discuss the complexity. We first spend $O(m)$, $m = |E|$, time obtaining a subgraph in which each block is minimally 2-connected. Then we successively delete an edge in an end-block which is incident with a cutvertex. When an isolated vertex appears, we delete that, too. Immediately after we delete an isolated vertex we find a Hamiltonian cycle in the square of the current graph. Thus there are less than $2n-4$ edge-deletions, by an aforementioned result of Dirac [42], and between two of these edge-deletions, it takes only $O(n)$ time to find a Hamiltonian cycle in the square of the current graph, by Theorem 3.1. Thus, the total time consumption is $O(m+n^2) = O(n^2)$.

□

3.6.1 Outputting the nested vertex-sets in near-linear time.

We now proceed to show that an encoding of the nested vertex-sets of the cycles can be computed in near-linear time.

Theorem 3.4. *Given an graph with n vertices and m edges, whose block-cutpoint tree is a path, and given any prescribed vertex x, there exists an $O(m + n\log^3 n \log\log^2 n)$ time algorithm which outputs the vertices of G in an order such that any suffix of size $\geqslant 3$ is connected by a cycle in G^2 containing x.*

Our main tool is a data structure for dynamic 2-connectivity in graphs by Holm et al. [84], improved from $O(\log^5 n)$ to $O(\log^4 n \log\log n)$ by Thorup [157], and later improved by Holm, Rotenberg, and Thorup [90]:

Theorem 3.5 ([84,90,157]). *There exists a deterministic fully dynamic algorithm for maintaining biconnectivity in a graph, using $O(\log^3 n \log\log^2 n)$ amortized time per operation.*

The data structure supports insertion and deletion of edges, as well as queries of the form Biconnected(u, v), which reports whether the two vertices lie in the same biconnected component.

The data structure keeps track of an underlying spanning tree of each connected component of the dynamic graph. It associates with each edge a level between 0 and $\lfloor \lg n \rfloor$, where spanning tree edges always have level $\lfloor \lg n \rfloor$. It maintains that the maximal number in vertices in a biconnected component of the subgraph G_i containing edges of level $\geqslant i$ is at most $\lceil n/2^i \rceil$. It uses a top-tree over the spanning tree, and maintains, for each path cluster, the cover-level of the path: $c^*(P)$. The cover-level is the maximal i such that the entire cluster-path P is bi-connected in G_i. In order to answer Biconnected(u, v), it simply calls expose(u, v), and returns whether the cover-level of the root cluster is $\leqslant 0$.

In order to use their data structure, however, we need it to support one more operation:

$$\text{FindFirstArticulation}(u, v),$$

which outputs the first articulation point on the spanning-tree path from u to v.

Lemma 3.4. *The data structure in Theorem 3.5 can be extended to support the operation* FindFirstArticulation(u, v) *for any pair of not biconnected vertices $u, v \in G$, in amortized $O(\log^3 n \log\log^2 n)$ time. The output is the articulation point closest to u which separates u from v.*

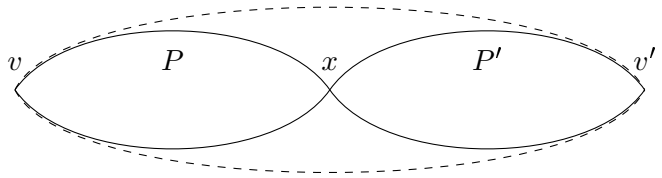

Figure 3.2: The path clusters P and P' merge.

Proof. To implement this, we maintain for each path-cluster P and for each boundary vertex $v \in \partial P$, the point $a_P(v)$ on the cluster-path closest to v

whose cover-level along the cluster-path does not exceed the cover-level of the cluster. That is, the first vertex to "blame" for the cover-level not being any higher.

This is easily done, by only a slight modification to the algorithm for merging path-clusters. Upon a merge of path-clusters P and P' whose common vertex is x and whose other boundary vertices are v and v', and whose cover levels are j and j', respectively, there are the following cases. We assume without loss of generality that $j \leqslant j'$, and denote the resulting cluster C.

- If x is not covered on level j, then $a_C(v) = a_C(v') = x$.
- Otherwise, if $j = j'$, then $a_C(v) = a_P(v)$ and $a_C(v') = a_P(v')$.
- Otherwise, $j < j'$, then $a_C(v) = a_P(v)$. For v' there are two cases:
  - If x is not covered on level $j+1$, then $a_C(v') = x$,
  - Otherwise, $a_C(v') = a_P(x)$.

We may thus for any not 2-connected vertices u, v answer FindFirstArticulation(u, v) by calling expose u, v and outputting $a_{\texttt{root}}(u)$. □

We now proceed to the proof of Theorem 3.4 via the following lemma:

Lemma 3.5. *Assume there is a data structure which for a graph of n vertices and $O(n)$ edges supports $O(n)$ edge-deletions, $O(n)$ connectivity queries, $O(n)$* Biconnected(u, v) *queries, and $O(n)$* FindFirstArticulation(u, v) *queries in total time $f(n)$, then:*

Given an graph on n vertices whose block-cutpoint tree is a path, and given any prescribed vertex x, there exists an $O(m + f(n))$ time algorithm which outputs the vertices of G in an order such that any suffix of size $\geqslant 3$ is connected by a cycle in G^2 containing x.

Proof. We implement the proof of Theorem 3.3. First, we may find the block-cutvertex graph in linear time using [154], and use the linear time algorithm of [74] on each block to obtain a spanning subgraph such that every block is minimally 2-connected. Dirac [42] proved that such a graph has at most $2n - 4$ edges.

We now have a graph G with n vertices and $O(n)$ edges. If the graph is not 2-connected, let u and v be non-cutpoint vertices of either endblock. Otherwise, choose u and v arbitrarily.

Algorithm 1 `VertexSets`: Near-linear time description of nested vertex sets

1: **Input:** A graph G, and vertices u, v, x of G.
2: Initialise a decremental biconnectivity structure for G.
3: $\triangleright$ Note, at this point, u and v are connected.
4: If only three vertices remain in the connected component of x, output them and Return.
5: **if** $x = u$ **then** $(u, v) := (v, u)$ $\triangleright$ Swap u and v.
6: **if** Biconnected(u, v) **then**
7: set $\alpha := v$
8: **else**
9: set $\alpha :=$ FindFirstArticulation(u, v).
10: Delete the tree-edge $w\alpha$ incident to α and on the tree-path towards u.
11: **if** $\neg$ Connected(α, w) **then** $\triangleright$ the block containing u consisted of only one edge $\alpha w = \alpha u$.
12: Output u,
13: set $u := \alpha$
14: goto Line 3.
15: $\triangleright$ We may now assume u and v are connected.
16: **if** Biconnected(α, u) **then**
17: goto Line 10.
18: **else**
19: goto Line 3.

Running time For the first part, a sparse 2-connected spanning subgraph can be found in $O(n+m)$ time, see [74, 154].

Throughout the course of the algorithm, we delete all edges of the graph, and perform $O(1)$ queries and gotos per deleted edge. Thus, the total running time for the second part will be $f(n)$.

Correctness follows from this being an implementation of the proof of Theorem 3.3. □

3.7 A modification of the algorithm

In the two previous theorems we used our initial algorithm on some modified graphs using the tricks in [27,155]. For the sake of completeness we remark that we can instead modify the algorithm so that we can work on the graph without modifying it.

Both algorithms rely on the following lemma about the Eulerification of the graph G with ear-decomposition $C^0, \ldots, C^k$ in Section 3.4.

Lemma 3.6. *Given an edge e of C^0, we may choose freely whether we want to double e by introducing a parallel edge, or not, while still maintaining the property that all vertices but one have a given degree parity, all vertices different from the starting vertex have indegree 1 or 2, and the starting vertex x_0^0 has indegree 0 or 1.*

Proof. When we choose not to double the first edge of C^0, we partition all edges of C^0 in two sets; those which have been doubled, and those which have not. By interchanging these sets and doubling exactly those edges that were not doubled before, the parity of the degree of vertices is unchanged. The edge e is doubled in exactly one of them. After choosing an appropriate doubling scheme, proceed as before, either deleting both copies of the last edge $(x_{l_0-1}^0, x_0^0)$, or both copies near y^0, or none, and orient as before. □

Consider first the case where we wish to find a Hamiltonian path between two prescribed vertices u, v in the square of a 2-connected graph G. It was shown in [27] that G^2 contains a Hamiltonian path from u to v.

Second proof of Theorem 3.2. As in the proof in Section 3.4, we wish to find an Euler walk J between u and v. As before, we shall choose J such that any vertex of indegree 2 has both incoming edges consecutive in J. Finally, when we traverse J, lifting the subwalk $z \leftarrow w \rightarrow z'$ to the edge $(z, z') \in G^2$,

we aim to obtain a Hamiltonian path P between u and v in G^2. To avoid having u, v occur twice in P, we shall ensure that each of u, v has at most one incoming edge, and any such incoming edge to u or v shall be an end of J. All other occurrences of u, v in J will be lifted.

To ensure that u and v are the only odd-degree vertices of the graph, and have indegree at most 1, we do the following. We can ensure that u, v both lie on the cycle C^0 of the proper ear decomposition by starting the depth first search of [148] with a cycle containing u and v. Then, we may choose u or v as x_0^0. We may assume without loss of generality that $x_0^0 = u$ and either $y^0 = v$, or y^0 is later than v along C^0. If $y^0 = v$, by Lemma 3.6, one may choose not to double the last edge on C^0, and then delete both copies of the edge near $y^0 = v$ that has been doubled (one of them must be doubled). If $y^0 \neq v$, by Lemma 3.6, we may choose not to double the edge after v, thus ensuring that v has indegree 1 while still maintaining that u has indegree $\leqslant 1$.

To make sure that any incoming edges to u, v are at the ends of J, we shall first delete any incoming edge (u', u) or (v', v) from the graph without disconnecting the graph. Then we find an Euler walk J' from u' to v', and finally, extend J' with the two in-edges: $J = (u, u')J'(v', v)$. We shall ensure that deleting any in-edge near u or v will not disconnect the graph, except that one of u, v may become an isolated vertex. For, if u has an in-edge, it is part of a double edge. Deleting an edge which is part of a double edge does not affect connectedness. If $v \neq y^0$ has even degree just before we double edges of C^0 (we call that graph G_1), then its in-edge (after we have doubled edges in C^0) will be part of a double edge. Again, the deletion of such an edge does not affect connectedness. If $v = y^0$, then v has only one incident edge after doubling edges on C^0, so deleting that edge will preserve connectedness except that v becomes an isolated vertex. So, there only remains the case that v has odd degree in G_1. As we can interchange between u and v we may assume that also u has odd degree in G_1. As u has odd degree in G_1, it has also odd degree in $G_1 - E(C^0)$. Therefore u is connected by a path P in $G_1 - E(C^0)$ to another vertex x of odd degree in G_1. That vertex x must be in C^0, as all vertices in G_1 but not on C^0 have even degree in G_1. Now we apply Lemma 3.1 to the three paths in $C^0 \cup P$ between x and u. Hence we may choose y^0 and an ordering of C^0 such that y^0 is later than x which is again later than (or equal to) v. We have earlier seen that deleting the directed edge (u', u) (if it exists) does not affect connectedness. It remain to prove that deleting the directed edge (v', v) (if it exists) also does not affect connectedness. So assume that (v', v) exists. Then v' is the predecessor of v on C^0. Note that if we delete a double edge on C^0, then that edge is on

the segment from x to u by the choice of y_0. This shows that, after deleting (v', v), we still have a path from v to v' using P and two appropriate paths on C^0. □

Finally we point out how to prove Lemma 3.3 directly.

Lemma 3.7. *Let G be a 2-connected graph, and let e be an edge of G. We may in linear time find a Hamiltonian cycle in $(G - e)^2$.*

Proof. First, find a minimally 2-connected spanning subgraph, G'. If $e \notin G'$, we are done. Otherwise, we may in linear time find a proper ear decomposition $C^0, \ldots, C^k$ with $e \in C^0$, and an enumeration $x_0^0, \ldots x_{l_0-1}^0$ of the vertices of C^0, such that $e = (x_{l_0-1}^0, x_{l_0}^0)$ is the last edge before x_0^0. Again, this follows from [148] by taking e as the first edge of the depth first search. Then, use Lemma 3.6 to ensure e is doubled, and thus, since it was the last edge before x, both copies are deleted. The Hamiltonian cycle found by the algorithm will now avoid the edge e. □

To prove Lemma 3.3 directly, assume G is a graph whose block-cutvertex tree is a path, and assume G is not 2-connected. Let x, y be non-cutvertices belonging to distinct endblocks. Then, $G' = G \cup (x, y)$ is 2-connected, and so by Lemma 3.7 we may find a Hamiltonian cycle in $(G' - (x, y))^2 = G^2$ in linear time.

Chapter 4

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

Jacob Holm, Eva Rotenberg, Mikkel Thorup

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 $\mathcal{O}(m + n\log n\log\log n)$. Finally,

we show how to obtain $\mathcal{O}(\log n/\log\log n)$ query time, matching the optimal trade-off between update and query time.

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

4.1 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 [131], this is equivalent to saying that a pair of connected vertices are two-edge connected if there exist two edge-disjoint paths between them. By edge-disjoint 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 [56] and Eppstein et al. [47]. Later, Frederickson [57] and Eppstein et al. [47] gave a data structure with $\mathcal{O}(\sqrt{n})$ update time for two-edge connectivity. Henzinger and King achieved polylogarithmic expected amortized time [78], 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 [77], $\mathcal{O}((\log n)^5)$ 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 in [83]; 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 [164]. Sacrificing determinism, an $\mathcal{O}(\log n(\log\log n)^3)$ structure for connectivity was presented by Thorup [157], and later improved to $\mathcal{O}(\log n(\log\log n)^2)$ by Huang et al. [97]. In the same paper, Thorup obtains an update time of $\mathcal{O}((\log n)^3\log\log n)$ for

deterministic two-edge connectivity. Interestingly, Kapron et al. [104] 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. We know of no similar result for bridge finding. The best lower bound known is by Pătraşcu et al. [141], 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 $t_u \lg \frac{t_q}{t_u} = \Omega(\lg n)$.

4.1.1 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 4.1. *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 for the first bridge, or $O(\log n/\log\log n + k)$ worst case time for the first k bridges.*
- *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 two-edge connected exactly when there is no bridge separating them, we have the following corollary:

Corollary 4.1. *There exists a data structure for dynamic multigraphs in the word RAM model with $\Omega(\log n)$ word size, that can answer two-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. [141]

As a stepping stone on the way to our main theorem, we show the following:

Theorem 4.2. *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.*

4.1.2 Applications

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 algorithm if implemented together with a decremental two-edge connectivity data structure with poly-logarithmic update time:

Theorem 4.3 (A. Kotzig '59 [118])**.** *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 [58] 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 [139] which states that any cubic, two-edge connected graph contains a perfect matching. An algorithm by Biedl et al. [20] finds a perfect matching in such graphs in $\mathcal{O}(n \log^4 n)$ time, by using the Holm et al two-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 two-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 [40] improved Biedl et al.'s algorithm for perfect matchings in two-edge connected cubic graphs, by having it rely only on dynamic connectivity, not two-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.

4.1.3 Techniques

As with the previous algorithms, our result is based on top trees [10] 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. [84] 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. [84], 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 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 [157]. 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 $\Theta(\log^c n)$.

4.1.4 Article outline

In Section 4.2, we recall how [84] fundamentally solves two-edge connectivity via a reduction to a certain set of operations on a dynamic forest. In Section 4.3, we recall how top trees can be used to maintain information in a dynamic forest, as shown in [10]. In Sections 4.4, 4.5, and 4.6, 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 4.2. Then, in Section 4.7, 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 4.1. We then revisit top trees in Section 4.8, 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 4.9. Finally, in Section 4.10, we show how to achieve optimal space, by only storing cluster information with large clusters, and otherwise calculating it from scratch when needed.

4.2 Reduction to operations on dynamic trees

In [84], two-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.
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.
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.
9. MinCoveredEdge(v, w). Returns a tree-edge on the path from v to w whose cover level is CoverLevel(v, w).
10. AddLabel(v, l, i). Associate the *user label* l to the vertex v at level i.
11. RemoveLabel(l). Remove the user label l from its vertex vertex(l).
12. FindFirstLabel(v, w, i). Find a user label at level i such that the associated vertex u has CoverLevel$(u, \text{meet}(u, v, w)) \geqslant i$ and minimizes the distance from v to meet(u, v, w).
13. FindSize(v, w, i). Find the number of vertices u such that

$$\text{CoverLevel}(u, \text{meet}(u, v, w)) \geqslant i.$$

Note that FindSize$(v, v, -1)$ is just the number of vertices in the tree containing v.

Lemma 4.1 (Essentially the high level algorithm from [84])**.** *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.

Proof. See Section 4.11 for a proof and pseudocode. □

#	Operation	Asymptotic worst case time per call, using structure in section				
		4.4	4.5	4.6	4.7	4.9
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$(v, v, -1)$	$\log n$	$\log n$	$\log n$	$\log n$	$\frac{\log n}{\log f(n)}$

Table 4.1: Overview of the worst case times achieved for each tree operation by the 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.

The algorithm in [84] 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 [157] 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 4.1). 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

Lemma 4.2 (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 4.2. Use the CoverLevel structure from Section 4.4, the FindSize structure from Section 4.5, and the FindFirstLabel structure from Section 4.6, and combine them into a single structure using Lemma 4.2. Then the reduction from Lemma 4.1 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 4.10. □

Proof of Theorem 4.1. Use the CoverLevel structure from Section 4.9, the FindSize structure from Section 4.5, as modified in Section 4.7 and 4.10, and the FindFirstLabel structure from Section 4.6, and combine them into a single structure using Lemma 4.2. Then the reduction from Lemma 4.1 gives the required bounds. □

4.3 Top trees

A *top tree* is a data structure for maintaining information about a dynamic forest. Given a tree T, a top tree $\mathcal{T}$ is a rooted tree over subtrees of T, such that each non-leaf node is the union of its children. The root of $\mathcal{T}$ is T, its leaves are the edges of T, and its nodes are *clusters*, which we will define in two steps. For any subgraph H of a graph G, the boundary ∂H consists of the vertices of H that have a neighbour in $G \backslash H$. A *cluster* is a connected subgraph with a boundary of size no larger than 2. We denote them by *point clusters* if the boundary has size $\leqslant 1$, and *path clusters* otherwise. For a path cluster C with boundary $\partial C = \{u, v\}$, denote by $\pi(C)$ the tree path between u and v, also denoted *the cluster path of* C. Similarly, for a point

cluster C with boundary vertex v, $\pi(C)$ is the trivial path consisting solely of v.

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 considered boundary vertices of every cluster containing them, including the root cluster. 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 two 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 their union has a boundary of size at most 2.

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.

Theorem 4.4 (Alstrup, Holm, de Lichtenberg, Thorup [10])**.** *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.*

4.4 A CoverLevel structure

In this section we show how to maintain a top tree supporting the CoverLevel operations. This part is is essentially the same as in [83, 84] (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 4.12.

For each cluster C we want to maintain the following two integers and up to two edges:

$$\begin{aligned}
\text{cover}_C &:= \min\{c(e) \mid e \in \pi(C)\} \cup \{\ell_{\max}\} \\
\text{globalcover}_C &:= \min\{c(e) \mid e \in C \backslash \pi(C)\} \cup \{\ell_{\max}\} \\
\text{minpathedge}_C &:= \operatorname*{arg\,min}_{e \in \pi(C)} c(e) \text{ if } |\partial C| = 2, \text{ and } \textbf{nil} \text{ otherwise} \\
\text{minglobaledge}_C &:= \operatorname*{arg\,min}_{e \in C \backslash \pi(C)} c(e) \text{ if } C \neq \pi(C), \text{ and } \textbf{nil} \text{ otherwise}
\end{aligned}$$

Then

$$\left.\begin{aligned}\text{CoverLevel}(v) &= \text{globalcover}_C\\ \text{MinCoveredEdge}(v) &= \text{minglobaledge}_C\end{aligned}\right\}$$
where C is the point cluster returned by Expose(v)

$$\left.\begin{aligned}\text{CoverLevel}(v,w) &= \text{cover}_C\\ \text{MinCoveredEdge}(v,w) &= \text{minpathedge}_C\end{aligned}\right\}$$
where C 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 $\text{cover}_C \geqslant \text{cover}_C^+$, and if $\text{cover}_C \leqslant \text{cover}_C^-$ then $\text{cover}_C = \text{cover}_C^+$.

This allows us to implement Cover(v,w,i) by first calling 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 Uncover(v,w,i) by first calling Expose(v,w), and then updating the returned path cluster C as follows if $\text{cover}_C \leqslant i$:

$$\text{cover}_C = -1 \qquad \text{cover}_C^+ = -1 \qquad \text{cover}_C^- = \max\left\{\text{cover}_C^-, i\right\}$$

Together, cover_C^- and cover_C^+ represent the fact that for each path descendant D of C, if $\text{cover}_D \leqslant \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$[1], 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 $\max\left\{\text{cover}_D, \text{cover}_D^-\right\} \leqslant \text{cover}_C^-$ we need to set

$$\text{cover}_D^- = \text{cover}_C^-$$

[1] In [83, 84] this condition is erroneously stated as $\text{cover}_D \leqslant \text{cover}_C^-$.

Furthermore, if $\text{cover}_D \leqslant \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$ we need to set

$$\text{cover}_D = \text{cover}_C^+ \qquad\qquad \text{cover}_D^+ = \text{cover}_C^+$$

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[2] $A_1, \ldots, A_k$ that we want to merge into a single new cluster C. For $1 \leqslant i \leqslant k$ define

$$\text{globalcover}'_{C,A_i} := \begin{cases} \text{globalcover}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \quad \text{or } \text{globalcover}_{A_i} \leqslant \text{cover}_{A_i} \\ \text{cover}_{A_i} & \text{otherwise} \end{cases}$$

$$\text{minglobaledge}'_{C,A_i} := \begin{cases} \text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C) \\ & \quad \text{or } \text{globalcover}_{A_i} \leqslant \text{cover}_{A_i} \\ \text{minpathedge}_{A_i} & \text{otherwise} \end{cases}$$

Note that for a point-cluster A_i, globalcover_{A_i} is always $\leqslant \text{cover}_{A_i} = l_{\max}$.

We then have the following relations between the data of the parent and the data of its children:

$$\begin{aligned} \text{cover}_C &= \ell_{\max} \text{ if } |\partial C| < 2, \text{ otherwise } \min_{1 \leqslant i < k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\ \text{minpathedge}_C &= \textbf{nil} \quad \text{if } |\partial C| < 2, \text{ otherwise } \text{minpathedge}_{A_j} \\ &\qquad \text{where } j = \operatorname*{arg\,min}_{1 \leqslant i < k, \partial A_i \subseteq \pi(C)} \text{cover}_{A_i} \\ \text{globalcover}_C &= \min_{1 \leqslant i < k} \text{globalcover}'_{C,A_i} \\ \text{minglobaledge}_C &= \text{minglobaledge}'_{C,A_j} \\ &\qquad \text{where } j = \operatorname*{arg\,min}_{1 \leqslant i < k} \text{globalcover}'_{C,A_i} \\ \text{cover}_C^- &= -1 \\ \text{cover}_C^+ &= -1 \end{aligned}$$

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 $\mathcal{O}(n)$.

[2] $k = 2$ for now, but we will reuse this in section 4.9 with a higher-degree top tree.

Note that we can extend this so for each cluster C, if all the least-covered edges (on or off the cluster path) lie in the same child of C, we have a pointer to the closest descendant D of C that is either a base cluster or has more than one child containing least-covered edges. We can use this structure to find the first k bridges in $\mathcal{O}(\log n + k)$ time.

4.5 A FindSize Structure

We now proceed to show how to extend the CoverLevel structure from Section 4.4 to support FindSize in $\mathcal{O}(\log n \log\log n)$ time per Merge and Split. Later, in Section 4.7 we will show how to reduce this to $\mathcal{O}((\log\log n)^2)$ time per Merge and Split. See Appendix 4.13 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 C, and let $0 \leqslant i < \ell_{\max}$. Define

$$\text{pointsize}_{C,v,i} := \big|\{u \in C \mid \text{CoverLevel}(u,v) \geqslant i\}\big|$$

For convenience, we will combine all the $\mathcal{O}(\log n)$ levels together into a single vector[3]

$$\text{pointsize}_{C,v} := \big(\text{pointsize}_{C,v,i}\big)_{\{0\leqslant i<\ell_{\max}\}}$$

Let $(C_v)_{\{v\in\pi(C)\}}$ be the point clusters that would result from deleting the edges of $\pi(C)$ from C. Then we can define the vector

$$\text{size}_C := \sum_{m\in\pi(C)} \text{pointsize}_{C_m,m}$$

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.

[3] All vectors and matrices in this section have indices ranging from 0 to $\ell_{\max} - 1$.

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 $v \in \partial C$ additionally maintain the following two size vectors for each $-1 \leqslant i \leqslant \ell_{\max}$:

$$\text{partsize}_{C,v,i} := \sum_{\substack{m \in \pi(C) \\ \text{CoverLevel}(v,m)=i}} \text{pointsize}_{C_m,m} \qquad \text{diagsize}_{C,v,i} := M(i) \cdot \text{partsize}_{C,v,i}$$

Where $M(i)$ is a diagonal matrix whose entries are defined (using Iverson brackets, see [116]) by

$$M(i)_{jj} = [i \geqslant j]$$

Note that these vectors are independent of cover_C^- and cover_C^+ as defined in Section 4.4. The corresponding "clean" vectors are not explicitly stored, but computed when needed as follows

$$\left.\begin{aligned} \text{partsize}'_{C,v,i} &= \begin{cases} \text{partsize}_{C,v,i} & \text{if } i > \ell \\ \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ 0 & \text{otherwise} \end{cases} \\ \text{diagsize}'_{C,v,i} &= \begin{cases} \text{diagsize}_{C,v,i} & \text{if } i > \ell \\ M(i) \cdot \sum_{j=-1}^{\ell} \text{partsize}_{C,v,j} & \text{if } i = \text{cover}_C^+ \\ 0 & \text{otherwise} \end{cases} \end{aligned}\right\} \quad \text{where } \ell = \max\left\{\text{cover}_C^-, \text{cover}_C^+\right\}$$

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 [84].

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 all types of merge in a normal binary top tree, except for the heterogeneous point clusters. 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

$$\begin{aligned}
\text{size}_C &= \text{size}_A + \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{partsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{partsize}'_{A,a,i} + \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{partsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases} \\
\text{diagsize}_{C,a,i} &= \begin{cases} \text{diagsize}'_{A,a,i} & \text{if } i > \text{cover}_A \\ \text{diagsize}'_{A,a,i} + M(i) \cdot \sum_{j=i}^{\ell_{\max}} \text{partsize}'_{B,c,j} & \text{if } i = \text{cover}_A \\ \text{diagsize}'_{B,c,i} & \text{if } i < \text{cover}_A \end{cases}
\end{aligned}$$

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

$$\begin{aligned}
\text{size}_C &= \left(\sum_{i=-1}^{\ell_{\max}} \text{diagsize}'_{A,a,i} \right) + M(\text{cover}_A) \cdot \text{size}_B \\
\text{partsize}_{C,a,i} &= \begin{cases} \text{size}_C & \text{if } i = \ell_{\max} \\ 0 & \text{otherwise} \end{cases} \\
\text{diagsize}_{C,a,i} &= \text{partsize}_{C,a,i}
\end{aligned}$$

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. [117, 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, $-1 \leqslant i \leqslant \ell_{\max}$ 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 $0 \leqslant j < \ell_{\max}$ 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 [83, 84, 157]. 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.

4.6 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 [10].

The idea is to maintain enough information in each cluster to determine if there is a result. Then we can start by using 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, 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[4]

$$\text{pointincident}_{C,v} := \left(\left[\exists \text{label } l \in C : \begin{array}{c} \text{CoverLevel}(v, \text{vertex}(l)) = i \\ \wedge \quad \ell(l) = i \end{array} \right] \right)_{\{0 \leqslant i < \ell_{\max}\}}$$

$$\text{incident}_C := \bigvee_{m \in \pi(C)} \text{pointincident}_{C_m, m}$$

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

$$\begin{aligned} \text{incident}_C &= ([v \text{ has associated labels at level } j])_{\{0 \leqslant j < \ell_{\max}\}} \\ \text{partincident}_{C,\text{vertex}(l),i} &= \begin{cases} \text{incident}_C & \text{if } i = \ell_{\max} \\ 0 & \text{otherwise} \end{cases} \\ \text{diagincident}_{C,\text{vertex}(l)} &= \text{partincident}_C \end{aligned}$$

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,

[4] Here, $[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$ is the *Iverson Bracket* (see [116]), and $\vee$ denotes bitwise OR.

let $v, w \in \bigcup_{i=1}^{k} \partial A_i$ (not necessarily distinct). Then we can define

$$\text{FindFirstLabel}((A_1, \cdots, A_k); v, w, i) = \begin{cases} A_x & \text{if } A_x \text{ is a label} \\ \text{FindFirstLabel}(\text{Split}(A_x); v_x, w_x, i) & \text{otherwise} \end{cases}$$

where for $1 \leqslant j \leqslant k$

$$v_j = \arg\min_{u \in \partial A_j} \text{dist}(v, u)$$

$$w_j = \arg\max_{u \in \partial A_j} \text{dist}(u, w)$$

and

$$I = \left\{ 1 \leqslant j \leqslant k \;\middle|\; \begin{array}{c} \text{CoverLevel}(v, v_j) \geqslant i \\ \wedge \quad \text{pointincident}_{A_j, v_j, i} = 1 \end{array} \right\}$$

$$x = \arg\min_{j \in I} \left(3 \cdot \text{dist}(v, \text{meet}(v_j, v, w)) + |\partial A_j \cap v \cdots w|\right)$$

$$\text{FindFirstLabel}(v, w, i) = \text{FindFirstLabel}(\text{Expose}(v, w); v, w, i)$$

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.

4.7 Approximate counting

As noted in [157], 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 $s' \leqslant s \leqslant e^{\epsilon} s'$, 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 $(1+2^{-b})^h \leqslant e^{2^{-b}h}$, 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 $h \leqslant h(n) \cdot \ell_{\max}$, 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 $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ 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 4.5. 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:

$$(M(i) \cdot x)_j = \begin{cases} x_j & \text{if } i \geqslant j \\ 0 & \text{otherwise} \end{cases}$$

And clearly this operation can also be done on $\mathcal{O}\left(\frac{\log n}{\log \log n}\right)$ 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 4.4 and the FindFirstLabel structure from Section 4.6 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 $\mathcal{O}(m + n(\log n)^2)$ algorithm from [83, 84] gives the

$\mathcal{O}((\log n)^3 \log n)$ time and $\mathcal{O}(m + n \log n \log\log n)$ algorithm briefly mentioned in [157].

4.8 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 [10].

4.8.1 Level-based top trees, labels, and fat-bottomed trees

As described in [10], 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. [10, 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 [10], we need not have single edges on the bottom most level. We may generalize this to instead have clusters of size Q as the leaves of the top tree.

Theorem 4.5 (Alstrup, Holm, de Lichtenberg, Thorup [10])**.** *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 $i \leqslant h$. 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 $H = \mathcal{O}(\log S)$.*

4.8.2 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 4.3. *Given any $B \geqslant 2$, one can maintain top trees of degree B and height $\mathcal{O}(\log n/\log B)$. 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 $\mathcal{O}(B(\log n/\log B))$ time.*

Proof. Given a binary levelled top tree $\mathcal{T}_2$ of height h, 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 $\mathcal{O}(B(\log n/\log B))$. □

4.8.3 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. The properties of the technique are captured in the following:

Lemma 4.4. *Given a top tree data structure of height $h(n) \in \mathcal{O}(\log n)$ that uses $s(n)$ space per cluster, and $t(n)$ worst case time per merge or split.*

Suppose that the complete information for a cluster of size q, including information that is shared with its children, has total size $s_0(q, n)$ and can be computed directly in time $t_0(q, n)$. Suppose further 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 top tree data structure, maintaining the same information, that uses linear space in total and has $\mathcal{O}(t(n) \cdot h(n) + t_0(q(n), n))$ update time for link, cut, and expose.

Proof. This follows directly from Theorem 4.5 by setting $Q = q(n)$. 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 of split and $\mathcal{O}(1)$ create and destroy per link cut and expose. These take $t(n)$ and $t_0(q(n), n)$ time respectively. □

4.9 A Faster CoverLevel Structure

If we allow ourselves to use bit tricks, we can improve the CoverLevel data structure from Section 4.4. The main idea is, for some $0 < \epsilon < 1$, to use top trees of degree $b(n) = (\log n)^\epsilon \in \mathcal{O}(w/\log \ell_{\max})$. 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 maintain a *buffer* with all the cover, cover^-, cover^+ and globalcover values one level up in the tree, in the parent cluster. 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 bit tricks to operate on the values for all children of a node in parallel.

Let C be a cluster with children $A_1, \ldots, A_k$. Since $k \leqslant w/\log \ell_{\max}$, we can define the following vectors that each fit into a constant number of words.

$$\begin{aligned}
\text{packedcover}_C &:= (\text{cover}_{A_i})_{\{1 \leqslant i \leqslant k\}} \\
\text{packedcover}^-_C &:= (\text{cover}^-_{A_i})_{\{1 \leqslant i \leqslant k\}} \\
\text{packedcover}^+_C &:= (\text{cover}^+_{A_i})_{\{1 \leqslant i \leqslant k\}} \\
\text{packedglobalcover}_C &:= (\text{globalcover}_{A_i})_{\{1 \leqslant i \leqslant k\}}
\end{aligned}$$

The description of Split and Merge from Section 4.4 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 $A_1, \ldots, A_k$, and define operations

- CleanToBuffer(C). For each $1 \leqslant i \leqslant k$: If A_i is a path child of C and $\max\{\text{packedcover}_{C,i}, \text{packedcover}^-_{C,i}\} \leqslant \text{cover}^-_C$, set:

$$\text{packedcover}^-_{C,i} = \text{cover}^-_C$$

Then if $\text{packedcover}_{C,i} \leqslant \max\left\{\text{cover}^-_C, \text{cover}^+_C\right\}$ set

$$\begin{aligned}\text{packedcover}_{C,i} &= \text{cover}^+_C\\ \text{packedcover}^+_{C,i} &= \text{cover}^+_C\end{aligned}$$

After updating all k children, set $\text{cover}^-_C = \text{cover}^+_C = -1$. Note that this can be done in parallel for all $1 \leqslant i \leqslant k$ in constant time using bit tricks.

- CleanToChild(C, i). If A_i is a path child of C and $\max\left\{\text{cover}_{A_i}, \text{cover}^-_{A_i}\right\} \leqslant \text{packedcover}^-_{C,i}$, set

$$\text{cover}^-_{A_i} = \text{packedcover}^-_{C,i}$$

Then if $\text{cover}_{A_i} \leqslant \max\left\{\text{packedcover}^-_{C,i}, \text{packedcover}^+_{C,i}\right\}$ set

$$\begin{aligned}\text{cover}_{A_i} &= \text{packedcover}^+_{C,i}\\ \text{cover}^+_{A_i} &= \text{packedcover}^+_{C,i}\end{aligned}$$

Finally set $\text{packedcover}^-_{C,i} = \text{packedcover}^+_{C,i} = -1$. Again, note that this takes constant time.

- ComputeFromChild(C, i). Set

$$\begin{aligned}\text{packedcover}_{C,i} &= \text{cover}_{A_i}\\ \text{packedcover}^-_{C,i} &= -1\\ \text{packedcover}^+_{C,i} &= -1\\ \text{packedglobalcover}_{C,i} &= \text{globalcover}_{A_i}\end{aligned}$$

- ComputeFromBuffer(C). For $1 \leqslant i \leqslant k$ define

$$\text{packedglobalcover}'_{C,i} = \begin{cases}\text{packedglobalcover}_{C,i} & \text{if } \partial A_i \subseteq \pi(C)\\ & \text{or } \text{packedglobalcover}_{C,i} \leqslant \text{packedcover}_{C,i}\\ \text{packedcover}_{C,i} & \text{otherwise}\end{cases}$$

$$\text{minglobaledge}'_{C,i} = \begin{cases}\text{minglobaledge}_{A_i} & \text{if } \partial A_i \subseteq \pi(C)\\ & \text{or } \text{globalcover}_{A_i} \leqslant \text{cover}_{A_i}\\ \text{minpathedge}_{A_i} & \text{otherwise}\end{cases}$$

We can then compute the data for C from the buffer as follows:

$$\text{cover}_C = \begin{cases} \min\limits_{\substack{1 \leqslant i < k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i} & \text{if } |\partial C| = 2 \\ \ell_{\max} & \text{otherwise} \end{cases}$$

$$\text{minpathedge}_C = \begin{cases} \text{minpathedge}_{A_j} & \text{if } |\partial C| = 2 \\ \quad \text{where } j = \arg\min\limits_{\substack{1 \leqslant i < k \\ \partial A_i \subseteq \pi(C)}} \text{packedcover}_{C,i} & \\ \textbf{nil} & \text{otherwise} \end{cases}$$

$$\text{globalcover}_C = \min_{1 \leqslant i < k} \text{packedglobalcover}'_{C,i}$$

$$\text{minglobaledge}_C = \text{minglobaledge}'_{C,j} \quad \text{where } j = \arg\min_{1 \leqslant i < k} \text{packedglobalcover}'_{C,i}$$

$$\text{cover}^-_C = -1$$

$$\text{cover}^+_C = -1$$

This can be computed in constant time, because $(\text{packedglobalcover}'_{C,i})_{\{1 \leqslant i \leqslant k\}}$ 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 $1 \leqslant i \leqslant k$ 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 $1 \leqslant i \leqslant k$, 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 v as internal vertices, and that once we have that, we can find a single child X of one of these clusters such that MinCoveredEdge$(v, w) = \text{minpathedge}_X$.

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.
- 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 $1 \leqslant i \leqslant k$ let v_i, w_i be the endpoints of the fragment closest to v, w respectively. Then

$$\begin{aligned}
\text{CoverLevel}(v, w) &= \min_{1 \leqslant i \leqslant k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v, w) &= \text{MinCoveredEdge}(v_j, w_j) \\
&\quad \text{where } j = \arg\min_{1 \leqslant i \leqslant k} \text{CoverLevel}(v_i, w_i) \\
\text{MinCoveredEdge}(v_j, w_j) &= \text{minpathedge}_X \\
&\quad \text{where } X = \arg\min_{Y \text{ path child of } C_{f_j}} \text{cover}_Y
\end{aligned}$$

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 $1 \leqslant i \leqslant k$ 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 $\text{packedcover}_{C_f,j} \leqslant i$, 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.

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 $\mathcal{O}(n)$.

Note that the pointers that allow us to find the first k least-covered edges can still be maintained in $\mathcal{O}(h)$ time per update, and allows us to find the first k least-covered edges in $\mathcal{O}(h + k)$ time.

4.10 Saving Space

We now apply the space-saving trick from Lemma 4.4 to the FindSize structures from Section 4.5 and 4.7. 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 4.5. *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 $-1 \leqslant i \leqslant \ell_{\max}$.

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 4.6. *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:

```
1: Vector x ← 0
2: Initialize empty max-queue Q
3: j ← ℓ_max
4: for w ← each vertex in the fragment that is on π(C) do
5:     Mark w as visited
6:     x_j ← x_j + 1
7:     for e ← each edge incident to w that is not on π(C) do
8:         if c(e) ⩾ 0 then
9:             Add e to Q with key c(e)
10: while Q is not empty do
11:     e ← EXTRACT-MAX(Q)
12:     while c(e) < j do
13:         x_{j−1} = x_j
```

14: $j \leftarrow j - 1$
15: $w \leftarrow$ the unvisited vertex at the end of e
16: Mark w as visited
17: $x_j \leftarrow x_j + 1$
18: **for** $e \leftarrow$ each edge incident to w that has an unvisited end **do**
19: **if** $c(e) \geqslant 0$ **then**
20: Add e to Q with key $c(e)$
21: $\text{partsize}_{C,v,i} \leftarrow x$
22: $\text{diagsize}_{C,v,i} \leftarrow M(i) \cdot x$

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 4.4 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.

4.11 Details of the high level algorithm

Lemma 4.1 (Essentially the high level algorithm from [84])**.** *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.

Proof. The only part of the high level algorithm from [84] 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 4.2) gives the desired bound. □

#	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 4.2: 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.

```
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)
17: function Insert(v, w, e)
18:     if ¬T.Connected(v, w) then
19:         T.Link(v, w, e)
20:         ℓ(e) ← ℓmax
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 = ℓmax 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
```

4.12 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
```

4.13 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

$$(\text{tree}_{\{x \leqslant i \leqslant y\}})$$

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: $C \leftarrow$ TopTree.Expose(v, w)
3: **return** $\text{size}_{C,i}$
4: **function** FS.Merge(C; A, B)
5: $\{c\} \leftarrow \partial A \cap \partial B$
6: **if** $c \in \pi(C)$ **then** ▷ Merge along path
7: **if** $|\partial C| <= 1$ **then**
8: $a \leftarrow c$, $b \leftarrow c$
9: **else**
10: $\{a, b\} \leftarrow \partial C$ with $a \in \partial A$ and $b \in \partial B$.
11: $\text{size}_C \leftarrow \text{size}_A + \text{size}_B$
12: **for** $(x, X) \leftarrow (a, A), (b, B)$ **do**
13: **if** $x = c$ **then**
14: $\text{tree}'_{X,x} \leftarrow \text{tree}_{X,x}$, $\text{undo}'_{X,x} \leftarrow$ **nil**
15: **else**
16: **for** $v \leftarrow x, c$ **do**
17: $\ell \leftarrow \max\left\{\text{cover}^-_X, \text{cover}^+_X\right\}$
18: $s \leftarrow (\text{tree}_{X,v}).\,\text{partsizesum}$
19: $d \leftarrow M(\text{cover}^+_X) * s$
20: $\text{tree}'_{X,v} \leftarrow \text{tree}_{X,v,\{i>\ell\}}$, $\text{undo}'_{X,v} \leftarrow \text{tree}_{X,v,\{i\leqslant\ell\}}$
21: $\text{tree}'_{X,v} \leftarrow \text{tree}'_{X,v} + [\text{cover}^+_X : s, d]$
22: **for** $(x, X, y, Y) \leftarrow (a, A, b, B), (b, B, a, A)$ **do**
23: $s \leftarrow (\text{tree}'_{Y,c,\{\text{cover}_X \leqslant i \leqslant \ell_{\max}\}}).\,\text{partsizesum}$
24: $p \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{partsize} + s$
25: $d \leftarrow (\text{tree}'_{X,x,\text{cover}_X}).\,\text{diagsize} + M(\text{cover}_X) * s$
26: **if** $x = c$ **then**
27: $\text{tree}''_{X,x} \leftarrow [\ell_{\max} : \text{size}_X, \text{size}_X]$, $\text{undo}''_{X,x} \leftarrow$ **nil**
28: **else**
29: $\text{tree}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i>\text{cover}_X\}}$, $\text{undo}''_{X,x} \leftarrow \text{tree}'_{X,x,\{i\leqslant\text{cover}_X\}}$
30: **if** $y = c$ **then**
31: $\text{tree}'''_{Y,c} \leftarrow$ **nil**, $\text{undo}'''_{Y,c} \leftarrow [\ell_{\max} : \text{size}_Y, \text{size}_Y]$
32: **else**
33: $\text{tree}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i<\text{cover}_X\}}$, $\text{undo}'''_{Y,c} \leftarrow \text{tree}'_{Y,c,\{i\geqslant\text{cover}_X\}}$
34: $\text{tree}_{C,x} \leftarrow \text{tree}''_{X,x} + [\text{cover}_X : p, d] + \text{tree}'''_{Y,c}$
35: **else** ▷ Merge off path
36: $\{a\} \leftarrow \partial C \setminus \{c\}$
37: **if** $a \notin \partial A$ **then**

Swap A and B
$\ell \leftarrow \max\left\{\text{cover}_A^-, \text{cover}_A^+\right\}$
$d \leftarrow (\text{tree}_{A,a,\{\ell<i\leqslant\ell_{\max}\}}).\,\text{diagsizesum}$
$p \leftarrow (\text{tree}_{A,a,\{-1\leqslant i\leqslant\ell\}}).\,\text{partsizesum}$
$\text{size}_C \leftarrow d + M(\text{cover}_A^+) * p + M(\text{cover}_A) * \text{size}_B$
$\text{tree}_{C,a} \leftarrow [\ell_{\max} : \text{size}_C, \text{size}_C]$

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**
      $\text{tree}''_{X,x} \leftarrow \text{tree}_{C,x,\{i>\text{cover}_X\}}$, $\text{tree}'''_{Y,c} \leftarrow \text{tree}_{C,x,\{i<\text{cover}_X\}}$
      if $y \neq c$ **then**
        $\text{tree}'_{Y,c} \leftarrow \text{tree}'''_{Y,c} + \text{undo}'''_{Y,c}$
      if $x \neq c$ **then**
        $\text{tree}'_{X,x} \leftarrow \text{tree}''_{X,x} + \text{undo}''_{X,x}$
    for $(x, X) \leftarrow (a, A), (b, B)$ **do**
      if $x \neq c$ **then**
        for $v \leftarrow x, c$ **do**
          $\text{tree}_{X,v} \leftarrow \text{tree}'_{X,v,\left\{i>\text{cover}_X^+\right\}} + \text{undo}'_{X,v}$

function FS.CREATE(C; edge e)
  $\text{size}_C \leftarrow 0$
  for $v \in \partial C$ **do**
    $\text{tree}_{C,v} \leftarrow [\ell_{\max} : 0, 0]$

function FS.CREATE(C; vertex label l)
  $\text{size}_C \leftarrow (1)_{\{0\leqslant i<\ell_{\max}\}}$
  for $v \in \partial C$ **do**
    $\text{tree}_{C,v} = [\ell_{\max} : \text{size}_C, \text{size}_C]$

Chapter 5

Decremental SPQR-trees for planar graphs

Jacob Holm, Giuseppe F. Italiano, Adam Karczmarz, Jakub Łącki and Eva Rotenberg

Abstract

We present a decremental data structure for maintaining the SPQR-tree of a planar graph subject to contractions and deletions of edges. The update time, amortized over $\Omega(n)$ operations, is $O(\log^2 n)$.

Via SPQR-trees, we show a decremental algorithm for maintaining 2- and 3-vertex connectivity in planar graphs. It answers queries in $O(1)$ time and processes edge deletions and contractions in $O(\log^2 n)$ amortized time. For 3-vertex connectivity in a planar graph subject to deletions, this is an exponential improvement over the previous best bound of $O(\sqrt{n})$ that has stood for over 20 years. In addition, the previous data structures only supported edge deletions.

5.1 Introduction

A graph algorithm is called *dynamic* if it is able to answer queries about a given property while the graph is undergoing a sequence of updates, such as edge insertions and deletions. It is *incremental* if it handles only insertions, *decremental* if it handles only deletions, and *fully dynamic* if it handles

both insertions and deletions. In designing dynamic graph algorithms, one is typically interested in achieving fast query times (either constant or polylogarithmic), while minimizing the update times. The ultimate goal is to perform fast both queries and updates, i.e., to have both query and update times either constant or polylogarithmic. So far, the quest for obtaining polylogarithmic time algorithms has been successful only in few cases. Indeed, efficient dynamic algorithms with *polylogarithmic* time per update are known only for few problems, such as dynamic connectivity, 2-connectivity, minimum spanning tree and maximal matchings in undirected graphs (see, e.g., [17, 80, 84, 91, 104, 150, 157, 163]). On the other hand, some dynamic problems appear to be inherently harder. For example, the fastest known algorithms for basic dynamic problems, such as reachability, transitive closure, and dynamic shortest paths have only *polynomial* times per update (see, e.g., [29,34,35,111,144,147,159]).

A similar situation holds for planar graphs where dynamic problems have been studied extensively, see e.g. [4,39,49,51,63,71,86,101,122–124,126,151]. Despite this long-time effort, the best algorithms known for some basic problems on planar graphs, such as dynamic shortest paths and dynamic planarity testing, still have polynomial update time bounds. For instance, for fully dynamic shortest paths on planar graphs the best known bound per operation is[1] $\widetilde{O}(n^{2/3})$ amortized [53, 101, 103, 113], while for fully dynamic planarity testing the best known bound per operation is $O(\sqrt{n})$ amortized [49].

In the last years, this exponential gap between polynomial and polylogarithmic bounds has sparkled some new exciting research. On one hand, it was shown that there are dynamic graph problems, including fully dynamic shortest paths, fully dynamic single-source reachability and fully dynamic strong connectivity, for which it may be difficult to achieve subpolynomial update bounds. This started with the pioneering work by Abboud and Vassilevska-Williams [3], who proved conditional lower bounds based on popular conjectures. Very recently, Abboud and Dahlgaard [2] proved polynomial update time lower bounds for dynamic shortest paths also on planar graphs, again based on popular conjectures.

On the other hand, the question of whether the best polynomial update bounds known for several other dynamic graph problems can be substantially improved (say to polylogarithmic bounds) has received much attention in the last years. For instance, there was a very recent improvement from polynomial to polylogarithmic bounds for decremental single-source reachability (and strongly connected components) in planar graphs: more precisely, the

[1] Throughout the paper,we use the notation $\widetilde{O}(f(n))$ to hide polylogarithmic factors.

improvement was from $O(\sqrt{n})$ amortized [122] to $O(\log^2 n \log\log n)$ amortized [99] (both amortizations are over sequences of $\Omega(n)$ updates). Other problems that received a lot of attention are fully dynamic connectivity and minimum spanning tree in general graphs. Up to last year, the best deterministic worst-case bound for both problems was $O(\sqrt{n})$ per update [48]: very recently, much effort has been devoted towards improving this bound (see e.g., [109, 134, 136, 164]).

In this paper, we follow the ambitious goal of achieving polylogarithmic update bounds for dynamic graph problems. In particular, we show how to improve the update times from polynomial to polylogarithmic for another important problem on planar graphs: decremental 3-vertex connectivity. Given a graph $G = (V, E)$ and two vertices $x, y \in V$ we say that x and y are 2-vertex connected (or, as we say in the following, *biconnected*) if there are at least two vertex-disjoint paths between x and y in G. We say that x and y are 3-vertex connected (or, as we say in the following, *triconnected*) if there are at least three vertex-disjoint paths between x and y in G. The decremental planar triconnectivity problem consists of maintaining a planar graph G subject to an arbitrary sequence of edge deletions, edge contractions, and query operations which test whether two arbitrary input vertices are triconnected. We remark that decremental triconnectivity on planar graphs is of particular importance. Apart from being a fundamental graph property, a triconnected planar graph has only one planar embedding, a property which is heavily used in graph drawing, planarity testing and testing for isomorphism [94, 96, 102].

Furthermore, our extended repertoire of operations, which includes edge contractions, contains all operations needed to obtain a graph minor, which is another important notion for planar graphs.

While polylogarithmic update bounds for decremental 2-edge and 3-edge connectivity, and for decremental biconnectivity in planar graphs have been known for more than two decades [63], decremental triconnectivity on planar graphs presents some special challenges. Indeed, while connectivity cuts for 2-edge and 3-edge connectivity, and for biconnectivity have simple counterparts in the dual graph or in the vertex-face graph (see Section 5.2 for a formal definition of vertex-face graph), triconnectivity cuts (separation pairs, i.e., pairs of vertices whose removal disconnects the graph) have a much more complicated structure in planar graphs. Roughly speaking, maintaining 2-edge and 3-edge connectivity cuts in a planar graph under edge deletions corresponds to maintaining respectively self-loops and cycles of length 2 (pairs of parallel edges) in the dual graph under edge contractions. On the other side, maintaining biconnectivity and triconnectivity cuts in a planar

graph under edge deletions corresponds to maintaining, respectively, cycles of length 2 and cycles of length 4 in the vertex-face graph. While detecting cycles of length 2 boils down to finding duplicates in the multiset of all edges, detecting cycles of length 4 under edge contractions is far more complex. We believe that this is the reason why designing a fast solution for decremental triconnectivity on planar graphs has been an elusive goal, and the best bound known of $O(\sqrt{n})$ per update [50] has been standing for over two decades.

Our results and techniques. In this paper, we show how to solve the decremental triconnectivity problem on planar graphs in constant time per query and $O(\log^2 n)$ amortized time per edge deletion or contraction, over any sequence of $\Omega(n)$ deletions and contractions. This is an exponential speed-up over the previous $O(\sqrt{n})$ long-standing bound [50]. To obtain our bounds, we also need to solve decremental biconnectivity on planar graphs in constant time per query and $O(\log^2 n)$ amortized time per edge deletion or contraction. (A better $O(\log n)$ amortized bound can be obtained if no contractions are allowed [86]). Our results are achieved with the help of two new tools, which may be of independent interest.

The first tool is an algorithm capable of detecting and reporting efficiently cycles of length 4 as they arise in a dynamic plane graph subject to edge contractions and edge insertions. The algorithm works for a graph with bounded face-degree, that is, where each face is delimited by at most some constant number of edges. Specifically, given a plane embedded graph with bounded face-degree subject to edge-contractions and insertions of edges across a face, after each dynamic operation we can report all edges that lie on a length-4 cycle because of this dynamic operation. The total running time is $O(n \log n)$. One of the challenges that we face is that a planar graph may have as many as $\Omega(n^2)$ distinct cycles of length 4. Still, we are able to show a surprisingly simple algorithm for solving this problem. The difficulty of the algorithm lies in the analysis — in fact, this analysis is the most technically involved part of this paper.

The second tool is a new data structure that maintains the SPQR-tree [36] of a planar graph, while the graph is updated with edge deletions and edge contractions, in $O(\log^2 n)$ amortized time per operation. While incremental algorithms for maintaining the SPQR tree were known for more than two decades [36,37], to the best of our knowledge no decremental algorithm was previously known.

Organization of the paper. The remainder of the paper is organized as follows. In Section 5.2, we introduce notation and definitions that we later use. Then, in Section 5.3 we present a high-level overview of our results. The details of the decremental algorithm for triconnectivity follow: Section 5.4 outlines an algorithm for detecting cycles of length 4 under contractions, with some details deferred to Appendix 5.7, while Section 5.5 presents our new algorithm for maintaining an SPQR-tree during edge deletions and contractions. Finally, Section 5.6 shows how to use the SPQR-trees in order to maintain information about triconnectivity.

5.2 Preliminaries

Throughout the paper we use the term *graph* to denote an undirected *multigraph*, that is we allow the graphs to have parallel edges and self-loops. Formally, each edge e of such a graph is a pair $(\{u, w\}, \mathrm{id}(e))$ consisting of a pair of vertices and a unique *integer identifier* used to distinguish between the parallel edges. For simplicity, in the following we skip the identifier and use just uw to denote one of the edges connecting vertices u and w. If the graph contains no parallel edges and no self-loops, we call it *simple*.

Given a graph G, we use $V(G)$ to denote the vertices, and $E(G)$ to denote the edges of G. For any $X \subseteq V(G)$ let $G[X]$ denote the subgraph $(X, \{(\{u, v\}, l) \in E(G) \mid u, v \in X\})$ of G *induced by* X.

The *components* of a graph G are the minimal subgraphs $H \subseteq G$ such that for every edge $uv \in E(G)$, $u \in V(H)$ if and only if $v \in V(H)$. The components of a graph partition the vertices and edges of the graph. A graph G is *connected* if it consists of a single component. For a positive integer k, a graph is *k-vertex connected* if and only if it is connected, has at least k vertices, and stays connected after removing any set of at most $k-1$ vertices. The *local vertex connectivity* of a pair of vertices u, v, denoted $\kappa(u, v)$, is the maximal number of internally vertex-disjoint u, v-paths. By Menger's Theorem [131], G is k-vertex connected if and only if $\kappa(u, v) \geqslant k$ for every pair of non-adjacent vertices u, v. We say that u, v are (locally) k-vertex connected if $\kappa(u, v) \geqslant k$. We follow the common practice of using *biconnected* as a synonym for 2-vertex connected and *triconnected* as a synonym for 3-vertex connected. An articulation point v of G is a vertex whose removal disconnects G. Thus a graph is biconnected if and only if it has no articulation points.

Let G be a graph and $e \in E(G)$. We use $G - e$ to denote the graph obtained from G by removing e. If e is not a self-loop, we use G/e to denote

the graph obtained by contracting e. A *cycle* C of length $|C| = k$ in a graph G is a cyclic sequence of edges $C = e_1, e_2 \ldots, e_k$ where $e_i = u_i u_{i+1}$ for $1 \leqslant i < k$ and $e_k = u_k u_1$. A cycle is *simple* if $\mathrm{id}(e_i) \neq \mathrm{id}(e_j)$ and $u_i \neq u_j$ for $i \neq j$. We sometimes abuse notation and treat cycle as a set of edges or a cyclic sequence of vertices. Note that this definition allows cycles of length 1 (a self-loop) or 2 (a pair of parallel edges).

Let G be a planar embedded graph. For each component H of G, let H^* denote the dual graph of H, defined as the graph obtained by creating a vertex for each face in the embedding of H, and an edge e^* for each edge e, connecting the two (not necessarily distinct) faces that e is incident to. Let G^* denote the graph obtained from G by taking the dual of each component. Each edge of $e \in E(G)$ naturally corresponds to an edge of G^*, which we denote e^*.

Each face f in a planar graph is bounded by a (not necessarily simple) cycle called the *face cycle* for f. We call the length of this cycle the *face-degree* of f. We call any other cycle a *separating cycle*.

Let G be a connected plane embedded multigraph with at least one edge. Define the set $E^\diamond(G)$ of *corners*[2] of G to be the the set of ordered pairs of (not necessarily distinct) edges (e_1, e_2) such that e_1 immediately precedes e_2 in the clockwise order around some vertex, denoted $v(e_1, e_2)$. Note that if $(e_1, e_2) \in E^\diamond(G)$, then $({e_2}^*, {e_1}^*) \in E^\diamond(G^*)$. We denote by $G^\diamond = (V(G) \cup V(G^*), E^\diamond(G))$ the *vertex-face* graph[3] of G (see Figure 5.1). This is a plane embedded multigraph with vertex set $V(G) \cup V(G^*)$, and an edge between $v(e_1, e_2)$ and $v({e_2}^*, {e_1}^*)$ for each corner $(e_1, e_2) \in E^\diamond(G)$. We use the following well-known facts about the vertex-face graph:

1. $G^\diamond$ is bipartite and planar, with a natural embedding given by the embedding of G.

2. The vertex-face graphs of G and G^* are the same: $G^\diamond = (G^*)^\diamond$.

3. There is a one-to-one correspondence between the edges of G and the faces of $G^\diamond$ (in the natural embedding, the interior of each face of $G^\diamond$ contains exactly one edge of G, see Fig 5.1).

4. $(G^\diamond)^*$ (also known as the *medial graph*) is 4-regular.

[2] For alternative definitions, see e.g. [88] and [145]. The latter uses the name *angles* for what we call corners.

[3] A.k.a. the *vertex-face incidence graph* [25], the *angle graph* [145], and the *radial graph* [15].

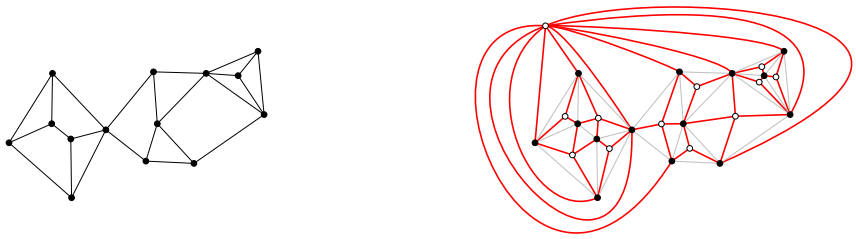

Figure 5.1: Left: a plane embedded graph. Right: the corresponding vertex-face graph (red) and the underlying graph (gray).

5. $G^\diamond$ is simple if and only if G is loopless and biconnected (See e.g. [26, Theorem 5(i)]).

6. $G^\diamond$ is simple, triconnected and has no separating 4-cycles if and only if G is simple and triconnected (See e.g. [26, Theorem 5(iv)]).

If v is an articulation point in G or has a self-loop, then in any planar embedding of G there is at least one face f whose face cycle contains v at least twice. Any such f is either an articulation point or has a self-loop in G^*, and v and f are connected by (at least) two edges in $G^\diamond$.

The dynamic operations on G correspond to dynamic operations on G^* and $G^\diamond$. Deleting a non-bridge edge e of G corresponds to contracting e^* in G^*, that is $(G-e)^* = G^*/e^*$. Similarly, contracting an edge e corresponds to deleting the corresponding edge from the dual, so $(G/e)^* = G^* - e^*$. Finally, deleting a non-bridge edge or contracting an edge corresponds to adding and then immediately contracting an edge across a face of $G^\diamond$ (and removing two duplicate edges).

The useful concept of a separation is well-defined, even for general graphs:

Definition 5.1. Given a graph $G = (V, E)$, a *separation* of G is a pair of vertex sets (V', V'') such that the induced subgraphs $G' = G[V'], G'' = G[V'']$ cover G, and $V' \backslash V''$ and $V'' \backslash V'$ are both nonempty. A separation is *balanced* if $\max\left\{|V'|, |V''|\right\} \leqslant \alpha |V|$ for some fixed constant $\frac{1}{2} \leqslant \alpha < 1$. If (V', V'') is a separation of G, the set $S = V' \cap V''$ is called a *separator* of G. A separator S is *small* if $|S| = O(\sqrt{|V|})$, and it is a *cycle separator* if the subgraph of G induced by S is Hamiltonian.

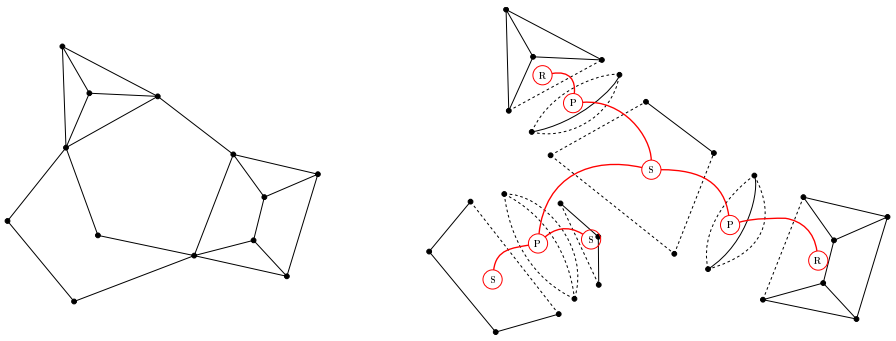

Figure 5.2: A biconnected graph and its SPQR tree. See Definition 5.3.

5.3 Overview of Our Approach

Our data structure for decremental triconnectivity in planar graphs consists of two main ingredients. Before describing them, we need few definitions. We recall that a graph G that is biconnected but not triconnected has at least one separation pair, i.e., a pair of vertices that can be removed to disconnect G:

Definition 5.2 (Hopcroft and Tarjan [93, p. 6])**.** Let $\{a, b\}$ be a pair of vertices in a biconnected multigraph G. Suppose the edges of G are divided into equivalence classes $E_1, E_2, \ldots, E_k$, such that two edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an end-point are in the same class. The classes E_i are called the *separation classes* of G with respect to $\{a, b\}$. If there are at least two separation classes, then $\{a, b\}$ is a *separation pair* of G unless (i) there are exactly two separation classes, and one class consists of a single edge, or (ii) there are exactly three classes, each consisting of a single edge.

Note that *separation pair*, which is a pair of vertices, should not be confused with *separation* (see Definition 5.1), which is a pair of vertex sets.

Our first ingredient for decremental triconnectivity is an algorithm for detecting efficiently separation pairs in planar graphs. The second ingredient is the maintenance of the SPQR-tree [36] for each biconnected component of a graph G under edge deletions and contractions. The SPQR-tree captures the structure of all separating pairs, and can be defined as follows:

Definition 5.3. The SPQR-tree for a biconnected multigraph $G = (V, E)$ with at least 3 edges is a tree with nodes labeled S, P, or R, where each node x has an associated *skeleton graph* $\Gamma(x)$ with the following properties:

- For every node x in the SPQR tree, $V(\Gamma(x)) \subseteq V$.
- For every node x in the SPQR tree, every edge in $\Gamma(x)$ is either in E or a *virtual edge* corresponding to an edge (x, y) in the SPQR-tree.
- For every edge $e \in E$ there is a unique node x in the SPQR-tree such that $e \in E(\Gamma(x))$.
- For every edge (x, y) in the SPQR tree, $V(\Gamma(x)) \cap V(\Gamma(y))$ is a separation pair $\{a, b\}$ in G, and there is a virtual edge ab in each of $\Gamma(x)$ and $\Gamma(y)$.
- If x is an S-node, $\Gamma(x)$ is a simple cycle with at least 3 edges.
- If x is a P-node, $\Gamma(x)$ consists of a pair of vertices with at least 3 parallel edges.
- If x is an R-node, $\Gamma(x)$ is a simple triconnected graph.
- No two S-nodes are neighbors, and no two P-nodes are neighbors.

It turns out (see e.g. [36]) that the SPQR-tree for a biconnected graph is unique. The (skeleton graphs associated with) nodes of the SPQR-tree are sometimes referred to as the triconnected *components* of G.

Detecting separating 4-cycles. We observe that there is a one-to-one correspondence between separation pairs in G and simple separating cycles of length 4 in the vertex-face graph $G^\diamond$. We call such cycles *separating 4-cycles* for short. We build a structure for 4-cycle detection by recursively using balanced separators, and by detecting, for each separator, the cycles that cross the separator. Detecting 4-cycles that cross a separator is not trivial, and our analysis introduces a potential function which reflects how well connected the non-separator vertices are with the separator, that is, how many neighbors on the separator they have. We exploit the fact that for a planar graph with separator S, there cannot be more than $O(|S|)$ vertices that have more than 4 neighbors in S.

The recursive use of separators can be sketched as follows: Let S be a small balanced separator in $G = (V, E)$ that induces a separation (V_1, V_2), that is, $V_1 \cap V_2 = S$ and $V_1 \cup V_2 = V$. Moreover, let $n = |V|$. We observe that each 4-cycle is fully contained in V_1 or V_2, or consists of two paths of length 2 that connect vertices of S. This motivates the following recursive approach. We compute a separator S of $O(\sqrt{n})$ vertices and then find all

paths of length 2 that connect vertices of S. Since the size of S is $O(\sqrt{n})$, there are only $O(n)$ pairs of vertices of S, and for each pair of vertices, we can easily check if the two-edge paths connecting them form any separating 4-cycles. It then remains to find the 4-cycles that are fully contained in either V_1 or V_2, which can be done recursively. Because S is a balanced separator, the recursion has $O(\log n)$ levels.

This algorithm can be made dynamic under contractions and edge insertions that respect the embedding of G. Contractions are easy to be handled, as they preserve planarity. Moreover, a separator S of a planar graph can be easily updated under contractions. Namely, whenever an edge uw is contracted, the resulting vertex belongs to the separator iff any of u and w did. Insertions that preserve planarity, however, are in general harder to accommodate. To handle this we introduce a new type of separators that we call *face-preserving* separators, which (like cycle-separators) always exist when the face-degree is bounded. These are still preserved by contractions, but also ensure that any edge across a face can be inserted.

All in all, there are $O(\log n)$ levels of size $O(n)$ each, where each level handles insertions and contractions in constant time, leading to a total of $O(n \log n)$ time.

Maintaining SPQR trees. The main challenge for maintaining an SPQR-tree is when an edge within a triconnected component is deleted. First of all, the data structure should be able to detect whether or not the component is still triconnected.

For any triconnected component Γ of G, we maintain a 4-cycle detection structure for the corresponding vertex-face graph $\Gamma^\diamond$. A separating 4-cycle in $\Gamma^\diamond$ corresponds to a separation pair in Γ, which would witness that Γ is no longer triconnected. The deletion or contraction of the edge e in the triconnected component Γ of G corresponds to an (embedding-respecting) insertion and immediate deletion of an edge in $\Gamma^\diamond$. This way by detecting 4-cycles in $\Gamma^\diamond$, we can detect when the corresponding triconnected component falls apart.

However, this is not the only challenge. If Γ does indeed cease to be triconnected, the SPQR-tree of $(\Gamma - e)$ (or (Γ/e) when doing a contraction) is a path $\mathcal{P}$. This is where we need the 4-cycle structure to output the edges contained in separating 4-cycles. Those edges correspond to a set of corners N of G. We use those corners to guide a search, which helps identify the non-largest components of the SPQR-path. More specifically, if a vertex v now belongs to two distinct triconnected components, there are two corners

in N that separate the edges incident to v into two groups of edges, each belonging to a distinct triconnected component. We can afford to build a 4-cycle detection structure for $\Gamma'^{\diamond}$ for any non-largest triconnected component Γ' on the path from scratch. To obtain the data structure representing the largest component, we delete or contract the corresponding edges from Γ while updating $\Gamma^{\diamond}$. Since an edge only becomes part of a structure built from scratch when its triconnected component size has been halved, this happens only $O(\log n)$ times per edge, so the total time used for rebuilding is $O(n \log^2 n)$. The second logarithm comes from rebuilding the data structure for 4-cycle detection, that takes $O(n \log n)$ time to initialize and process any number of operations.

Finally, since no two S-nodes can be neighbors and no two P-nodes can be neighbors, some S- or P-nodes in $\mathcal{P}$ may have to be merged with their (at most 2) neighbors of the same type outside $\mathcal{P}$. To handle this step efficiently, we keep the SPQR tree rooted in an arbitrary node. While merging the skeleton graphs of two S- or P- nodes can be done in constant time, what can be more costly is updating the parent pointers in the children of the merged nodes. Hence, we move the children of the node with fewer children to the other node. This way, each node changes parent at most $O(\log n)$ times before it is deleted or split. The total number of distinct SPQR-nodes that exist throughout the lifetime of the data structure is only $O(n)$, so the total time used for maintaining the parent pointers is $O(n \log n)$.

Since SPQR-trees are only defined for biconnected graphs, another challenge is to maintain SPQR-trees for each biconnected component, even as the decremental update operations cause the biconnected components to fall apart. We recall here that the structure of the biconnected components of a connected graph can be described by a tree called the *block-cutpoint tree* [75, p. 36], or *BC-tree* for short. This tree has a vertex for each biconnected component (block) and for each articulation point of the graph. There is an edge in the BC-tree for each pair of a block and an articulation point that belongs to that block. If the tree is rooted arbitrarily at any block, each non-root block has a unique articulation point separating it from its parent.

To handle updates, we notice that the SPQR-tree points to the fragile places where the graph is about to cease to be biconnected: An edge deletion in an S-node will break up a block in the BC-tree into path, and an edge contraction in a P-node breaks a block in the BC-tree into a star. Upon such an update, we remove the aforementioned S- or P-node from the SPQR-tree, breaking it up into an SPQR-forest. Each tree corresponds to a new block in the BC-tree. They form a path (or a star), and the ordering along the path,

as well as the articulation points, can be read directly from the SPQR-tree.

On the other hand, in order to even know which SPQR tree to modify during an update, we can use the BC-tree to search for the right SPQR-structure in which to perform the operation.

Bi- and triconnectivity. Finally, we use SPQR-trees to facilitate triconnectivity queries. First of all, vertices need to be biconnected in order to be triconnected. To facilitate biconnectivity queries, it is enough that each vertex v knows the name of the block $B(v)$ closest to the root in the BC-tree that contains it, and each block b knows the name of the vertex separating it from the parent $p(b)$. Then, any two vertices u and w are biconnected if and only if one of the following occur: $B(u) = B(v)$, or $u = p(B(v))$, or $v = p(B(u))$.

The information we maintain for triconnectivity is similar, using the SPQR-tree. Namely: Each non-root node x in the SPQR-tree knows the *virtual edge* (see Definition 5.3) that separates it from its parent. Each vertex v knows the name of the node $C(v)$ closest to the root that contains it, and, in a special case, at most two other nodes that are the children of $C(v)$. Queries are handled similarly to above.

The main challenge is to handle updates. Note that the change to the SPQR-tree may involve both the split and merge of nodes. In particular, we have one split and up to several merges when a triconnected component falls apart into an SPQR-path. However, upon a merge, we can afford to update the information regarding vertices in the non-largest components, costing only an additive $\log n$ to the amortized running time. Similarly, upon a split, we update any information that relates to vertices in the non-largest components only.

The total running time is thus $O(n \log n + f(n))$, where $f(n)$ is the running time for maintaining the SPQR-tree.

5.4 Detecting 4-Cycles Under Edge Contractions and Insertions

In this section we give an algorithm for detecting 4-cycles (simple cycles of length 4) in a planar embedded graph that undergoes contractions and edge insertions that respect the embedding. We say that a 4-cycle in a planar embedded graph G is a *face 4-cycle* if it is a cycle bounding a face of G, and a *separating 4-cycle* otherwise. For our purposes, only the separating

4-cycles are interesting, but we note in passing that new face 4-cycles are easy to detect under edge insertions and contractions:

Observation 5.1. *An embedding-respecting edge insertion creates two new faces, and we may check in constant time whether each of them has degree* 4 *or not. An edge contraction affects degrees of only two faces (the two incident to the contracted edge), and we may check in constant time whether their new degree is* 4 *or not.*

Since no two parallel edges can lie on the same 4-cycle, and no self-loop can be contained in a 4-cycle, we can assume the input graph is simple. However, when we contract edges, new parallel edges and self-loops may arise. To handle this, we could detect and remove parallel edges, but it turns out that both the algorithm and the analysis become simpler if we keep (most of) the additional edges, as long as no two parallel edges are consecutive in the circular ordering around both their endpoints.

If G has a face bounded by two edges, we can *simplify* the graph by deleting one of them. For our purposes we do not really care which one is deleted, but we need a rule that is consistent. For presentational purposes, we assume that we always keep the edge e with larger $\mathrm{id}(e)$.

This motivates the following definition of a quasi-simple graph[4]:

Definition 5.4. A plane embedded graph is *quasi-simple* if the dual of each non-simple component has minimum degree 3. Given a plane embedded graph G and a set of vertices X, we define the subgraph of G *quasi-induced* by X to be the unique quasi-simple subgraph of G with vertex set X and the maximum total sum of $\mathrm{id}(e)$ values. Let $d_X(v)$ denote the degree of v in the subgraph quasi-induced by $X \cup \{v\}$.

Roughly speaking, a quasi-simple graph is obtained from a plane embedded multigraph by merging parallel edges that lie next to each other in the circular orderings around both their endpoints. Note that in a quasi-simple connected graph of at least 3 vertices, every face has degree 3 or more. We can thus use Euler's formula on each component to obtain the following:

Observation 5.2. *In a quasi-simple planar graph with* $n \geqslant 3$ *vertices, the number of edges is at most* $3n - 6$.

The main goal of this section is to prove the following theorem.

[4] In [114] these graphs are called *semi-strict*.

Theorem 5.1. *Let G be an n-vertex quasi-simple plane embedded graph with bounded face degree. There exists a data structure that maintains G under contractions and embedding-respecting insertions, and after each update operation reports edges that become members of some separating 4-cycle. It runs in $O(n \log n)$ total time.*

In order to detect 4-cycles, we use planar separators. In fact, in order to maintain our data structure dynamically, we need something a little bit stronger.

Definition 5.5. Given a planar graph G, a separation (A, B) of G is said to be *face-preserving* if for any face f of G, all vertices of f belong to A or all vertices of f belong to B.

For instance, given a cycle separator K, we can form a face-preserving separation (A, B) such that $A \cap B = K$. Namely, K corresponds to a Jordan curve dividing the plane into two parts, S_A, S_B, where every face lies entirely in one part. Define A by all vertices incident to faces on S_A, and B similarly. Then, $A \cup B = G$, and $A \cap B = K$.

Lemma 5.1. *Let (A, B) be a face-preserving separation in G. Let G' be the result of an embedding-respecting edge insertion or an edge contraction, and let A', B' be the vertices corresponding to A and B in G'. Then (A', B') is a face-preserving separation in G', and $|A' \cap B'| \leqslant |A \cap B|$.*

Proof. If an edge uv is inserted that respects the embedding, it is inserted into some face f. By definition at least one of A, B contain all vertices on f, and in particular it also contains all the vertices of the two new faces that appear in G'. Since $A = A'$ and $B = B'$ in this case, the result follows.

If an edge uv is contracted, the resulting graph G' has the same faces as G, and the separation (A', B') is clearly face-preserving. If $u, v \in A \cap B$, then $|A' \cap B'| = |A \cap B| - 1$. Otherwise uv has an endpoint outside $A \cup B$. Without loss of generality, we can assume that $u \in A \backslash B$. In that case, $v \in A$ and $B = B'$ and it follows that $|A'| = |B'|$. □

In our algorithm we need to maintain separations under edge insertions and contractions. Let (A, B) be a separation in G. When an edge is inserted, we do not modify the separation. When an edge uw is contracted into a vertex x, we obtain a new separation (A', B') as follows. If $u \in A$ or $w \in A$, we set $A' = (A \backslash \{u, w\}) \cup \{x\}$. Otherwise, $A' = A$. The set B' is defined analogously. Thanks to this convention, we obtain the following.

Lemma 5.2. *Let (A, B) be a face-preserving separation in G. Let G' be the result of an embedding-respecting edge insertion or edge contraction, and let A', B' be the vertices corresponding to A and B in G'. Then (A', B') is a face-preserving separation in G', and $|A \cap B| - 1 \leqslant |A' \cap B'| \leqslant |A \cap B|$.*

Proof. If an edge is inserted that respects the embedding, it is inserted into some face f. By definition at least one of A, B contain all vertices on f, and in particular it also contains all the vertices of the two new faces that appear in G'. Since $A = A'$ and $B = B'$ in this case, the result follows.

If an edge uv is contracted, the resulting graph G' has the same faces as G, and the separation (A', B') is clearly face-preserving. If $u, v \in A \cap B$, then $|A' \cap B'| = |A \cap B| - 1$. Otherwise uv has an endpoint outside $A \cup B$. Without loss of generality, we can assume that $u \in A \backslash B$. In that case, $v \in A$ and $B = B'$ and it follows that $|A' \cap B'| = |A \cap B|$. □

Definition 5.6. Given a graph G, a *separator tree* is a binary tree where each node x is associated with an induced subgraph H_x of G, such that for some constant $n_0 > 0$:

- If x is the root, $H_x = G$.
- If $|V(H_x)| > n_0$ then x has children y, z such that $(V(H_y), V(H_z))$ is a balanced separation of H_x with a small separator $S_x = V(H_y) \cap V(H_z)$.
- If $|V(H_x)| \leqslant n_0$ then x is a leaf.

A separator tree is a *cycle separator tree* if S_x is a cycle separator, and it is *face preserving* if $(V(H_y), V(H_z))$ is face-preserving, for all nodes x with children y and z.

Lemma 5.3. *Given a planar graph with bounded face degree, we can in $O(n \log n)$ time build a face-preserving separator tree where each node x explicitly stores S_x and H_x. This tree has height $O(\log n)$, and uses $O(n \log n)$ space.*

We construct the tree in three steps. First, we take our graph G and make a triangulation $G^{\triangle}$. Then, referring to a result by Klein, Mozes, and Sommer [115], we make a cycle separator tree for $G^{\triangle}$. Finally, we transform the cycle separator tree for $G^{\triangle}$ to a face preserving separator tree for G.

Proof. Let G be a graph with maximum face-degree k and let $G^{\triangle}$ be a triangulation of G. Then using the algorithm from [115, Theorem 3], we can in linear time compute a cycle separator tree for $G^{\triangle}$. Since the cycle separator tree is balanced, it has height $h \in O(\log n)$. Since the children of

each node partition the faces, the total number of faces (and hence vertices) in graphs associated with depth i nodes is $O(n)$ for each $0 \leqslant i < h$. Thus the total size of all these graphs in the cycle separator tree is $O(n \cdot h) = O(n \log n)$.

The cycle separator tree from [115, Theorem 3] has the additional property that for each node x with n_x vertices in H_x, both the separator S_x and the boundary of x, B_x bounding the region corresponding to x in the plane, have size $O(\sqrt{n_x})$.

Now, for each edge $uv \in E(G^{\triangle})\backslash E(G)$, and for each cycle separator S_x that contains uv, we can add all the at most $k-2$ remaining vertices on the face in G crossed by uv to S_x. Similarly, for each boundary B_x containing uv, we can add all the at most $k-2$ remaining vertices on the face in G to H_x. This increases the size of each separator S_x (and of each associated graph H_x) by at most a factor $k-2$, and makes the tree face-preserving for G. The total time to explicitly construct the face-preserving separator tree and all the associated graphs is $O(kn \log n)$. □

Lemma 5.4. *Let G be a graph and (A, B) be a separation of G. Then, any 4-cycle either has exactly one vertex in $A\backslash B$, one vertex in $B\backslash A$, and the remaining two vertices in $A \cap B$, or the 4-cycle is completely contained in at least one of A or B.*

Proof. From Definition 5.1, for each edge e of G, both endpoints of e are in A or B. Thus, an edge that has one endpoint in $A\backslash B$ has its other endpoint in A. Using these facts, the lemma follows by simple case analysis. □

It follows that we can use the separator tree to detect 4-cycles as follows. For each leaf x of the separator tree, the graph H_x has constant size, so 4-cycles inside H_x can be detected in constant total time. In any other node, we have a graph with a separator K, and we need to dynamically detect 4-cycles that cross K under edge contractions and embedding-respecting edge insertions. Referring to Lemma 5.4 above, we only need to detect the two halves of a 4-cycle, that is, length-2 paths between vertices of K.

Lemma 5.5. *Let G be a plane embedded graph on n vertices and K be a set of vertices of G of size $|K| = O(\sqrt{n})$. Assume that G undergoes edge contractions and insertions respecting the embedding. There exists a data structure that after each update operation can report the edges of G that become members of separating 4-cycles whose two opposing vertices lie on K. Its total running time is $O(n + k)$, where k is the total number of edge contractions and insertions.*

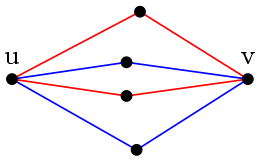

Figure 5.3: 4 paths all participate in separating 4-cycles.

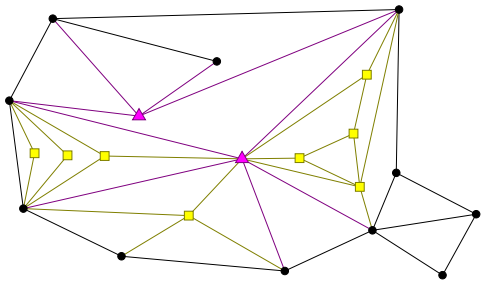

Figure 5.4: The sets M (magenta), Y (yellow), and K (black) from Lemma 5.6.

We now proceed with the description of the data structure of the above lemma. First note that if a pair of vertices is connected by at least 4 paths of length two, all edges on those paths lie on separating 4-cycles (see Figure 5.3). Thus, if we can keep, for every pair of vertices of K, the list of all length 2 paths between them, we need to check at most 2 existing paths when a new path arrives, and then report at most 8 edges (4 new length-2 paths) that now belong to separating 4-cycles.

To report every edge only once, we also keep a Boolean flag for each edge that indicates whether it has been reported before, and check that flag before reporting.

We thus only need to argue that we can detect all the length-2 paths between K-vertices that arise in the graph under contractions and edge insertions, in $O(n + k)$ total time. We do that by constructing a potential function Φ that is initially $O(n)$, remains nonnegative, and drops at each operation proportionally to the amount of work done. We start by partitioning the vertices into 3 sets, that we need to treat differently.

Lemma 5.6. *Given a planar graph $G = (V, E)$ and a vertex set $K \subseteq V$, let M denote the vertices $m \in V \backslash K$ that have $d_K(m) \geqslant 4$ (see Definition 5.4), and let Y denote $V \backslash (M \cup K)$. Then Y, M and K form a partition of V, and $|M| \leqslant |K| - 2$.*

Proof. The partition property follows trivially from the definition. Consider the maximal quasi-simple bipartite subgraph H of G with bipartition (M, K). By definition, each $m \in M$ has at least 4 neighbors in the subgraph of G quasi-induced by $K \cup \{m\}$. Thus for each $m \in M$, $d_H(m) \geqslant 4$, and so $4|M| \leqslant E(H)$. Since H is bipartite and quasi-simple, by Euler's formula we have $E(H) \leqslant 2(|M| + |K|) - 4$. Combining the two we get

$4|M| \leqslant E(H) \leqslant 2(|M| + |K|) - 4$, which implies $|M| \leqslant |K| - 2$. □

The aim of the data structure is to notice when new common neighbors of pairs of vertices in K appear. The idea is that since K has size $O(\sqrt{|V|})$, there are only $O(|V|)$ pairs of vertices of K. For each such pair, we maintain a doubly-linked list of all length-2 paths between them, and for each edge we maintain a doubly-linked list of all such paths it participates in.

When an edge uw is inserted, new length-2 paths can only appear if $u \in K$ and/or $w \in K$. In this case the number of candidate paths to check is bounded by $d_K(u) + d_K(w)$, and this can be done in constant time per path.

When an edge uw is contracted, new length-2 paths between vertices of K may appear in other ways. For example, we have new paths between neighbors of u contained in K and neighbors of w in K. Other cases are possible if u or w belongs to K.

We now define a potential function that decreases by at least the number of candidate paths after each contraction. We can decide if each candidate path is an actual length-2 path between vertices in K in constant time.

It is defined in stages:

$$\Phi_q(X) = \sum_{v \in X} d_V(v)$$
$$\Phi_v(X) = 4|X| - \frac{1}{2}\sum_{v \in X} d_X(v) = 4|V(G_X)| - |E(G_X)|$$
$$\Phi_s(X) = 63(\Phi_v(X))^2 - \sum_{v \in X} (d_X(v))^2$$
$$\Phi = 6\Phi_v(V) + 3\Phi_q(Y \cup M) + \Phi_s(M \cup K)$$

Lemma 5.7. *The potential Φ is initially $O(n)$ and remains nonnegative. The embedding-respecting insertion or contraction of an edge uv decreases Φ by at least the number of candidate paths.*

The proof is straight-forward but tedious. See Appendix 5.7 for details. As a result, Lemma 5.5 follows, and we are finally ready to prove Theorem 5.1.

Proof of Theorem 5.1. Given a planar graph G with bounded face-degree, we build a face-preserving separator tree as in Lemma 5.3 in $O(n \log n)$ time. For each internal vertex of the tree, we may detect new 4-cycles crossing the separator due to Lemma 5.5. The leaves have size at most $n_0 = O(1)$, and we can detect 4-cycles in the leaves in $O(1)$ time.

We can distinguish between face 4-cycles and separating 4-cycles. Namely, we can choose to report only when an edge first lies on any 4-cycle, or when it first lies on a separating one, as described in Lemma 5.5.

An edge insertion in the graph H_x needs to be duplicated in each child of x that contains both vertices. However, the drop in the Φ potentials for each node is large enough to pay for each cascading insertion. Whenever a contraction in the graph H_x for a node x of the separator tree introduces a new edge between two separator vertices, that edge may need to be added to the subtree containing the other side of the separation, but again that is paid for. In general, if we update graphs closer to the root first, the changes only propagate down and every change is paid for by a corresponding drop in the potential. □

5.5 Decremental SPQR-trees

In this section we will show how to use this to maintain an SPQR-tree (see Definition 5.3) for each biconnected component of G with at least 3 edges under arbitrary edge deletions and contractions. We start by giving some useful facts.

For a planar graph, there is a nice duality, as proven by Angelini et al. [14, Lemma 1]. Define the dual SPQR-tree as the tree obtained from the SPQR-tree by interchanging S- and P-nodes, taking the dual of the skeletons, and substituting virtual edges by their duals.

Lemma 5.8 (Angelini et al [14])**.** *The SPQR-tree of G^* is the dual SPQR-tree of G.*

Let G be a connected graph. Since $(G^\diamond)^*$ is 4-regular, $G^\diamond$ is quasi-simple and has bounded face-degree. Furthermore, any edge deletion or contraction in G that leaves G connected, corresponds to an edge insertion and immediate contraction in $G^\diamond$. Thus by Theorem 5.1 we can maintain a data structure for G under connectivity-preserving edge deletions and contractions, that after each update operation reports the corners that become part of a separating 4-cycle in $G^\diamond$.

We also note for any cycle in $G^\diamond$, the faces of $G^\diamond$ (the edges of G) lying on the opposite sides of the cycle belong to distinct separation classes (see Definition 5.2).

Lemma 5.9. *For any pair of edges in a biconnected graph G, their corresponding faces of $G^\diamond$ are separated by a 4-cycle (v_1, f_1, v_2, f_2) if and only if*

they belong to different separation classes with respect to v_1, v_2 in G and with respect to f_1, f_2 in G^*.

Proof. Let C be the 4-cycle. Consider a path L in G containing edges e_1 and e_2. Consider the set of faces F in $G^\diamond$ that are incident to a vertex on L. L does not cross v_1, v_2 if and only if F is completely contained on one side of C, which happens if and only if e_1 and e_2 are not separated by C. An identical argument can be made about f_1, f_2 in G^* □

Lemma 5.10. *Let G be a biconnected graph. If a 4-cycle $C = (v_1, f_1, v_2, f_2)$ in $G^\diamond$ is a separating cycle, then v_1, v_2 is a separation pair of G and f_1, f_2 is a separation pair of G^*.*

Proof. If C is a separating cycle, there is at least two faces on either side. By Lemma 5.9 there are thus at least two different separation classes with respect to v_1, v_2 (or f_1, f_2). If there exactly 2 separation classes, each consists of at least two edges. If there are exactly 3 classes, at least one of them consists of at least two edges. □

Lemma 5.11. *Let G be a loopless biconnected plane graph and u, w be a separation pair in G. Consider the set of edges E_x incident to $x \in \{u, w\}$. Then, the edges of E_x belonging to each separation class of u, w are consecutive in the circular ordering both around x.*

Proof. The proof is by contradiction. Assume that the circular order of some 4 edges incident to u is e_1, e_2, e_3, e_4. Moreover, assume that only e_1 and e_3 belong to the same separation class. From Definition 5.2 there is a path that begins with e_1 and ends with e_3 that does not contain u or w as its internal point. Thus, this path is a cycle C that does not go through w. Hence, every path from either e_2 or e_4, that ends in w and does not contain u as an internal point, has to go through C. This contradicts the fact that e_2 and e_4 are in different separation classes than e_1 and e_3. Clearly, the same argument applies to w. □

Lemma 5.12. *Let G be a loopless biconnected plane graph. Let F be a subset of edges of G, such that F is a separation class for some pair u, w of vertices of G. Then there exists a 4-cycle (possibly non-separating) in $G^\diamond$ that separates the set of faces that correspond to F from all other faces.*

Proof. Throughout the proof by separation class we mean one of separation classes defined by u and w. Clearly, each separation class has to have an edge incident to u or w (otherwise, since the graph is connected, it would

not be maximal). In fact, since G is biconnected, each separation class has edges incident both to u and w. If a separation class had edges incident only to one of the two vertices, this vertex would be an articulation point.

Denote the edges incident to u in circular order by $e_1, \ldots, e_k$. For convenience, let $e_{k+1} := e_1$ and $e_0 := e_k$. Moreover, assume that $e_i \in F$ iff $a \leqslant i \leqslant b$, where $1 \leqslant a \leqslant b \leqslant k$. Note that by Lemma 5.11, a and b are well-defined (for some way of breaking the circular ordering into a sequence $e_1, \ldots, e_k$).

Let f_1 be the face that comes in the circular order between e_{a-1} and e_a and f_2 be the face that comes between e_b and e_{b+1}. Note that $f_1 \neq f_2$.

We now show that there is a 4-cycle in $G^\diamond$ that contains u, w, f_1 and f_2. To that end, we prove that both u and w lie on f_1. Indeed, the cycle bounding f_1 is simple and contains edges from two separation classes. Thus, it has to contain both u and w. Similarly, both u and w lie on f_2.

This implies, that $G^\diamond$ contains a 4-cycle C_F going through u, w, f_1 and f_2, and by construction, C_F separates the faces corresponding to F from all other edges. □

Lemma 5.13. *Let G be loopless biconnected graph. If v_1, v_2 is a separation pair in G, then there exists a separation pair f_1, f_2 in G^* such that (v_1, f_1, v_2, f_2) is a separating cycle in $G^\diamond$.*

Proof. If every separation class with respect to v_1, v_2 consists of a single edge, then G consists of two vertices connected by multiple edges and the lemma is trivial. It suffices to use the fact that since v_1, v_2 is a separation pair, there are at least 4 edges in G. Otherwise there is a separation class F with at least two edges, such there are at least two edges in $E(G)\backslash F$. Now apply Lemma 5.12, to get a delimiting cycle C in $G^\diamond$ that separates faces corresponding to F from all other faces. Denote the vertices of C by v_1, f_1, v_2, f_2. Since both F and $E(G)\backslash F$ are nontrivial (both contain more than one edge), C is a separating cycle in $G^\diamond$. It then follows from Lemma 5.10 that f_1, f_2 form a separation pair in G^*. □

Lemma 5.14. *If G is triconnected, $e \in E[G]$, and x is an R-node in the SPQR-tree for $G - e$, then there exists a sequence of $\big|E[G]\big| - \big|E[\Gamma(x)]\big|$ edge deletions and contractions that transform $G - e$ into $\Gamma(x)$ while keeping the graph connected at all times.*

Proof. For each neighbor y of x in the SPQR-tree T we proceed as follows. Let a, b the separation pair corresponding to the edge in T between x and y. Consider all nodes reachable from y in T with a path that does not contain

x. Let D be the set of non-virtual edges in all these nodes. While there is an edge e in D that is not a self-loop and not an edge between a and b, contract it. Then if there are any self-loops delete them. When all edges in D go between a and b, delete edges until there is only one left. □

Lemma 5.15. *Let G be a triconnected plane graph and $uw \in E(G)$. Assume that $G - e$ is not triconnected. Then, the SPQR-tree of $G - e$ is a path H (we call it an SPQR-path). Moreover, given all edges that lie on 4-cycles in $(G - e)^\diamond$, we can compute all nodes of H except for the largest one in time that is linear in their size.*

Proof. Let us first prove that H is indeed a path. Since $G-e$ is biconnected, there exist two internally vertex-disjoint paths between u and w. No separation pair in $G - e$ can have both vertices on the same of these paths, since otherwise there would be a separation pair in G. Moreover, observe that each separation pair defines at most two separation classes that consist of more than one edge (otherwise, it is also a separation pair in G). Thus, we can split $G - e$ into two subgraphs by using an arbitrary separation pair in $G - e$. By repeating the same reasoning on both subgraphs, we get that H is a path. Observe that u and w belong to the nodes at the opposite ends of H.

Note that since we know the edges belonging to 4-cycles in $(G - e)^\diamond$, by using Lemma 5.9 we also know all separation pairs in $G-e$. We now describe how to compute all components of H (i.e. the skeleton graphs stored in the nodes) except the largest one. Consider an algorithm that starts from one end of H and discovers the components one by one, each in linear time. Observe that if each component of H has size at most $|E(G)|/2$, we can afford to detect all components of H, without affecting the total running time. However, to prepare for the opposite case, we need to do the search in parallel, starting from both ends of H. Let y be the largest component of H. Observe that only one search can start exploring edges of y. As soon as the other search reaches y, we have discovered all separation pairs (that is why we need to know all separation pairs upfront), and both searches can stop. Thus, one of the searches only uses time that is at most the total size of all non-largest components of H. Since the other search runs in parallel, it runs in the same asymptotic time.

To complete the proof it remains to describe how the search procedure works. Recall that by Lemma 5.11, for each separation pair u, w of $G-e$, the edges belonging to each separation class come in consecutive order around u and w. Observe that the edges of the 4-cycles of $(G - e)^\diamond$ correspond to

the corners of $G - e$ that lie between edges belonging to distinct separation classes. Thus, once we mark these corners in $G - e$, we can run a DFS-search that, once started from an edge belonging to a skeleton graph of an S- or R-node, explores all edges of this graph (and only those).

Observe moreover, that from our earlier analysis it follows that the endpoints u and w of e do not belong to any separation pair. This implies that both u and w are contained in S- or R- nodes. Let us fix on the search starting from u. Note that it discovers the entire component containing u and the separation pairs that separates it from the rest of the SPQR-tree. If the skeleton graph of this component is a path connecting the vertices of the separation pair, we have found an S-node. Otherwise, we have found an R-node.

Now assume we have found some prefix of the SPQR-path that ends at a separation pair a, b. If there is an edge ab (this edge comes next in the circular ordering after the edges we have visited, so it is easy to find), the next node on the SPQR-path is a P-node. After we have processed all edges between a and b, we insert a virtual edge between a and b and continue the search starting from this edge in a similar way to the search that has discovered the first node on the SPQR-path. Clearly, the algorithm runs in linear time. □

In the algorithm we maintain one SPQR-tree for each biconnected component with at least 3 edges. We now describe how these trees are updated upon edge deletions. The procedures, depending on the type of the SPQR-tree node are given as Algorithms 2, 3 and 4. Note that the lines 4 and 5 in Algorithm 3 only introduce notation, that is the values of the variables are not computed. We now show that the algorithms are correct. In each proof, the goal is to show that after the procedure the tree T satisfies Definition 5.3.

Lemma 5.16. *Algorithm 2 is correct.*

Proof. If after the edge deletion, $\Gamma(x)$ sill has at least 3 edges, then clearly T is a valid SPQR-tree. Otherwise, $\Gamma(x)$ has exactly two edges and we consider three cases. Recall that the number of virtual edges in a node is equal to the number of the node's neighbors in the SPQR-tree. If $\Gamma(x)$ has no virtual edges, then x is the only node of T, and thus this biconnected component now only has 2 edges, so we should delete the entire SPQR-tree. If $\Gamma(x)$ has exactly one virtual edge, x has exactly one neighbor. In this case, simply x represents one edge of the graph, so it has to be merged with its only neighbor and the virtual edge in the neighbor becomes non-virtual. If $\Gamma(x)$ has two virtual edges we remove x and the neighbors of x become neighbors.

Algorithm 2 Removing an edge e from a P-node x of T

```
1: function REMOVEP(e, x, T)
2:     remove e from Γ(x)
3:     if Γ(x) has two edges then
4:         if Γ(x) has no virtual edges then
5:             delete T
6:         else if Γ(x) has one virtual edge then
7:             y := the only neighbor of x
8:             e_x := the virtual edge in Γ(y) corresponding to x
9:             replace e_x by the non-virtual edge of Γ(x)
10:            remove x from T
11:        else if Γ(x) has two virtual edges then
12:            {y, z} := neighbors of x in T
13:            remove x from T, making y and z neighbors in T
14:            if y and z are S-nodes then
15:                merge y and z into one node
```

Note that the neighbors of x cannot be P-nodes. Thus, unless x has two neighboring S-nodes, we obtain a valid SPQR-tree. In the remaining case, it is easy to see that the two S-nodes can be merged into one S-node. □

Lemma 5.17. *Algorithm 3 is correct.*

Proof. If after removing the edge, $\Gamma(x)$ is triconnected, clearly the tree is a valid SPQR-tree. Otherwise, by Lemma 5.15, $\Gamma(x)$ is represented by a SPQR-path. It is easy to see that after replacing x by the SPQR-path X', we obtain a valid SPQR-tree, unless there are two neighboring S-nodes or P-nodes. Since the SPQR-path is a SPQR-tree, each such pair contains exactly one node z from the SPQR-path. If it is a P node, it can not be the end of the path, and so has at least 2 virtual edges to its neighbors on the path, and at most one more virtual edge to a neighbor z' outside the path. If it is an S-node it may have up to 2 virtual edges to a neighbor z' outside the path. Clearly, if z and z' have the same type, they can be merged into one node, and this yields a valid SPQR-tree. □

Lemma 5.18. *Algorithm 4 is correct.*

Proof. Observe that after removing an edge $e = uw$, each vertex of $\Gamma(x)$ distinct from u and w is an articulation point. Thus, each neighbor of x

Algorithm 3 Removing an edge e from an R-node x of T

```
1: function REMOVER(e, x, T)
2:     remove e from Γ(x)
3:     if Γ(x) has a separation pair then
4:         X' := SPQR-path representing Γ(x)
5:         x_big := the node of X', such that Γ(x_big) has the most edges
6:         compute all nodes of X', except x_big
7:         remove and contract edges of Γ(x) to obtain Γ(x_big)
8:         replace x in T by X' (connecting each child of x to the correct
   node of X')
9:         for each S- or P-node z ∈ X' do
10:            for each of the at most 2 neighbors z' of z outside X' do
11:                if z' has the same type as z then
12:                    merge z with z'
```

now belongs to a different biconnected component. Thus, we update T by deleting x, which breaks T into a piece for each neighbor y. For each piece we create a new BC-node z.

Each non-virtual edge of $\Gamma(x)$ becomes a biconnected component by itself, so we could simply ignore these edges from now on. For each virtual edge of $\Gamma(x)$, we delete the corresponding edge in the neighbor of x using an appropriate function. From Lemmas 5.16 and 5.17 it follows that the SPQR-trees are updated correctly. □

We can now prove the main theorem of this section. Note that, as in the block-cutpoint tree, we root each SPQR-tree in an arbitrary vertex.

Theorem 5.2. *There is a data structure that can be initialized on a simple planar graph G on n vertices in $O(n \log n)$ time, and supports any sequence of edge deletions or contractions in total time $O(n \log^2 n)$, while maintaining an explicit representation of a rooted SPQR-tree for each biconnected component with at least 3 edges, including all the skeleton graphs for the triconnected components. Moreover, in the process of handling updates, the total number of times a node of an SPQR-tree changes its parent is $O(n \log n)$.*

Proof. We first partition the graph into biconnected components, and, as sketched in Section 6.1.3, maintain the block-cutpoint tree explicitly. Thus, given two vertices u, v, we can in $O(1)$ time access the biconnected component containing both of them, along with its auxiliary data. Now, for each biconnected component C_i, we compute the SPQR-tree T. This can be done

Algorithm 4 Removing an edge e from an S-node x of T

```
1: function removeS(e, x, T)
2:     remove e from Γ(x)
3:     remove x from T
4:     for each edge e' in Γ(x) do
5:         Make a new BC-node z
6:         if e' is a virtual edge then
7:             y :=neighbor of x in T corresponding to e'
8:             Make the tree containing y the SPQR-tree for the new BC-
   node
9:             if y is a P-node then
10:                removeP(y, e', T)
11:            else
12:                removeR(y, e', T)
13:        else
```

in linear time due to [72]. We also root each SPQR-tree in an arbitrary node, and keep the trees rooted, as they are updated.

For each node x of T we maintain the graph $\Gamma(x)$. Each virtual edge of $\Gamma(x)$ has pointer to the neighbor of x it represents. Moreover, for each R-node r, we keep a data structure for detecting separating 4-cycles in the vertex-face graph $(\Gamma(r))^\diamond$, as described in Section 5.4. Any separating 4-cycle in that graph $(\Gamma(r))^\diamond$ corresponds to a separation pair in $\Gamma(r)$. Since the component is an R component, there are no separating 4-cycles to begin with, but some may appear after an update.

Since the total size of the R-components is n, it follows from Theorem 5.1 that the entire construction time is $O(n \log n)$.

Deletion. When an edge e is removed we find node x of the SPQR-tree, such that e is a non-virtual edge in x. Then, we proceed according to Algorithms 2, 3 and 4.

Whenever an edge fg is deleted from an R-node r, we update the corresponding 4-cycle detection structure for $(\Gamma(r))^\diamond$. We first insert the dual edge fg^* to the vertex-face graph, and then contract along that edge. This allows us to detect whether $\Gamma(r)$ has any separation pairs after each edge deletion.

Let us now analyze the running time. When processing an edge deletion, the following changes can take place in a SPQR-tree (all other changes can be handled in $O(1)$ time):

- an R-node is split into multiple nodes,
- two P-nodes or S-nodes are merged,
- an S- or P- node is deleted.

What is important, a P-node or an S-node can never get split. Hence, each edge of G can first belong to nodes that are split, but once it becomes a part of a P- or an S-node, its node can only be merged with other nodes.

Note that when two S- or P-nodes are merged, we can merge their skeleton graphs in constant time. These skeleton graphs have only two common nodes, and their lists of adjacent edges can be merged in constant time thanks to Lemma 5.11. When nodes are merged, we also have to update the parent pointers of their children. To bound the number of these updates, we merge the node with fewer children into the node with more. It follows that the number of parent updates caused by these merges is $O(n \log n)$, and so is the impact on the running time.

Similar analysis applies to the case when an R-node r is split into a SPQR-path. By Lemma 5.15 we can compute all but the largest node of the SPQR-path in linear time. Since the size of the skeleton graph in each of these nodes is at most half the size of $\Gamma(r)$, each edge takes part in this computation at most $O(\log n)$ times. For every new R-nodes, we also initialize their associated data structures for detecting 4-cycles. We charge the running time of each data structure to this initialization. From Theorem 5.1 we get that recomputing all the nodes and data structures takes $O(n \log^2 n)$ total time.

Taking care of the largest component of the SPQR-path is even easier, as we can simply reuse the skeleton graph of r and its associated data structure for detecting 4-cycles. To update the skeleton graph, we simply use Lemma 5.14.

After an R-node r is split into a SPQR-path H we also need to update the parent pointers in the children of r. However, the number of children to update is at most the number of edges in the non-largest components of the SPQR-path. As we have argued, the total number of such edges across all deletions is $O(n \log n)$.

Contraction. The contraction of an edge of the embedded planar graph G corresponds to the deletion of an edge of its dual graph, G^*. As observed by Angelini et al. [14, Lemma 1], the SPQR-tree of G^* is the dual SPQR-tree of G. That means, that if the edge belonged to a P-node of the SPQR-tree,

its contraction is handled like the deletion of an edge in a S-node, and vice versa.

If the contracted edge e belongs to an R-node, that R node may expand to a path in the SPQR-tree (because deletion in G^* may expand an R-node into a path). In the vertex-face graph, we may find all edges participating in new separating 4-cycles, corresponding to separating corners of the graph. Let (f, g) denote e^*. To find the new components, we simply apply Lemma 5.15 to the dual graph and proceed analogously to a deletion. □

5.6 Decremental Triconnectivity

To answer triconnectivity queries, we maintain a rooted SPQR-decomposition (see e.g. [36,72]) of each biconnected component of the planar graph.

Now it follows from the definition that pair of vertices in a biconnected graph are triconnected if and only if there exists a P or R component in the SPQR-tree containing them both. By associating a constant amount of information with every vertex in G and every node in the SPQR-tree, we can answer triconnectivity queries in constant time:

Definition 5.7. A *triconnectivity query structure* for a biconnected graph consists of a rooted SPQR-tree, and the following additional information:

- For each node x in the SPQR-tree except the root, a pointer $e(x)$ to the virtual edge that separates it from its parent.
- For each vertex v, a pointer $C(v)$ to the node containing v that is closest to the root.
- For each vertex v that points to an S-node x, a set $D(v)$ of pointers to the at most 2 children of x that contain v.

Lemma 5.19. *Given the triconnectivity query structure described in Definition 5.7, we can answer triconnectivity for any pair of vertices in constant time.*

Proof. Given vertices u and v. If $C(u) = C(v)$ and $C(u)$ is not an S-node, then u and v are triconnected. If $C(u) = C(v)$ is an S-node, $D(u) \cap D(v)$ is non-empty if and only if u and v are triconnected. If $C(u) \neq C(v)$, then u and v are triconnected if and only if either u is an endpoint of $e(C(v))$, or, v is an endpoint of $e(C(u))$. □

Given Theorem 5.1, we have the tools ready for maintaining triconnectivity:

Theorem 5.3. *There is a data structure that can be initialized on a planar graph G on n vertices in $O(n \log n)$ time, and supports any sequence of k edge deletions or contractions in total time $O((n+k)\log^2 n)$, while supporting queries to pairwise triconnectivity in worst-case constant time per query.*

Proof. For each vertex v and for each SPQR-node x, we associate the information $e(x), C(v), D(v)$ described in Definition 5.7.

Query. To answer a triconnected query (u, v), we first ask if (u, v) are biconnected. If not, they re not triconnected either. If they are, we get the SPQR-tree associated with their common biconnected component, and use Lemma 5.19. This answers the query in worst case constant time.

Updates. Our data structure for SPQR trees already maintain $e(x)$, so the main difficulty is in maintaining $C(v)$ and $D(v)$ for each vertex. Let x be the value of $C(v)$ before the change, let x' the new value, and suppose $x \neq x'$.

If x and x' are both R-nodes, $|E(\Gamma(x'))| < \frac{1}{2}|E(\Gamma(x))|$ so we are already using $\Omega(|E(\Gamma(x'))|) = \Omega(|V(\Gamma(x'))|)$ time to rebuild $\Gamma(x')^\diamond$. We can thus afford to update $C(v)$ for all $v \in V(\Gamma(x'))$.

If x is an R-node and x' is not, then $C(c)$ was split into $k > 1$ new nodes. In this case we are already using $\Omega(k)$ time maintaining the SPQR tree, so we can afford to use an additional $O(k)$ time on updating $C(v)$ for the at most $O(k)$ vertices from $V(\Gamma(x))$ whose new $C(v)$ is not an R-node.

If x is a P-node, then it has to be the root (since $C(v) = x$), so this can happen for at most 2 vertices per update and we can easily afford that.

If x is an S-node, then either the biconnected component was split into $k > 1$ new components and we can afford to spend $O(k)$ time on updating $C(v)$ for the $k-2$ vertices in S that were pointing to x. Or x was merged into another S-node. The total cost is linear in the total number of times some node changes parent due to such a merge, which is $O(n \log n)$.

Finally for each node x' that has a new parent p in the SPQR-tree, if p is an S-node, $e(x')$ has the two vertices whose $D(v)$ need to be changed, and this can be done in constant time. The total number of times this happens is $O(n \log n)$. □

5.7 Omitted Proofs from Section 5.4

Lemma 5.7. *The potential Φ is initially $O(n)$ and remains nonnegative. The embedding-respecting insertion or contraction of an edge uv decreases Φ by at least the number of candidate paths.*

Proof. To see the first statement, note that

$$1 \leqslant |K| \leqslant |M \cup K| \leqslant \Phi_v(M \cup K) \leqslant 4\,|M \cup K| \leqslant 4(2\,|K| - 2) = 8(|K| - 1)$$

where the inequality $|M \cup K| \leqslant \Phi_v(M \cup K)$, stems from $\Phi_v(M \cup K) - |M \cup K| = 3\,|V(G_{M\cup K})| - |E(G_{M\cup K})| \geqslant 0$, which is true by Observation 5.2. The last inequality is because there cannot be more than $|K| - 2$ vertices in M by Lemma 5.6. By a similar argument, we see that $\Phi_v(V) \geqslant 0$, and, being a sum of nonnegative terms, so is $\Phi_q(Y \cup M) \geqslant 0$.

We may thus realize that Φ is always positive:

$$\begin{aligned}
\Phi \geqslant \Phi_s(M \cup K) &\geqslant 63(\Phi_v(M \cup K))^2 - \sum_{v \in M \cup K} (d_{M\cup K}(v))^2 \\
&\geqslant 63\,|M \cup K|^2 - \left(\sum_{v \in M \cup K} d_{M\cup K}(v)\right)^2 \\
&\geqslant 63\,|M \cup K|^2 - (6\,|M \cup K| - 12)^2 \\
&= 27\,|M \cup K|^2 + 144(|M \cup K| - 1) \\
&\geqslant 0
\end{aligned}$$

Furthermore, note that Φ is initiated at $O(n)$. $\Phi_q(Y \cup M) \leqslant E$ which is $O(n)$ as the graph is planar. $\Phi_v(M \cup K) \leqslant 8|K| = O(\sqrt{n})$, and thus $(\Phi_v(M \cup K))^2 = O(n)$.

Before continuing with the second statement in the theorem, we consider how the different terms of Φ behave during changes.

First we observe, that $\Phi_q(Y \cup M)$ has the following properties when contractions occur:

1. $\Delta\Phi_q(Y \cup M) \leqslant -2$ when a pair of vertices in $Y \cup M$ are contracted.
2. $\Delta\Phi_q(Y \cup M) \leqslant -d_V(v)$ when a vertex $v \in Y \cup M$ is contracted with a vertex in K.
3. $\Delta\Phi_q(Y \cup M) \leqslant 0$ when a pair of vertices in K are contracted.

Furthermore, we observe that for any $X \subseteq V$, $\Phi_v(X)$ has the following properties when changes occur:

1. $\Delta\Phi_v(X) \leqslant 0$ when a vertex of degree $\geqslant 4$ is added to X.
2. $-4 \leqslant \Delta\Phi_v(X) \leqslant -1$ when a vertex degree $\leqslant 3$ is deleted from X.
3. $\Delta\Phi_v(X) = -1$ when an edge is added to G_X.
4. $-3 \leqslant \Delta\Phi_v(X) \leqslant -1$ when contracting any edge and reducing to a quasi-simple graph. ($\Delta\Phi_v(X) = -(3-\eta)$, where $0 \leqslant \eta \leqslant 2$ is the number of additional edges deleted).

All the statements have similar proofs, so take for instance statement 4. Here, we decrease the first term by 4 but increase the second by $\eta + 1$, and thus, the resulting change is between -3 and -1.

When an edge uv is inserted, it can only create new length-2 paths between vertices of K if at least one of its ends is in K. Suppose without loss of generality that $u \in K$. Then we have the following cases for where v is before the insertion:

$v \in Y$**:** Then v had at most 3 neighbors in K before uv was added, and thus at most 3 candidate paths need to be checked. In this case $\Phi_v(V)$ drops by one, $\Phi_q(Y \cup M)$ increases by one, and $\Phi_s(M \cup K)$ is unchanged. Thus Φ drops by 3.

$v \in M$**:** In this case there are $d_K(v)$ new candidate paths, $\Phi_v(V)$ drops by one and $\Phi_q(Y \cup M)$ increases by one just like before. However, now $\Phi_v(M \cup K)$ drops by one, so $\Phi_v(M \cup K)^2$ drops by $2\Phi_v(M \cup K) - 1$ and so $\Phi_s(M \cup K)$ drops by $63(2\Phi_v(M \cup K) - 1) - (2d_{M\cup K}(u) - 1) - (2d_{M\cup K}(v) - 1) \geqslant 63(2\,|M \cup K| - 1) - (2(6\,|M \cup K| - 12) - 1) - (2(6\,|M \cup K| - 12) - 1) \geqslant 102\,|M \cup K| + 225$, which is much larger than $d_K(v)$.

$v \in K$ In this case there are $d_K(u)+d_K(v)$ new candidate paths, $\Phi_v(V)$ drops by one and $\Phi_q(Y \cup M)$ is unchanged. However, as in the previous case $\Phi_v(M \cup K)$ drops by one so $\Phi_s(M \cup K)$ drops by at least $102\,|M \cup K| + 225$, which is much larger than $d_K(u) + d_K(v)$.

Finally we consider the case where an edge uv is contracted. To check the lemma, one simply has to check the different combinations of which partition the two elements and the result belong to:

(Y, Y) **merge to** Y**:** $\Phi_q(Y \cup M)$ drops by at least 2, and the other terms in the potential are unchanged, so Φ drops by at least 6. Since the result v is in Y, there are at most 3 paths of length 2 in $G_{K\cup\{v\}}$ with v as a middle vertex, and at most 2 of them are new.

(Y, Y) **merge to** M**:** The product of the degrees, and therefore the number of candidate paths is at most 9. $\Delta\Phi_v(M \cup K) \leqslant 0$ and $\Delta\Phi_q(Y \cup M) \leqslant -2$, and the term $\sum_{v \in M \cup K}(d_{M \cup K}(v))^2$ drops by the sum of degrees squared. Thus, $\Delta\Phi \leqslant -2 + 0 - 8 < -9$, and we are done.

(M, Y)**-merge** Suppose $u \in M$ and $v \in Y$. Let w be the node that u, v are contracted to, it will be an M-node.

We call an edge (v, k) important if it participates in a new length 2 path between different vertices of K. Each important edge incident to $v \in Y$ will become part of the graph quasi-induced by $M \cup K$, and therefore cause a drop in $\Phi_v(M \cup K)$. This drop is enough to pay for all new paths containing that edge.

(K, Y)**-merge** Suppose $u \in K$ and $v \in Y$. Let w be the node that u, v are contracted to, it will be an K-node.

There are two types of new length-two (K, K) paths that arise: Paths having w as the middle vertex, and paths having w as an end vertex.

The paths with w as a middle vertex are accounted for just like the previous case.

Each new path having w as an end vertex must have a neighbor of v as middle vertex. There are (less than) $d_V(v)$ of these neighbors. Since $\Delta\Phi_q(Y \cup M) \leqslant -d_V(v)$ we can afford to look at each of them, and pay for at most 2 new paths for each.

Now consider a neighbor m of v that is middle vertex of some new path. If $m \in Y$ (after the contraction), then there is at most 2 new paths involving m, and the drop in $\Phi_q(Y \cup M)$ pays for them. If $m \in M \cup K$, then $d_{M \cup K}(m)$ has increased, and $\Phi_s(M \cup K)$ drops appropriately.

(M, M)**-merge** Suppose $u, v \in M$ are merged to the new vertex w. Note that $w \in M$. Let $X = \{x_1, \ldots, x_k\}$ be the set of common neighbors of u, v in $G_{M \cup K}$ that lose an edge when quasi-simplifying after the contraction, and note that $0 \leqslant \eta \leqslant 2$. Let $\Phi_v = \Phi_v(M \cup K)$, then

$$\begin{aligned} \Delta(\Phi_v^2) &= (\Phi_v + \Delta\Phi_v)^2 - \Phi_v^2 \\ &= (\Phi_v - (3 - \eta))^2 - \Phi_v^2 \\ &= (3 - \eta)^2 - 2(3 - \eta)\Phi_v \end{aligned}$$

Let $a = d_{M \cup K}(u)$, $b = d_{M \cup K}(v)$, and for $i \in \{1, \ldots, \eta\}$ let $c_i =$

$d_{M\cup K}(x_i)$. Then

$$a, b, c_i \leqslant a + b + \sum_{i=1}^{\eta} c_i \leqslant \sum_{y\in M\cup K} d_{M\cup K}(y) \leqslant 6\,|M \cup K| - 12 \leqslant 6\Phi_v - 12$$

And finally

$$\begin{aligned}
\Delta\Phi_s(M \cup K) &\leqslant 63\Delta(\Phi_v^2) \\
&\quad - \left(\left((a+b-(\eta+2))^2 + \sum_{i=1}^{\eta}(c_i-1)^2\right)\right. \\
&\qquad \left. - \left(a^2 + b^2 + \sum_{i=1}^{\eta} c_i^2\right)\right) \\
&= 63((3-\eta)^2 - 2(3-\eta)\Phi_v) \\
&\quad - \left(2ab + (\eta+2)^2 - 2(\eta+2)a - 2(\eta+2)b\right. \\
&\qquad \left. - \sum_{i=1}^{\eta}(2c_i - 1)\right) \\
&= 63((3-\eta)^2 - 2(3-\eta)\Phi_v) \\
&\quad - 2ab - ((\eta+2)^2 + \eta) \\
&\quad + \left(2(\eta+2)a + 2(\eta+2)b + \sum_{i=1}^{\eta} 2c_i\right) \\
&\leqslant 63((3-\eta)^2 - 2(3-\eta)\Phi_v) \\
&\quad - 2ab - ((\eta+2)^2 + \eta) \\
&\quad + \left(2(\eta+2) + 2(\eta+2) + \sum_{i=1}^{\eta} 2\right)(6\Phi_v - 12) \\
&= 63((3-\eta)^2 - 2(3-\eta)\Phi_v) \\
&\quad - 2ab - ((\eta+2)^2 + \eta) + (6\eta+8)(6\Phi_v - 12) \\
&= -2ab + \begin{cases} 467 - 330\Phi_v & \text{if } \eta = 0 \\ 74 - 168\Phi_v & \text{if } \eta = 1 \\ -195 - 6\Phi_v & \text{if } \eta = 2 \end{cases} \\
&\leqslant -2ab - 6\Phi_v \qquad \text{for } \Phi_v \geqslant 2 \qquad (5.1)
\end{aligned}$$

In particular, $\Delta\Phi \leqslant -ab = -d_{M\cup K}(u)\cdot d_{M\cup K}(v) \leqslant -d_S(u)\cdot d_S(v)$, as desired.

(K, M)-merge: Suppose $u \in K$ and $v \in M$. Let w be the node that u, v are contracted to, it will be an K-node.

There are two types of new length-two (K, K) paths that arise: Paths having w as the middle vertex, and paths having w as an end vertex.

The paths with w as a middle vertex are accounted for just like the previous case.

Each new path having w as an end vertex must have a neighbor of v as middle vertex. There are (less than) $d_V(v)$ of these neighbors. Since $\Delta\Phi_q(Y \cup M) \leqslant -d_V(v)$ we can afford to look at each of them, and pay for at most 2 new paths for each.

Now consider a neighbor m of v that is middle vertex of some new path. If $m \in Y$ (after the contraction), then there is at most 2 new paths involving m, and the drop in Φ_q pays for them.

Let M be the set of neighbors of v in $M \cup K$ that is middle of some new path. The number of such paths is at most $|E(G_{M\cup K})| \leqslant 3\,|M \cup K| - 6 \leqslant 3\Phi_v(M \cup K) - 6$, since each must contain a unique edge from $G_{M\cup K}$. And the $-6\Phi_v(M \cup K)$ pays for them.

(K, K)-merge: Suppose $u, v \in K$. Let w be the node that u, v are contracted to, it will be an K-node.

There are two types of new length-two (K, K) paths that arise: Paths having w as the middle vertex, and paths having w as an end vertex.

The paths with w as a middle vertex are accounted for just like the previous two cases.

Each new path having w as an end vertex was already an (K, K) path with either u or v before the merge. For each of u, v there is at most $|K| - 1$ such pairs that (may) need to be updated, so the total cost of updating these is less than $2(|K| - 1) \leqslant 2\Phi_v$. The $-6\Phi_v$ can pay for these updates. □

Chapter 6

Online bipartite matching with amortized $\mathcal{O}(\log^2 n)$ replacements

Aaron Bernstein, Jacob Holm, Eva Rotenberg

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 known strategy 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.

6.1 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 [106]. 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 [28] 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 [19]. A more formal description is given in Section 6.6.1.

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 6.6.2

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 [41, 46], and in Hopcroft and Karp's algorithm for maximum matching in bipartite graphs [92].

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.

6.1.1 Previous work

The problem of online bipartite matchings with replacements was introduced in 1995 by Grove, Kao, Krishnan, and Vitter [66], 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 [28] 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 Zyck later analyzed the SAP protocol for forests, giving an upper bound of $\mathcal{O}(n \log^2 n)$ replacements [23], later improved to the optimal $\mathcal{O}(n \log n)$ total replacements [24]. 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. [22] 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 [92]).

The lower bound of $\Omega(\log n)$ by Grove et al. [66] has not been improved since, and is conjectured by Chadhuri et al. [28] 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, [70] and [19] 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 [98], and there have been several recent improvements for online Steiner tree with recourse [67,69,123,130]. There are many papers on online scheduling that try to minimize the number of job reassignments [13,52,140,146,149,162]. The model has also been studied in the context of flows [70,162], and there is a very recent result on online set cover with recourse [68].

6.1.2 Our results

Theorem 6.1. *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 [22], and is only a log factor from the lower bound of $\Omega(n \log n)$ by [66]. 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 6.3 and 6.4) 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 6.2. *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 algorithm by Hopcroft and Karp [92], which is matched by Bosek et al. [22] in the online setting.

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 6.6.1 to obtain:

Theorem 6.3. *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 6.4. *For any positive integers* n *and* $L \leqslant \sqrt{n/2}$ *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 6.5. *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.*

6.1.3 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 $\geqslant h$, then the current matching is at most $2n/h$ edges from optimal [92]. Thus

the algorithm augments down at most $2n/h$ augmenting paths of length $\geqslant h$. Let $P_1, P_2, ..., P_k$ denote all the paths augmented down by the algorithm in decreasing order of $|P_i|$; then $k \leqslant n$, and $|P_i| = h$ implies $i \leqslant 2n/h$. But then $|P_i| \leqslant 2n/i$, so $\sum_{1 \leqslant i \leqslant k} |P_i| \leqslant 2n \sum_{1 \leqslant i \leqslant k} \frac{1}{i} = 2n(\ln(k)+\mathcal{O}(1)) = \mathcal{O}(n \log k) = \mathcal{O}(n \log n)$.

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 6.1. *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 6.1. Note that the SAP protocol exactly follows the condition of Lemma 6.1. Now, Given any $0 \leqslant i \leqslant \log_2(n) + 1$, we say that an augmenting path is at level i if its length is in the interval $[2^i, 2^{i+1})$. By Lemma 6.1, 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 6.3 and 6.4 is devoted to proving Lemma 6.1. 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. [70] keeps track of changes to the "height" of vertices in M, while the algorithm with $\mathcal{O}(n\sqrt{n})$ reassignments [22] 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 [23,28,66]. 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.

6.1.4 Paper outline

In Section 6.2, we introduce the terminology necessary to understand the paper. In Section 6.3, we introduce and reason about the abstraction of a balanced server flow, a number that reflects the necessity of each server. In Section 6.4, we use the balanced server flow to prove Lemma 6.1, which proves our main theorem that SAP makes a total of $\mathcal{O}(n \log^2 n)$ replacements. In Section 6.5, we give an efficient implementation of SAP. Finally, in Section 6.6, we present our results on capacitated online assignment, and for minimizing maximum server load in the online assignment problem.

6.2 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 $C \neq \varnothing$. 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 6.6 (Halls Marriage Theorem [73])**.** *There is a matching of size $|C|$ if and only if $|K| \leqslant |N(K)|$ for all $K \subseteq C$.*

Definition 6.1. 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. [28] 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 6.1. *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 6.2. *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. □

6.3 The Server Flow Abstraction

We now formalize the notion of server necessities from Section 6.1.3 by using a flow-based notation.

Definition 6.2. 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.

Note 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 $|N(c)| \geqslant 1$ for all $c \in C$.

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

Lemma 6.3. *If* $\max_{\varnothing \subseteq K \subseteq C} \frac{|K|}{|N(K)|} = \frac{p}{q}$, *then there exists a server flow where every server* $s \in S$ *has* $\alpha(s) \leqslant \frac{p}{q}$.

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 $|K^*| \leqslant q\,|K| \leqslant p\,|N(K)| = |N(K^*)|$, 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

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

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, $\sum_{c \in N(s)} x_{cs} \leqslant \frac{p}{q}$. Thus, $(x_e)_{e \in E}$ realizes the desired server flow. □

Definition 6.3. We call the server flow α *balanced*, if additionally:

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

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 $K \subseteq C$, $A(K) = \bigcup_{c \in K} A(c)$.

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 6.4. *If* α *is a balanced server flow, then*

$$\forall T \subseteq S : \left|\{c \in C \mid A(c) \subseteq T\}\right| \leqslant \sum_{s \in T} \alpha(s) \leqslant \left|\{c \in C \mid A(c) \cap T \neq \varnothing\}\right|$$

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. □

We now show that the generalization of Hall's Marriage Theorem from Lemma 6.3 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 6.5. *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 $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \varnothing\}$ 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, $\widehat{K} \neq \varnothing$, 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 6.4

$$|\widehat{K}| = |K| \leqslant \widehat{\alpha}\,|\widehat{S}| \leqslant |\widehat{K}|\,. \qquad \square$$

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

Lemma 6.6. *Let α be a balanced server flow, and let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$. Then*

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

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 $K \subseteq C$, $|K| \leqslant \sum_{s \in N(K)} \alpha(s) \leqslant \widehat{\alpha}\,|N(K)|$, so $|K| \leqslant \widehat{\alpha}\,|N(K)|$. Let $\widehat{K}$ be defined as in Lemma 6.5. Then $\widehat{\alpha} = \frac{|\widehat{K}|}{|N(\widehat{K})|} \leqslant \max_{\varnothing \subseteq K \subseteq C} \frac{|K|}{|N(K)|} \leqslant \widehat{\alpha}$. Finally, if $|K| = \sum_{s \in N(K)} \alpha(s) = \widehat{\alpha}\,|N(K)|$ then $\alpha(s) \leqslant \widehat{\alpha}$ for all $s \in S$ implies $\alpha(s) = \widehat{\alpha}$ for $s \in N(K)$. □

Corollary 6.1. *If $\max_{\varnothing \subseteq K \subseteq C} \frac{|K|}{|N(K)|} = \frac{|C|}{|S|}$ there is a unique balanced server flow.*

Proof. By Lemma 6.3 there exists a server flow with $\alpha(s) \leqslant \frac{|C|}{|S|}$ 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 6.6. □

Lemma 6.7. *A unique balanced server flow exists if and only if* $|N(c)| \geqslant 1$ *for all* $c \in C$.

Proof. As already noted, $|N(c)| \geqslant 1$ 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 $\widehat{\alpha} = \max_{\varnothing \subseteq K \subseteq C} \frac{|K|}{|N(K)|}$ 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 two nonempty $K_1, K_2 \subseteq C$ we have $\widehat{\alpha} \geqslant \frac{|K_1 \cup K_2|}{|N(K_1) \cup N(K_2)|} \geqslant \min\left\{ \frac{|K_1|}{|N(K_1)|}, \frac{|K_2|}{|N(K_2)|} \right\}$, so $|\widehat{C}| = \widehat{\alpha}\,|N(\widehat{C})|$. If $N(\widehat{C}) = S$ then $\widehat{C} = C$ (otherwise $\frac{|\widehat{C}|}{|N(\widehat{C})|} < \frac{|C|}{|S|} \leqslant \widehat{\alpha}$) and by Corollary 6.1 we are done, so suppose $\varnothing \subseteq N(\widehat{C}) \subseteq S$. Consider the subgraph G_1 induced by $\widehat{C} \cup N(\widehat{C})$ and the subgraph G_2 induced by $(C \backslash \widehat{C}) \cup (S \backslash N(\widehat{C}))$.

By Corollary 6.1, 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 $C \backslash \widehat{C}$ and $N(\widehat{C})$) realize a balanced server flow for G.

For uniqueness, note that by Lemma 6.6 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 $s \in S \backslash N(\widehat{C})$, 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 $\widehat{K} = \{c \in C \mid A(c) \cap \widehat{S} \neq \varnothing\}$ 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 6.5, $|\widehat{K}| = \widehat{\alpha}\,|N(\widehat{K})|$ 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 6.6 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 $\widehat{K}$, $A(c) \cap \widehat{S} = \varnothing$ for all $c \in C \backslash \widehat{C}$. And there are clearly no edges between $\widehat{C}$ and $S \backslash N(\widehat{C})$. But then, for any $(x_e)_{e \in E}$ realizing α', the subset $(x_{cs})_{c \in C \backslash \widehat{C}, s \in S \backslash N(\widehat{C})}$ 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 $s \in S \backslash N(\widehat{C})$. □

From now on, let α denote the unique balanced server flow. We want to understand how the balanced server flow changes as vertices 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 $\Delta\alpha(s) = \alpha^{\text{NEW}}(s) - \alpha^{\text{OLD}}(s)$.

Lemma 6.8. *When a new client c is added, $\Delta\alpha(s) \geqslant 0$ 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 $S^* = \varnothing$. Say for contradiction that $S^* \neq \varnothing$, and let $\alpha^* = \min_{s \in S^*} \alpha^{\text{NEW}}(s)$. We will now partition S into three sets.

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

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

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, $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leqslant \left|C^\Delta\right|$. 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 $\alpha^{\text{OLD}}(v) \leqslant \alpha^* < \alpha^{\text{OLD}}(s)$ 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 $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) \geqslant \left|C^\Delta\right|$. This contradicts the earlier fact that $\sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s) \leqslant \left|C^\Delta\right|$, since by definition of S^Δ we have $\sum_{s \in S^\Delta} \alpha^{\text{NEW}}(s) < \sum_{s \in S^\Delta} \alpha^{\text{OLD}}(s)$. □

Lemma 6.9. *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 $S^+ = \left\{ s \in S \mid \alpha^{\text{OLD}}(s) \geqslant \min_{v \in N(c)} \alpha^{\text{OLD}}(v) \right\}$ and define $S^- = S \backslash S^+$. We want to show that $\Delta\alpha(s) = 0$ for all servers s in S^-.

Define C^+ to contain all client vertices incident to S^+; that is $C^+ = \{c \in C \mid N(c) \cap S^+ \neq \varnothing\}$. Let $C^- = C \backslash C^+$. Note that because the flow is balanced there are no edges in G from C^+ to S^- and there are no *active* edges from C^- to S^+ before the insertion of c. 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 $\sum_{s \in S^-} \alpha^{\text{NEW}}(s) \leqslant |C^-|$. But this means that $\sum_{s \in S^-} \Delta\alpha(s) \leqslant 0$, 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 6.8. □

6.4 Analyzing replacements in maximum matching

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

Lemma 6.10. *The graph $(C \cup S, E)$ contains a matching of size $|C|$, if and only if $\alpha(s) \leqslant 1$ for all $s \in S$.*

Proof. Let $\widehat{\alpha} = \max_{s \in S} \alpha(s)$. It follows directly from Lemma 6.6 that $|K| \leqslant |N(K)|$ for all $K \subseteq C$ if and only if $\widehat{\alpha} \leqslant 1$. The corollary then follows from Hall's Theorem (Theorem 6.6) □

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 6.1, 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 6.4. 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 6.2. *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 6.2). By Observation 6.1, 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 6.5. Given any $s \in S$, let $\alpha_M(s)$ be the flow into s in some balanced server flow in G_M; by Lemma 6.7 $\alpha_M(s)$ is uniquely defined.

Observation 6.3. *By construction G_M contains a matching of size $|C_M|$, so by Lemma 6.10 $\alpha_M(s) \leqslant 1$ for all $s \in S$. Finally, note that since $C_M \subseteq C$, we clearly have $\alpha_M(s) \leqslant \alpha(s)$*

Definition 6.6. 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 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 $s \in N(c)$.

Lemma 6.11 (Expansion Lemma)**.** *Suppose every client is matched, 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 $\alpha_M(s') \leqslant 1 - \epsilon$.

For $i \geqslant 1$, 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

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

Thus

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

Suppose there is no active augmenting tail from s of length $\leqslant 2(i-1)$, then every server in $A(K_i)$ is matched, and the clients they are matched to are exactly K_{i+1}. So $k_{i+1} = |A(K_i)|$ and thus $|C_M| \geqslant k_{i+1} \geqslant \frac{1}{1-\epsilon} k_i \geqslant (\frac{1}{1-\epsilon})^i k_1 = (\frac{1}{1-\epsilon})^i$. It follows that $i \leqslant \frac{\ln |C_M|}{\ln \frac{1}{1-\epsilon}} \leqslant \frac{1}{\epsilon} \ln |C_M|$, where the last inequality follows from $1 - \epsilon \leqslant e^{-\epsilon}$. 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 6.1.3 implies Theorem 6.1.

Lemma 6.1. *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 $n = |C| \geqslant |C_M|$. The lemma clearly holds for $h \leqslant 4\ln(n)$ 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 6.2 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 $|C^*| \leqslant 4n\ln(n)/h$. 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 6.11, each server $s \in N(c)$ has $\alpha_M(s) \geqslant 1 - 2\ln(n)/h$. Let S^* be the set of all servers that at some point have $\alpha_M(s) \geqslant 1 - 2\ln(n)/h$; by Lemma 6.8, this is exactly the set of servers s such that $\alpha_M(s) \geqslant 1 - 2\ln(n)/h$ after all clients have been inserted. By Lemma 6.9, 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 each client contributes one unit of flow, and by Observation 6.3 $\alpha_M(s) \leqslant 1$ for all $s \in S$, the total number of such clients is $|C^*| \leqslant (2\log(n)/h)\,|S^*|$. 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 $n \geqslant |C_M| \geqslant (1 - 2\ln(n)/h)\,|S^*| > |S^*|/2$, where the last inequality follows from the assumption that $h > 4\ln(n)$. □

6.5 Implementation

In the previous section we proved that augmenting along a shortest 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 producing the matchings. In this section, we show an efficient implementation for finding the augmenting paths quickly, and thus maintaining 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 [92].

Definition 6.7. Define the height of a vertex v (server or client) to be the length of the shortest augmenting tail (Definition 6.6) from v. If no augmenting tail exists, we set the height to $2n$.

Observation 6.4. *Heights are monotonically increasing. This follows directly from standard arguments about shortest augmenting paths. For a formal proof see Lemma 5.3 in [70].*

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 paths of length at most $\sqrt{n}\sqrt{\ln(n)}$. We will find longer augmenting paths using the trivial $\mathcal{O}(m)$ algorithm, and use Lemma 6.1 to bound the number of such paths. Our analysis will also require the following lemma:

Lemma 6.12. *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 6.11. 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 6.3), Lemma 6.5 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.

Theorem 6.2. *There is an implementation of the SAP protocol that runs in total time $\mathcal{O}(m\sqrt{n}\sqrt{\log n})$.*

Proof. Every vertex will keep track of its height up to $h = \sqrt{n}\sqrt{\log n}$. If its height is larger than h, it will simply mark itself as a *high* vertex. Each vertex v will also track the height of each of its neighbors with $\mathcal{O}(h)$ buckets: $N_1(v), N_2(v), N_3(v), \ldots, N_h(v)$, and $N^*(v)$ for the high neighbors. Whenever a neighbor u of v changes height, we will have to change the corresponding bucket of v. Every vertex v will also keep track of its lowest non-empty bucket.

When a new client c arrives, it will use the height information to find a shortest augmenting path. First off, put all of the neighbors of c in their corresponding buckets: this takes $\mathcal{O}(\deg(c))$ time. Now to find the shortest augmenting path from c we consider two cases. The first is that c has a neighbor who is not high. In this case, take the edge from c to its neighbor with minimum height by picking an arbitrary neighbor from the lowest non-empty bucket. Now follow the backwards (matched) edge to a new vertex

c_1, and again, take the edge to the vertex of minimum height, breaking ties arbitrarily. This will obviously compute a shortest augmenting path.

The second case is when all the neighbors of c are high. In this case we can just brute-force compute a shortest augmenting path in time $\mathcal{O}(m)$. If we find an augmenting path, we augment down it; *if we do not find an augmenting path, then we remove all servers and clients encountered during the search for this augmenting path, and continue the algorithm in the graph with these vertices removed.*

correctness We want to show that our implementation chooses a shortest augmenting path at every step. Although our implementation clearly always chooses the shortest augmenting path in the remaining graph, there is the potential problem that this remaining graph might not be the original graph, since the algorithm sometimes deletes vertices from the graph due to a failed brute-force search. We show that this is not a problem because whenever our implementation deletes a vertex v at time t, then in the original graph there is never an augmenting path through v after time t. 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 6.12, this implies that $\alpha_M(s) = 1$. But then by Lemma 6.8 we have $\alpha_M(s) = 1$ for all future client insertions as well. (Recall that by Observation 6.3 we never have $\alpha_M(s) > 1$.) Thus by Lemma 6.12 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 paths 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 maintain the height buckets
2. the time to brute-force compute augmenting paths when all the neighbors of an incoming client are high vertices.
3. the time to follow the augmenting paths given the height buckets.

Item 3 takes $\mathcal{O}(n \log^2 n)$ time because we need $\mathcal{O}(1)$ time to follow each edge in the path, and by Theorem 6.1 the total length of augmenting paths is $\mathcal{O}(n \log^2 n)$. For Item 2 we consider two cases. The first is brute-force searches which result in finding an augmenting path. These take a total of $\mathcal{O}(mn \log(n)/h) = \mathcal{O}(m\sqrt{n}\sqrt{\log n})$ time because by Lemma 6.1 during the

course of the entire algorithm there are at most $\mathcal{O}(n \log(n)/h)$ augmenting paths of length $\geqslant h$, 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.

Finally, For Item 1, we observe that if augmenting along a path causes a vertex v to change height, then either v belongs to that augmenting path, or v has a neighbor whose height changes. Thus, to maintain the height buckets it is enough for each vertex v to inform all neighbors w whenever the height of v changes. In particular each neighbor w changes the location of v in bucket structure in $\mathcal{O}(1)$ time and then checks in $\mathcal{O}(1)$ time whether its height changed by looking in its lowest non-empty bucket; the lowest non-empty bucket of w is easy to keep track of because heights never decrease (Observation 6.4), so vertices only move up the bucket structure of w. If the height of w did not change then w does no further work. Otherwise repeat from w, i.e. update the buckets of all of the neighbors of w. All in all we do $\mathcal{O}(\deg(v))$ work whenever the height of a vertex v changes. Since heights never decrease, this can happen at most h times per vertex v, leading to a total time of $\mathcal{O}(mh) = m\sqrt{n}\sqrt{\log n}$. □

6.6 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.

6.6.1 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 6.3. *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 [19]. 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 6.1 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.) □

6.6.2 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 6.8. 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.

[70] and [19] 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, $\ell(\mathcal{A}_t) \leqslant 8\textsc{opt}(G_t)$. 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 6.4. *For any positive integers n and $L \leqslant \sqrt{n/2}$ divisible by 4 there exists a graph $G = (C \cup S, E)$ with $|C| = n$ and $\text{OPT}(G) = L$, 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 6.13. *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 $|C| = L^2$, $|S| = L$, $\text{OPT}(G) = L$, 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 $\text{OPT}(G) = L$.

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 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

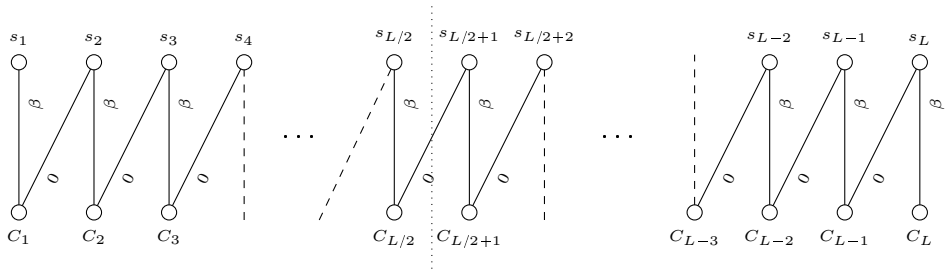

Figure 6.1: Number of assignments of each type after first $L/2$ clients added to each block, and after each up-heavy phase. Each C_i has β clients. Each server has β clients assigned.

$\text{OPT}(G^{\text{OLD}}) = \beta$, 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 6.1).

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 $\beta \geqslant L/2$ and consider the following assignment $\mathcal{A}^{\text{NEW}}$: for $i \leqslant L/2$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + 2 - i \geqslant 2$ clients from C_i to s_i and i clients in C_i to s_{i+1}; for $L/2 < i \leqslant L$, $\mathcal{A}^{\text{NEW}}$ assigns $\beta + i - L \geqslant 0$ 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 6.2).

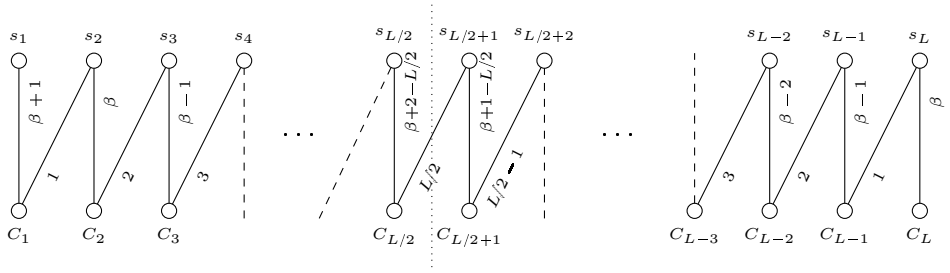

Figure 6.2: Number of assignments of each type after each down-heavy phase. Each C_i has $\beta + 2$ clients for $1 \leqslant i \leqslant L/2$ and β clients for $L/2 + 1 \leqslant i \leqslant L$. Each server has $\beta + 1$ clients assigned.

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 $i \leqslant \beta/2$, $\mathcal{A}$ 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 $L/2 < i \leqslant L$, $\mathcal{A}$ 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 $L/2 < i \leqslant L$. 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 $\sum_{L/2<i\leqslant L}(L-i) = \Omega(L^2)$. 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 6.4. Recall the assumption of the Theorem that $n/2 \geqslant L^2$. Simply let the graph G consist of $\lfloor n/L^2 \rfloor$ separate instances of the graph in Lemma 6.13, 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 [19]. 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 6.1, 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 6.5. *For the uncapacitated problem of online maximum matching with replacements, let us say that instead of starting with $C = \varnothing$, the algorithm starts with some initial set of clients $C_0 \subseteq C$ 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 6.1, 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 6.1 follows directly from the key Lemma 6.1, which in term follows from the expansion argument in Lemma 6.11. 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 6.5. *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 6.3 from [19]

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 6.5 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 $L \geqslant \sqrt{n}/\log n$, 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. □

6.7 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. [66]. 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 [22]. 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. [28] 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 [92].

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 6.1 from $\mathcal{O}(n \ln(n)/h)$ to $\mathcal{O}(n/h)$. (In the implementation of Section 6.5 we would then set $h = \sqrt{n}$ instead of $h = \sqrt{n}\sqrt{\log n}$.)

Chapter 7

Bibliography

[1] Aamand, A., Hjuler, N., Holm, J., and Rotenberg, E. One-way trail orientations. *ArXiv e-prints arXiv:1708.07389* (Aug. 2017). In submission.

[2] Abboud, A., and Dahlgaard, S. Popular conjectures as a barrier for dynamic planar graph algorithms. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA* (2016), pp. 477–486.

[3] Abboud, A., and Williams, V. V. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014* (2014), pp. 434–443.

[4] Abraham, I., Chechik, S., and Gavoille, C. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012* (2012), pp. 1199–1218.

[5] Abrahamsen, M., Bodwin, G., Rotenberg, E., and Stöckel, M. Graph reconstruction with a betweenness oracle. In *33rd Symposium on Theoretical Aspects of Computer Science (STACS 2016)* (2016), N. Ollinger and H. Vollmer, Eds., Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 5:1–5:14.

[6] Abrahamsen, M., Holm, J., Rotenberg, E., and Wulff-Nilsen, C. Best Laid Plans of Lions and Men. In *33rd International Symposium on Computational Geometry (SoCG 2017)* (2017),

B. Aronov and M. J. Katz, Eds., vol. 77 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16.

[7] Alstrup, S., Gavoille, C., Halvorsen, E. B., and Petersen, H. Simpler, faster and shorter labels for distances in graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms, SODA* (2016), pp. 338–350.

[8] Alstrup, S., Georgakopoulos, A., Rotenberg, E., and Thomassen, C. A Hamiltonian cycle in the square of a 2-connected graph in linear time. In submission.

[9] Alstrup, S., and Holm, J. Improved algorithms for finding level ancestors in dynamic trees. In *ICALP* (2000), U. Montanari, J. D. P. Rolim, and E. Welzl, Eds., vol. 1853 of *Lecture Notes in Computer Science*, Springer, pp. 73–84.

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

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

[12] Andersson, A., Hagerup, T., Nilsson, S., and Raman, R. Sorting in linear time? *Journal of Computer and System Sciences 57*, 1 (1998), 74–93. Announced at STOC'95.

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

[14] Angelini, P., Bläsius, T., and Rutter, I. Testing mutual duality of planar graphs. *Int. J. Comput. Geometry Appl. 24*, 4 (2014), 325–346.

[15] Archdeacon, D., and Richter, R. B. The construction and classification of self-dual spherical polyhedra. *Journal of Combinatorial Theory, Series B 54*, 1 (1992), 37 – 63.

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

[17] Baswana, S., Gupta, M., and Sen, S. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM Journal on Computing 44*, 1 (2015), 88–113.

[18] Bernstein, A., Holm, J., and Rotenberg, E. Online bipartite matching with amortized $O(\log^2 n)$ replacements. *ArXiv e-prints arXiv:1707.06063* (July 2017). In submission.

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

[20] Biedl, T. C., Bose, P., Demaine, E. D., and Lubiw, A. Efficient algorithms for petersen's matching theorem. *Journal of Algorithms 38*, 1 (2001), 110 – 134.

[21] Bondy, J., and Murty, U. *Graph Theory with Applications: By J.A. Bondy and U.S.R. Murty*. Macmillan, 1976.

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

[23] Bosek, B., Leniowski, D., Sankowski, P., and Zych, A. 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* (Cham, 2015), Springer International Publishing, pp. 59–71.

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

[25] Brightwell, G. R., and Scheinerman, E. R. Representations of planar graphs. *SIAM Journal on Discrete Mathematics 6*, 2 (1993), 214–229.

[26] Brinkmann, G., Greenberg, S., Greenhill, C., Mckay, B. D., Thomas, R., and Wollan, P. Generation of simple quadrangulations of the sphere. *Discrete Math. 305*, 1-3 (Dec. 2005), 33–54.

[27] Chartrand, G., Hobbs, A. M., Jung, H. A., Kapoor, S. F., and Nash-Williams, C. The square of a block is hamiltonian connected. *Journal of Combinatorial Theory, Series B 16*, 3 (1974), 290 – 292.

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

[29] Chechik, S., Hansen, T. D., Italiano, G. F., Łącki, J., and Parotsidis, N. Decremental single-source reachability and strongly connected components in $\widetilde{O}(m\sqrt{n})$ total update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA* (2016), pp. 315–324.

[30] Chen, D., and Xu, J. Shortest path queries in planar graphs. In *STOC '00* (2000), pp. 469–478.

[31] Cohen-Addad, V., de Mesmay, A., Rotenberg, E., and Roytman, A. The bane of low-dimensionality clustering. In submission., 2017.

[32] Dahlgaard, S., Knudsen, M. B. T., Rotenberg, E., and Thorup, M. Hashing for statistics over k-partitions. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS'15* (2015), IEEE, pp. 1292–1310.

[33] Dahlgaard, S., Knudsen, M. B. T., Rotenberg, E., and Thorup, M. The power of two choices with simple tabulation. In *SODA'16* (2016), SIAM, pp. 1631–1642.

[34] Demetrescu, C., and Italiano, G. F. A new approach to dynamic all pairs shortest paths. *J. ACM 51*, 6 (2004), 968–992.

[35] Demetrescu, C., and Italiano, G. F. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica 51*, 4 (2008), 387–427.

[36] Di Battista, G., and Tamassia, R. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica 15*, 4 (1996), 302–318.

[37] Di Battista, G., and Tamassia, R. On-line planarity testing. *SIAM Journal on Computing 25*, 5 (1996), 956–997.

[38] Diestel, R. *Graph Theory* (4th edition). Springer-Verlag, 2010.

[39] Diks, K., and Sankowski, P. Dynamic plane transitive closure. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings* (2007), pp. 594–604.

[40] Diks, K., and Stanczyk, P. *Perfect Matching for Biconnected Cubic Graphs in $O(n \log^2 n)$ Time*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 321–333.

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

[42] Dirac, G. A. Minimally 2-connected graphs. *J. Reine Angew. Math. 228* (1967), 204–216.

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

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

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

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

[47] Eppstein, D., Galil, Z., and Italiano, G. F. Improved sparsification. Tech. rep., 1993.

[48] Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. Sparsification – A technique for speeding up dynamic graph algorithms. *J. ACM 44*, 5 (Sept. 1997), 669–696. Announced at FOCS '92.

[49] Eppstein, D., Galil, Z., Italiano, G. F., and Spencer, T. H. Separator based sparsification I: Planarity testing and minimum spanning trees. *Journal of Computer and System Sciences 52*, 1 (1996), 3–27.

[50] Eppstein, D., Galil, Z., Italiano, G. F., and Spencer, T. H. Separator-based sparsification II: Edge and vertex connectivity. *SIAM Journal on Computing 28*, 1 (1998), 341–381. Announced at STOC '93.

[51] Eppstein, D., Italiano, G. F., Tamassia, R., Tarjan, R. E., Westbrook, J., and Yung, M. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms 13*, 1 (1992), 33–54.

[52] Epstein, L., and Levin, A. *Robust Algorithms for Preemptive Scheduling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 567–578.

[53] Fakcharoenphol, J., and Rao, S. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences 72*, 5 (2006), 868–889.

[54] Fleischner, H. The square of every two-connected graph is hamiltonian. *Journal of Combinatorial Theory, Series B 16*, 1 (1974), 29 – 34.

[55] Fleischner, H. In the square of graphs, hamiltonicity and pancyclicity, hamiltonian connectedness and panconnectedness are equivalent concepts. *Monatshefte für Mathematik 82*, 2 (1976), 125–149.

[56] Frederickson, G. N. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing 14*, 4 (1985), 781–798.

[57] Frederickson, G. N. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM Journal on Computing 26*, 2 (1997), 484–538.

[58] Gabow, H. N., Kaplan, H., and Tarjan, R. E. Unique maximum matching algorithms. *J. Algorithms 40*, 2 (2001), 159–183. Announced at STOC '99.

[59] Galil, Z., Italiano, G. F., and Sarnak, N. Fully dynamic planarity testing with applications. *J. ACM 46* (1999), 28–91.

[60] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[61] Georgakopoulos, A. Infinite hamilton cycles in squares of locally finite graphs. *Advances in Mathematics 220*, 3 (2009), 670 – 705.

[62] Georgakopoulos, A. A short proof of fleischner's theorem. *Discrete Math. 309*, 23-24 (Dec. 2009), 6632–6634.

[63] Giammarresi, D., and Italiano, G. F. Decremental 2- and 3-connectivity on planar graphs. *Algorithmica 16*, 3 (1996), 263–287. Announced at SWAT 1992.

[64] Gibb, D., Kapron, B. M., King, V., and Thorn, N. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR abs/1509.06464* (2015).

[65] Goodrich, M. T. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences 51*, 3 (1995), 374–389.

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

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

[68] Gupta, A., Krishnaswamy, R., Kumar, A., and Panigrahi, D. Online and dynamic algorithms for set cover. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing* (New York, NY, USA, 2017), STOC 2017, ACM, pp. 537–550.

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

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

[71] Gustedt, J. Efficient union-find for planar graphs and other sparse graph classes. *Theor. Comput. Sci. 203*, 1 (1998), 123–141.

[72] Gutwenger, C., and Mutzel, P. *A Linear Time Implementation of SPQR-Trees.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 77–90.

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

[74] Han, X., Kelsen, P., Ramachandran, V., and Tarjan, R. Computing minimal spanning subgraphs in linear time. *SIAM Journal on Computing 24*, 6 (1995), 1332–1358.

[75] Harary, F. *Graph Theory.* Addison-Wesley Series in Mathematics. Addison Wesley, 1969.

[76] Harel, D., and Tarjan, R. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing 13*, 2 (1984), 338–355.

[77] Henzinger, M. R., and King, V. Fully dynamic 2-edge connectivity algorithm in polylogarithmic time per operation, 1997.

[78] Henzinger, M. R., and King, V. Maintaining minimum spanning trees in dynamic graphs. In *Automata, Languages and Programming: 24th International Colloquium, ICALP '97 Bologna, Italy, July 7–11, 1997 Proceedings* (Berlin, Heidelberg, 1997), P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds., Springer Berlin Heidelberg, pp. 594–604.

[79] Henzinger, M. R., and King, V. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM 46*, 4 (July 1999), 502–516. Announced at STOC '95.

[80] Henzinger, M. R., and Thorup, M. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms 11*, 4 (1997), 369–379.

[81] Hierholzer, C., and Wiener, C. Ueber die möglichkeit, einen linienzug ohne wiederholung und ohne unterbrechung zu umfahren. *Mathematische Annalen 6*, 1 (1873), 30–32.

[82] Hobbs, A. M. The square of a block is vertex pancyclic. *Journal of Combinatorial Theory, Series B 20*, 1 (1976), 1 – 4.

[83] Holm, J., de Lichtenberg, K., and Thorup, M. 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* (New York, NY, USA, 1998), STOC '98, ACM, pp. 79–89.

[84] Holm, J., de Lichtenberg, K., and Thorup, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM 48*, 4 (July 2001), 723–760.

[85] Holm, J., Italiano, G. F., Karczmarz, A., Łącki, J., and Rotenberg, E. Decremental SPQR-trees for planar graphs. In submission, 2017.

[86] Holm, J., Italiano, G. F., Karczmarz, A., Łącki, J., Rotenberg, E., and Sankowski, P. Contracting a Planar Graph Efficiently. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria* (Sept. 2017).

[87] Holm, J., and Rotenberg, E. Dynamic planar embeddings of dynamic graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany* (2015), E. W. Mayr and N. Ollinger, Eds., vol. 30 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 434–446.

[88] Holm, J., and Rotenberg, E. Dynamic planar embeddings of dynamic graphs. *Theory of Computing Systems* (2017), 1–30.

[89] Holm, J., Rotenberg, E., and Thorup, M. Planar reachability in linear space and constant time. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS'15* (Oct 2015), pp. 370–389.

[90] Holm, J., Rotenberg, E., and Thorup, M. Dynamic bridge-finding in $\tilde{O}(\log^2 n)$ amortized time. *arXiv preprint arXiv:1707.06311* (2017). In submission.

[91] Holm, J., Rotenberg, E., and Wulff-Nilsen, C. Faster fully-dynamic minimum spanning forest. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, Sept. 14-16, 2015, Proceedings* (2015), pp. 742–753.

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

[93] Hopcroft, J. E., and Tarjan, R. E. Dividing a graph into triconnected components. *SIAM Journal on Computing 2*, 3 (1973), 135–158.

[94] Hopcroft, J. E., and Tarjan, R. E. A V log V algorithm for isomorphism of triconnected planar graphs. *Journal of Computer and System Sciences 7*, 3 (1973), 323–331.

[95] Hopcroft, J. E., and Tarjan, R. E. Efficient planarity testing. *J. ACM 21*, 4 (1974), 549–568.

[96] Hopcroft, J. E., and Wong, J. K. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA* (1974), pp. 172–184.

[97] Huang, S.-E., Huang, D., Kopelowitz, T., and Pettie, S. 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* (Philadelphia, PA, USA, 2017), SODA '17, Society for Industrial and Applied Mathematics, pp. 510–520.

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

[99] Italiano, G. F., Karczmarz, A., Łącki, J., and Sankowski, P. Decremental single-source reachability in planar digraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017* (2017), pp. 1108–1121.

[100] Italiano, G. F., La Poutré, J. A., and Rauch, M. H. *Fully dynamic planarity testing in planar embedded graphs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 212–223.

[101] Italiano, G. F., Nussbaum, Y., Sankowski, P., and Wulff-Nilsen, C. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011* (2011), pp. 313–322.

[102] Kant, G. Algorithms for drawing planar graphs, 2001.

[103] Kaplan, H., Mozes, S., Nussbaum, Y., and Sharir, M. Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012* (2012), pp. 338–355.

[104] Kapron, B. M., King, V., and Mountjoy, B. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2013), SODA '13, Society for Industrial and Applied Mathematics, pp. 1131–1142.

[105] Karp, R. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum Press, 1972, pp. 85–103.

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

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

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

[109] Kejlberg-Rasmussen, C., Kopelowitz, T., Pettie, S., and Thorup, M. Faster worst case deterministic dynamic connectivity. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark* (2016), pp. 53:1–53:15.

[110] Kernighan, B., and Ritchie, D. *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[111] King, V. 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* (1999), pp. 81–91.

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

[113] Klein, P. N. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, BC, Canada, January 23-25, 2005* (2005), pp. 146–155.

[114] Klein, P. N., and Mozes, S. Optimization algorithms for planar graphs, 2017.

[115] Klein, P. N., Mozes, S., and Sommer, C. Structured recursive separator decompositions for planar graphs in linear time. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013* (2013), pp. 505–514.

[116] Knuth, D. E. Two notes on notation. *The American Mathematical Monthly 99*, 5 (1992), 403–422.

[117] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[118] Kotzig, A. *On the theory of finite graphs with a linear factor II.* 1959.

[119] Kriesell, M. Minimal connectivity. In *Topics in Structural Graph Theory, edited by L. Beineke and R. Wilson* (2012), Cambridge University Press.

[120] La Poutré, J. A. Alpha-algorithms for incremental planarity testing (preliminary version). In *STOC '94* (1994), ACM, pp. 706–715.

[121] Lau, H. T. *Finding a Hamiltonian Cycle in the Square of a Block.* PhD thesis, McGill University, 1980.

[122] Łącki, J. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms 9*, 3 (2013), 27:1–27:15.

[123] Łącki, J., Oćwieja, J., Pilipczuk, M., Sankowski, P., and Zych, A. 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* (2015), pp. 11–20.

[124] Łącki, J., and Sankowski, P. Min-cuts and shortest cycles in planar graphs in $O(n \log \log n)$ time. In *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings* (2011), pp. 155–166.

[125] Łącki, J., and Sankowski, P. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany* (2015), pp. 608–621.

[126] Łącki, J., and Sankowski, P. Optimal decremental connectivity in planar graphs. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany* (2015), pp. 608–621.

[127] Lipton, R. J., and Tarjan, R. E. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics 36*, 2 (1979), 177–189.

[128] Mac Lane, S. A structural characterization of planar combinatorial graphs. *Duke Math. J. 3*, 3 (09 1937), 460–472.

[129] Mader, W. Ecken vom Grad n in minimalen n-fach zusammenhängenden Graphen. *Arch. Math. (Basel) 23* (1972), 219––224.

[130] Megow, N., Skutella, M., Verschae, J., and Wiese, A. The power of recourse for online mst and tsp. *SIAM Journal on Computing 45*, 3 (2016), 859–880.

[131] Menger, K. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae 10* (1927), 96–115.

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

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

[134] Nanongkai, D., and Saranurak, T. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and O($n^{1/2 - \epsilon}$)-time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017* (2017), pp. 1122–1129.

[135] Nanongkai, D., Saranurak, T., and Wulff-Nilsen, C. Dynamic minimum spanning forest with subpolynomial worst-case update time. *arXiv preprint arXiv:1708.03962* (2017).

[136] Nanongkai, D., Saranurak, T., and Wulff-Nilsen, C. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *Proceedings of the 58th Annual Symposium on Foundations of Computer Science, FOCS 2017* (2017). To appear.

[137] Parker, R. G., and Rardin, R. L. Guaranteed performance heuristics for the bottleneck travelling salesman problem. *Operations Research Letters 2*, 6 (1984), 269–272.

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

[139] Petersen, J. Die Theorie der regulären graphs. *Acta Math. 15* (1891), 193–220.

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

[141] Pătraşcu, M., and Demaine, E. D. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing 35*, 4 (2006), 932–963.

[142] Raman, R. Fast algorithms for shortest paths and sorting, 1996.

[143] Robbins, H. E. A theorem on graphs, with an application to a problem of traffic control. *The American Mathematical Monthly 46*, 5 (1939), 281–283.

[144] Roditty, L., and Zwick, U. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing 37*, 5 (2008), 1455–1471.

[145] Rosenstiehl, P. Embedding in the plane with orientation constraints: The angle graph. *Annals of the New York Academy of Sciences 555*, 1 (1989), 340–346.

[146] Sanders, P., Sivadasan, N., and Skutella, M. Online scheduling with bounded migration. *Math. Oper. Res. 34*, 2 (2009), 481–498.

[147] Sankowski, P. Dynamic transitive closure via dynamic matrix inverse (extended abstract). In *45th Symposium on Foundations of Computer Science FOCS 2004, 17-19 October 2004, Rome, Italy, Proceedings* (2004), pp. 509–517.

[148] Schmidt, J. M. A simple test on 2-vertex- and 2-edge-connectivity. *Information Processing Letters 113*, 7 (2013), 241 – 244.

[149] Skutella, M., and Verschae, J. A robust PTAS for machine covering and packing. In *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I* (2010), M. de Berg and U. Meyer, Eds., vol. 6346 of *Lecture Notes in Computer Science*, Springer, pp. 36–47.

[150] Solomon, S. Fully dynamic maximal matching in constant update time. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA* (2016), pp. 325–334.

[151] Subramanian, S. A fully dynamic data structure for reachability in planar digraphs. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings* (1993), pp. 372–383.

[152] Tamassia, R., and Tollis, I. A unified approach to visibility representations of planar graphs. *Disc. Comput. Geom. 1*, 1 (1986), 321–341.

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

[154] Tarjan, R. Depth first search and linear graph algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160.

[155] Thomassen, C. The square of a graph is vertex pancyclic provided its block-decomposition is a path. Unpublished manuscript, 1975.

[156] Thomassen, C. Hamiltonian paths in squares of infinite locally finite blocks. 269–277.

[157] Thorup, M. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2000), STOC '00, ACM, pp. 343–350.

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

[159] Thorup, M. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005* (2005), pp. 112–119.

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

[161] Říha, S. A new proof of the theorem by Fleischner. *Journal of Combinatorial Theory, Series B 52*, 1 (1991), 117–123.

[162] Westbrook, J. Load balancing for response time. *Journal of Algorithms 35*, 1 (2000), 1 – 16.

[163] Wulff-Nilsen, C. 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* (2013), pp. 1757–1769.

[164] Wulff-Nilsen, C. Faster deterministic fully-dynamic graph connectivity. In *Encyclopedia of Algorithms*. 2016, pp. 738–741.