Dynamic Algorithms for Shortest Paths and Matching

Aaron Bernstein

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

©2016

Aaron Bernstein

All Rights Reserved

ABSTRACT

Dynamic Algorithms for Shortest Paths and Matching

Aaron Bernstein

There is a long history of research in theoretical computer science devoted to designing efficient algorithms for graph problems. In many modern applications the graph in question is changing over time, and we would like to avoid rerunning our algorithm on the entire graph every time a small change occurs. The evolving nature of graphs motivates the *dynamic* graph model, in which the goal is to minimize the amount of work needed to reoptimize the solution when the graph changes. There is a large body of literature on dynamic algorithms for basic problems that arise in graphs. This thesis presents several improved dynamic algorithms for two fundamental graph problems: shortest paths, and matching.

Table of Contents

1 Introduction **1**

1.1 The Model . . . 2

1.2 A Brief History of Dynamic Algorithms . . . 3

1.3 Overview of the Thesis . . . 6

I Partially Dynamic Shortest-Paths Algorithms **7**

2 Dynamic Shortests Paths Introduction **8**

2.1 Preliminaries . . . 9

2.2 The Even and Shiloach Tree . . . 11

3 Single Source Shortest Paths: Randomized **13**

3.1 Related Work . . . 13

3.2 Our Results . . . 14

3.3 Techniques . . . 15

3.4 Framework . . . 16

3.5 The Emulator . . . 20

3.5.1 The Techniques of Thorup and Zwick . . . 20

3.5.2 Defining the Emulator . . . 21

3.5.3 Proving Theorem 4 . . . 23

3.6 Conclusions . . . 25

4 Single Source Shortest Paths: Deterministic **27**
4.1 Our Results . . . 29
4.2 High Level Overview . . . 29
4.3 Preliminaries . . . 31
4.4 The Threshold Graph . . . 32
4.5 The Decremental SSSP Algorithm . . . 38
4.6 From Decremental to Incremental SSSP . . . 39
4.7 Conclusions . . . 40

5 All Pairs Shortest Paths in Directed Weighted Graphs **41**
5.1 Our Results . . . 42
5.2 Preliminaries . . . 45
5.3 Hop Distances and the Even and Shiloach Tree . . . 46
5.4 The Basic Approach . . . 47
5.5 A Simplified Not Quite O(mn) Algorithm . . . 51
5.5.1 The Algorithm . . . 52
5.5.2 Running Time Analysis . . . 54
5.5.3 Approximation Error Analysis . . . 56
5.6 The Final O(mn) Algorithm . . . 58
5.6.1 The Algorithm . . . 58
5.6.2 Running Time Analysis . . . 60
5.6.3 Approximation Error Analysis . . . 61
5.7 The h-SSSP Algorithm . . . 63
5.7.1 Limiting the dependence on Delta to O(Delta) . . . 66
5.8 Final Touches . . . 73
5.8.1 Removing the Assumption that We Know R in Advance . . . 73
5.8.2 The Incremental Setting . . . 74
5.8.3 A Fully Dynamic Algorithm . . . 75
5.9 Conclusions . . . 77

II Fully Dynamic Maximum Matching **79**

6 Dynamic Matching Introduction **80**

6.1 Preliminaries . . . 81

6.2 Previous Work . . . 82

7 Fully Dynamic Matching with Small Approximation Ratios **85**

7.1 Our Results . . . 86

7.2 Techniques . . . 88

7.3 Preliminaries . . . 90

7.4 The Framework . . . 91

7.4.1 General Graphs . . . 92

7.4.2 Small Arboricity Graphs . . . 93

7.5 A gamma-Restricted Fractional Matching Contains a Large Integral Matching . . . 94

7.6 An Edge Degree Constrained Subgraph Contains a Large Matching . . . 98

7.7 Maintaining an Edge Degree Constrained Subgraph . . . 102

7.8 Appendix of the More Technical Proofs . . . 106

7.8.1 Full proof of Theorem 20 . . . 106

7.8.2 Proof of Lemma 31 . . . 109

7.8.3 A Violation Oracle: Proof of Lemma 34 . . . 115

7.8.4 Maintaining an Edge Degree Constrained Subgraph in General Graphs . . . 117

7.8.5 Dynamic Orientation: Proving Theorem 17 . . . 123

7.9 Conclusions . . . 129

III Bibliography **131**

Bibliography **132**

Acknowledgments

I thank my advisor Cliff Stein for his time, his patience, and his generosity. His door was always open, and I am very grateful for our many hours of discussion over the past six years, and for his invaluable guidance.

I thank David Karger for first kindling my love of algorithms. His advanced algorithms class was an inspiration, and I am very grateful for the support and encouragement he provided me throughout my undergraduate years.

I would also like to thank my other collaborators Liam Roditty, Shiri Chechik, and Tsvi Kopelowitz for making the act of research such a pleasant one.

Thank you to my dear friend Raju Krishnamoorthy for his indispensable advice both moral and amoral.

Most importantly, thank you to my mother and sister. Without them I am nothing.

To my mother

Chapter 1

Introduction

Graphs are an extremely versatile mathematical object, capable of representing any situation in which we have objects and pair-wise relationships between those objects. Graphs model many different settings, such as social networks, the communication infrastructure, the interconnections of financial markets and the wiring of the human brain. In order to understand these and other problems modeled by graphs we need algorithms for processing these graphs. There is a long and rich history of designing algorithms for solving basic problems in graph theory such as graph search, shortest paths, flows, and matchings.

A critical but somewhat less studied aspect of graphs is their dynamic nature – graphs are often not given all at once, and they change over time. Given the sheer size of the graphs involved in settings such as the ones mentioned above, we cannot afford to rerun our algorithm on the entire graph every time a small change occurs. For this reason, starting in the early 80's [Even and Shiloach, 1981] researchers have devised efficient algorithms for solving basic problems in *dynamic* graphs, trying to minimize the amount of work needed to reoptimize the solution when the graph changes due to an edge/vertex being inserted/deleted, or to the modification to some numeric parameter. This thesis presents improved dynamic algorithm for two fundamental graph problems: shortest paths, and maximum matching.

1.1 The Model

The dynamic graph model is analogous to the standard data structure model. The algorithm is given an original graph, and must process an online sequence of updates and queries, where the update changes the graph in some way, while the query asks for information about the current version of the graph.

There are many variations on the dynamic model depending on exactly what types of updates are allowed. The standard model is the *fully dynamic* one, where an update can insert an edge, delete an edge, and in the case of a weighted graph it may also change the weight of an edge. The query then depends on the specific graph problem being considered. For example, in dynamic connectivity, the query CONNECTED(x, y) asks whether there is a path between vertices x and y in the current version of the graph; in dynamic single source shortest paths, the query DISTANCE(v) asks for the shortest distance from the fixed source s to vertex v. There are some problems for which instead of queries, it is more natural to require that the algorithm maintain some substructure in the graph, such as a maximum matching or a minimum spanning tree. A common restriction of the fully dynamic setting is the *partially dynamic* setting where updates consist of only insertions, or only deletions. The former case is known as *incremental*, the latter as *decremental*.

The efficiency of a dynamic algorithm is judged by two parameters: the *update* time is the time required to process an update, while the *query* time is the time required to process a query. There are also secondary concerns, such as whether the algorithm is randomized or deterministic, and whether the update time is amortized or worst-case. For any given dynamic problem there are often many possible trade-offs between update time and query time, as well as additional trade-offs depending on whether randomization and/or amortization is permitted. For most applications it is crucial that the user has easy access to information about the current graph, so typically the goal is to keep the query time small (constant or polylogarithmic), while minimizing the update time as much as possible.

The gap between randomized and deterministic algorithms tends to be especially pronounced in dynamic algorithms. The reason for this is that most (but not all) randomized algorithms must assume a significantly weaker adversary. In particular, they assume a *nonadaptive* adversary whose update sequence is fixed in advance, and does not depend on the algorithm's answers to queries. For example, randomized algorithms for dynamic shortest paths assume that the updates are independent

of the shortest paths returned by the query procedure of the algorithm; randomized algorithms for dynamic matching assume that the updates are independent of the matching maintained by the algorithm. The nonadaptivity assumption often leads to significantly faster algorithms in the context of shortest paths and matching. The price of this assumption is that it makes the algorithm unsuitable to certain applications; in particular, a nonadaptive algorithm cannot be used as a black-box data structure.

We now proceed to give a brief overview of the literature on dynamic graph algorithms as a whole. This is followed by a more detailed review of the existing work on dynamic shortest paths and dynamic matching, since these are the two problems addressed in this thesis.

1.2 A Brief History of Dynamic Algorithms

As far as we know, the first dynamic algorithm dates back to a result of Even Shiloach from 1981 which shows how to maintain a shortest path tree under deletions. Dynamic algorithms began to receive a lot of attention in the 90's, especially in the context of fully dynamic connectivity. In this problem, an update can add or delete an edge from an undirected graph, while the query CONNECTED(x, y) asks whether two vertices are connected in the current version of the graph. The main open question was whether it is possible to achieve polylog update and query times. The breakthrough 1995 paper of Henzinger and King [Henzinger and King, 1999] answered this question in the affirmative for randomized algorithms; Holm, Lichtenberg, and Thorup obtained such bounds deterministically in 1998 [Holm et al., 1998] (see also the journal version [Holm et al., 2001]). Since then, essentially every fundamental graph problem has been studied in the dynamic setting, with many dozens of papers on the topic. Much of this attention has been focused on basic graph problems such as connectivity and minimum spanning tree (e.g. [Frederickson, 1985; Eppstein et al., 1997; Henzinger and King, 1999; Henzinger and Thorup, 1997; Holm et al., 1998; Thorup, 2000; Kapron et al., 2013]), reachability and strongly connected components in directed graphs (e.g. [King, 1999; Demetrescu and Italiano, 2004; Demetrescu and Italiano, 2005; Roditty and Zwick, 2008a; Roditty, 2013; Lacki, 2011; Henzinger et al., 2014b; Henzinger et al., 2015a]), maximum matching (e.g. [Ivkovic and Lloyd, 1994; Sankowski, 2007; Onak and Rubinfeld, 2010a; Baswana et al., 2011a; Neiman and Solomon, 2013a; Gupta and Peng, 2013; Bernstein and Stein,

2015; Bernstein and Stein, 2016; Bhattacharya et al., 2015a] and many others), and shortest paths (e.g. [King, 1999; Demetrescu and Italiano, 2001; Baswana et al., 2003; Demetrescu and Italiano, 2004; Roditty and Zwick, 2012; Thorup, 2005; Baswana et al., 2007; Bernstein, 2009; Bernstein and Roditty, 2011; Bernstein, 2013; Henzinger et al., 2013; Henzinger et al., 2014b; Henzinger et al., 2014a] and many many others). There also exists rich literature on dynamic algorithms for a wide variety of other problems, such as dynamic minimum cut [Eppstein et al., 1997; Thorup, 2007], dynamic topological sort [Pearce and Kelly, 2006; Katriel and Bodlaender, 2006; Bender et al., 2009], and dynamic planarity testing [Italiano et al., 1993; Battista and Tamassia, 1996; Galil et al., 1999].

Shortest Paths: There are dozens of papers on dynamic shortest paths in particular. As before, the most general model is the fully dynamic one, where an update is allowed to insert or delete edges into the graph. In dynamic *all pairs* shortest paths (APSP), the query DISTANCE(x, y) can ask for the shortest distance between any pair of vertices x and y, while in dynamic *single source* shortest paths (SSSP), there is a fixed source s, and query DISTANCE(x) asks for the shortest distance from s to a vertex x. All existing algorithms can be extended to find the actual shortest *path* (not just distance), but outputting the path might by necessity take $O(n)$ time if the path is long, so since we typically want to keep the query time small, most researchers focus on answering dynamic distance queries.

For fully dynamic *single source* shortest paths, no non-trivial algorithm is known for maintaining exact distances. To this day, the fastest algorithm is to simply recompute the shortest path tree from scratch after each update, yielding an $O(m)$ update time and $O(1)$ query time. If the query only asks for an *approximate* shortest distance, then one can achieve a faster update time in dense graphs by using the dynamic spanner of Baswana *et al.*[Baswana et al., 2012]; for example, there is an algorithm with $\widetilde{O}(n^{1.5})$ update time and $O(1)$ query time that maintains 3-approximate shortest distances. [1]

For fully dynamic all pairs shortest paths the situation is more optimistic. The trivial algorithm achieves update time $\widetilde{O}(mn)$ and query time $O(1)$ by simply recomputing shortest paths from scratch after every update. There were a couple improvements in the 90's and early 2000's (e.g.

[1] $\widetilde{O}$-notation hides log factors, so $f(n) = \widetilde{O}(g(n))$ if $f(n) = O(g(n)\text{polylog}(n))$.

[King, 1999; Demetrescu and Italiano, 2001]) that were all eclipsed by the breakthrough paper of Demetrescu and Italiano in 2004 [Demetrescu and Italiano, 2004], which achieves update time $\widetilde{O}(n^2)$ and query time $O(1)$. The algorithm of Demetrescu and Italiano works for weighted directed graphs and is still the state of the art; nothing better is known even if the graph is unweighted and undirected. Since then, several results have achieved $o(n^2)$ update time, but at the cost of a polynomial (but sublinear) query time [Sankowski, 2005; Roditty and Zwick, 2012]. There are also algorithms that simultaneously achieve $o(n^2)$ update time and small (at most polylog) query time by settling for *approximate* shortest distances [Bernstein, 2009; Bernstein and Roditty, 2011; Henzinger et al., 2013; Abraham et al., 2014].

There has also been a lot of research devoted to achieving faster update times by focusing on the *partially dynamic* setting, where the update sequence consist of only edge insertions or only edge deletions. This setting is the focus of Part I of this thesis.

Matching: There is a large amount of research on the problem of maintaining a maximum matching in a dynamic graph. Most of this work has focused on unweighted graphs, where the goal is simply to maintain a matching with as many edges as possible. The trivial approach is to recompute a maximum matching from scratch after every update; using the classic Micali-Vazirani algorithm [Micali and Vazirani, 1980b; Vazirani, 1994] yields update time $O(m\sqrt{n})$. It is not hard to achieve update time $O(m)$ by observing that changing a single edge in the graph can only change the maximum matching by a single augmenting path. The only other dynamic algorithm for the exact case is from a 2007 paper of Sankowski [Sankowski, 2007], which achieves update time $O(n^{1.495})$ using fast matrix multiplication.

We would ideally like sublinear update times, so researchers turned to the question of maintaining an *approximate* maximum matching. In 2010, Onak and Rubinfeld presented a randomized algorithm that maintains a constant-approximate matching in $O(\log^2(n))$ update time. Since then, there have been a large number of papers improving upon this algorithm and achieving various update-approximation trade-offs. We present a more detailed discussion in Part II of the thesis.

1.3 Overview of the Thesis

This thesis places each result into its own chapter. In particular, the thesis is broken up into two parts: Part I contains three results for partially dynamic shortest paths, while Part II contains one (large) result for dynamic matching. But some readers might prefer to start with a high level overview of all the different results. To this end, I will now point out the overview sections that provide a summary of our results and the techniques we used to obtain them.

Chapter 2 in Part I serves as a general introduction to the partially dynamic shortest paths problem. The next three chapters then describe each of the three results in Part I in detail. For a high level overview, I would recommend the following: Sections 3.1, 3.2, and 3.3 in Chapter 3; Sections 4.1 and 4.2 in Chapter 4; and Sections 5.1, 5.2, 5.3, and 5.4 in Chapter 5.

Chapter 6 in Part II serves as a general introduction to the dynamic matching problem. I would then recommend Sections 7.1, 7.2, 7.3, 7.4 in Chapter 7 for a high-level overview of our result and the new technical framework we use to obtain it.

Part I

Partially Dynamic Shortest-Paths Algorithms

Chapter 2

Dynamic Shortests Paths Introduction

As far as we know, there are more papers on dynamic shortest paths than on any single other dynamic graph problem. This is in part because computing shortest paths efficiently is one of the most fundamental problems in graph algorithms, but another reason is that shortest paths seems to be an especially difficult problem in the dynamic setting. For more basic problems such as connectivity or minimum spanning tree, researchers were eventually able to develop fully dynamic algorithms with polylog update and query times. For shortest paths, such a goal seems out of reach, and even sublinear update times are difficult to achieve. For this reason, much of the research on the problem attempts to relax various requirements of the model in an effort to achieve more efficient algorithms.

In section 1.2 we provided a brief overview of fully dynamic algorithms for shortest paths. For fully dynamic single source shortest paths nothing non-trivial is known unless we allow a large approximation ratio (at least 3). For fully dynamic all pairs shortest paths we saw that progress largely seemed to halt at the $O(n^2)$ update time barrier of [Demetrescu and Italiano, 2004]: all existing algorithms to go beyond this update barrier either require a polynomial (but sublinear) query time, or require an approximation ratio of at least two.

For this reason, much of the research on dynamic shortest paths has turned to *partially dynamic* algorithms, which are the focus of this chapter. In Chapter 2 we define the setting more formally. Chapters 3 and 4 are devoted to partially dynamic algorithms for *single source* shortest paths; they contain a discussion of existing work on the problem, and present the first randomized and deterministic algorithms respectively to go beyond a natural barrier that stood for 3 decades. Chapter 5 contains a discussion of partially dynamic algorithms for *all pairs* shortest paths, and presents a

new result for weighted directed graphs.

2.1 Preliminaries

We start with some basic definitions that are shared by all the results discussed in Part I. Let $G = (V, E)$ be the main graph that is subject to a series of updates. As we process our updates, G always refers to the *current* version of the graph. Let $w(x, y)$ be the weight of edge (x, y) in G; if the problem specifies that G is unweighted, then $w(x, y) = 1$ for all edges. Let $\pi(x, y)$ be the shortest $x - y$ path in G; if there are multiple shortest paths from x to y we can use any tie breaking strategy which ensures that any subpath of a shortest path is itself a shortest path (For an example of such a tie-breaking strategy, see section 3.4 of [Demetrescu and Italiano, 2004]). Define $\delta(x, y)$ to be the length of $\pi(x, y)$, or ∞ if no $x - y$ path exists. Note that $w(x, y)$, $\pi(x, y)$, and $\delta(x, y)$ can all change as G itself changes due to edge updates.

Many of our algorithms rely on auxiliary graphs that are different from G. For any graph H, let $w_H(x, y)$ the weight of edge (x, y) in H, let $\pi_H(x, y)$ be the shortest $x - y$ path in H, and let $\delta_H(x, y)$ be the length of $\pi_H(x, y)$.

We now formally define the partially dynamic model. A partially dynamic algorithm allows either only deletions (*decremental*), or only incremental (*incremental*). We define these two cases separately, although all of the algorithms discussed in Part I can handle either setting. Note that in weighted graphs, the decremental/incremental settings also allow certain edge weight changes; the basic idea is that in a decremental algorithm shortest distances are monotonically increasing, while in an incremental one shortest distances are monotonically decreasing.

Definition 1 *Given a weighted graph G subject to an online sequence of updates, the update sequence is said to be* decremental *if every update is either an edge deletion or an edge weight increase; if G is unweighted, each update must be an edge deletion. The update sequence is said to* incremental *if every update is either an edge insertion or an edge weight decrease; if G is unweighted, each update must be an edge insertion. We say that an algorithm is decremental (resp. incremental) if it can process a decremental (resp. incremental) update sequence.*

Definition 2 *Given a graph G and a fixed source s, a decremental algorithm for* single source *shortest paths (SSSP) must process an online sequence of updates and queries, where the updates*

are decremental and the query DISTANCE*(v) asks for the shortest distance $\delta(s, v)$ for any vertex v. A decremental algorithm for* all pairs *shortest paths (APSP) must be capable of answering queries* DISTANCE*(u, v), which ask for the shortest distance $\delta(u, v)$ for any pair of vertices u and v. An incremental SSSP/APSP algorithm is defined in the same way, except that the updates must be incremental.*

All of the algorithms discussed in Part I only return approximate shortest distances. We now define this notion formally

Definition 3 *A dynamic shortest path algorithm is said to return α-approximate shortest distances if given any query that asks for the shortest distance $\delta(u, v)$, the algorithm returns a distance $\widehat{\delta}(u, v)$ such that $\delta(u, v) \leq \widehat{\delta}(u, v) \leq \alpha\delta(u, v)$.*

In a fully dynamic algorithm, the update time refers to the time (amortized or worst-case) required to process a single update. However, for partially dynamic algorithms it is usually easier to analyze the *total* update time over all updates. Note that if the graph is unweighted then the total number of updates is bounded: in a decremental algorithm edges are deleted one by one until none are left; in an incremental algorithm edges are inserted one by one until we reach the final graph. In particular, the update time will often be expressed in terms of m, which in a decremental setting is the number of edges in the original graph, and in an incremental setting is the number of edges in the final graph. If the graph is weighted there could be a potentially infinite number of updates, so the update time will also depend on another variable Δ, which refers to the total number of updates.

Definition 4 *Let A be a partially dynamic algorithm. We say that A has query time T_q if it answers every query in worst-case time at most T_q. We say that A has total update time T_u if it spends a total of T_u time processing all of the updates in the update sequence.*

Definition 5 *Given a graph G subject to a sequence of edge deletions, insertions, and weight changes, define* MAX-EDGES*(G) to the number of pairs (u, v) such that edge (u, v) is in the graph at* some *point during the update sequence. Note that if the sequence is decremental then* MAX-EDGES*(G) is simply the number of edges in the original graph, and if the sequence is incremental then* MAX-EDGES*(G) is the number of edges in the final graph.*

2.2 The Even and Shiloach Tree

Almost all existing partially dynamic algorithms for maintaining a shortest path tree under deletions use as a building block an algorithm for Even and Shiloach [Even and Shiloach, 1981] for maintaining a partially dynamic shortest path tree up to a given distance d. The algorithm of Even and Shiloach only works for undirected graphs, but was later extended by King [King, 1999] to work for directed graphs. We will require several similar versions of this result, so we start by stating the algorithm in its full generality, which actually applies to the fully dynamic setting – that is, the update sequence can contain both edge insertions and edge deletions. We then state some corollaries for the partially dynamic setting that easily follow.

Theorem 1 *[King, 1999] Let G be a dynamic directed weighted graph and let s be a fixed source. Say that G is subject to an online sequence of updates and queries, where the updates form a fully dynamic sequence (edge insertions, deletions, and weight changes), while the query* DISTANCE*(v) asks for the shortest distance $\delta(s,v)$ for any vertex v. There exists an algorithm with the following properties:*

- *The query time of the algorithm is $O(1)$.*
- *The time to process some update σ is $O(1 + E(\sigma))$, where $E(\sigma)$ is the number of edges incident to some vertex v for which $\delta(s,v)$ changed as a result of the update.*

Definition 6 *Given a dynamic graph G and a distance bound d, we say that a decremental/incremental SSSP/APSP algorithm runs* up to distance d *if for any query that asks for $\delta(u,v)$, the algorithm outputs a value $\widehat{\delta}(u,v)$ that satisfies the following properties:*

- *If $\delta(u,v) \leq d$ then $\widehat{\delta}(u,v) = \delta(u,v)$.*
- *If $\delta(u,v) > d$ then $\widehat{\delta}(u,v) \geq \delta(u,v)$.*

If the algorithm is α-approximate, then the first property only requires that $\delta(u,v) \leq \widehat{\delta}(u,v) \leq \alpha\delta(u,v)$.

Lemma 1 *[King, 1999] Let $G = (V,E)$ be a directed graph with positive integer weights subject to a* decremental *or* incremental *sequence of updates, let s be a fixed source, let d be a distance bound,*

and assume that all edge weights are upper bounded by d. There is a dynamic SSSP algorithm that maintains distances from s up to distance d that has query time $O(1)$ and total update time $O(md)$ over the entire sequence of updates to G, where $m = \text{MAX-EDGES}(G)$.

Proof: Because weights are positive and integral, all edge weights are between 1 and d, so since each weight either only increases (decremental) or only decreases (incremental), the algorithm has to process at most $O(md)$ edge weight changes.

Now, observe that the algorithm only has to process vertices v for which $1 \leq \delta(s, v)$: in the decremental setting, once we have $\delta(s, v) > d$ we will always have $\delta(s, v) > d$ so we can delete v from the graph and return $\widehat{\delta}(s, v) = \infty$ from that point on; in the incremental setting, it is easy to modify King's algorithm [King, 1999] to ignore v until $\delta(s, v) \leq d$, after which point we will always have $\delta(s, v) \leq d$. Thus, given any vertex v, the distance $\delta(s, v)$ can change at most d times over the entire sequence of updates: in a decremental setting every time $\delta(s, v)$ changes it increases by at least 1, which can happen at most d times before $\delta(s, v) > d$; in the incremental setting, we first process v when $\delta(s, v) \leq d$, after which point the distance can decrease at most d times before $\delta(s, v) = 1$, which is the minimum possible distance.

Thus, we explore all the edges incident to a vertex v at most d times, leading to a total update time of $O(md)$. □

Corollary 1 *Given an* unweighted *graph and a fixed source, there exists a decremental as well as incremental SSSP algorithm with query time $O(1)$ and total update $O(mn)$ over the entire sequence of edge deletions, where n is the number of vertices in the graph (which never changes), and $m = \text{MAX-EDGES}(G)$.*

Proof: Since the graph is unweighted, we always have $\delta(s, v) \leq n$ or $\delta(s, v) = \infty$, so it suffices to maintain distances up to $d = n$. By Lemma 1 this requires $O(mn)$ total update time, as desired. □

Definition 7 *Let G be a graph subject to a decremental/incremental sequence of edge updates. let s be a fixed source, and let d be some depth bound. We define* $\mathbf{ES}(G, s, d)$ *to be the decremental/incremental algorithm for maintaining distances from s up to distance d with the bounds given in Lemma 1 ($O(1)$ query time, $O(md)$ total update time). In words, we denote this algorithm as running an Even and Shiloach tree in G from source s up to depth d.*

Chapter 3

Single Source Shortest Paths: Randomized

Publication History: This chapter is based on results originally published by Liam Roditty and I in SODA 2011 [Bernstein and Roditty, 2011]. As discussed below in Section 3.3, however, both the algorithm and the analysis have been significantly altered.

3.1 Related Work

As discussed in Section 1.2, Fully Dynamic SSSP has proved to be quite difficult in the dynamic setting, even in unweighted, undirected graphs. The trivial algorithm recomputes a shortest path tree from scratch after every update, which takes time $O(m)$ using a simple breadth first search. This is still the best algorithm known, although using the dynamic spanner of Baswana [Baswana et al., 2012] it is possible to achieve a better update time in dense graphs at the cost of a large approximation ratio: for example, update time $\widetilde{O}(n^{1.5})$ for a 3-approximation.

For partially dynamic algorithms (decremental or incremental), the trivial algorithm again achieves update time $O(m)$. Equivalently, the trivial algorithm has *total* update time $O(m^2)$, since a partially dynamic algorithm on an unweighted graph performs at most m updates, where $m = \text{MAX-EDGES}(G)$ (see Definition 5).

The first improvement to this bound goes back all the way to 1981 [Even and Shiloach, 1981]: Even and Shiloach showed how to achieve total update time $O(mn)$ in undirected unweighted

graphs (see Corollary 1 above). A similar result was independently found by Dinitz [Dinitz, 2006], and was later generalized to directed graphs by King [King, 1999]. However, even though dynamic shortest paths as a whole quickly became a very active field, no progress was made over this simple $O(mn)$ algorithm.

Roditty and Zwick [Roditty and Zwick, 2004b] presented an explanation for this lack of progress, by showing a reduction from boolean matrix multiplication to both the incremental and the decremental SSSP problem in unweighted undirected graphs. This reduction implies that unless a major breakthrough in combinatorial boolean matrix multiplication is provided, no combinatorial algorithm for the problem can go beyond the $O(mn)$ total update time of Even and Shiloach (except perhaps by log factors). Very recently, Henzinger *et al* [Henzinger et al., 2015b] proved the same $O(mn)$ lower bound for *any* type of algorithm (not just a combinatorial one), assuming their online boolean matrix-vector multiplication conjecture.

These lower bounds motivated the study of the approximate version of this problem. Liam Roditty and I presented the first algorithm to go beyond this $O(mn)$ total update time bound [Bernstein and Roditty, 2011]; the approximation is the best that could be hoped for, as the algorithm returns $(1 + \epsilon)$-approximate distances. This result is the focus of the current chapter.

3.2 Our Results

We achieve two results. The main result is for partially dynamic SSSP, as described above. Our technique is rather general, however, and ends up yielding new results for partially dynamic APSP as well.

Theorem 2 *Let G be an unweighted, undirected graph, and let s be a fixed source. There is an algorithm that solves $(1 + \epsilon)$-approximate decremental or incremental SSSP with the following bounds: the query time is $O(1)$ and the total update time is (with high probability) $O(n^{2+O(1/\sqrt{\log(n)})}) = O(n^{2+o(1)})$.*

Theorem 3 *Let G be an unweighted, undirected graphs, and let $k \geq 2$ be a fixed integer. There exists an algorithm that solves $(2k - 1 + \epsilon)$-approximate decremental or incremental APSP with the following bounds: the query time is $O(k)$ (which is constant), and the total update time is (with high probability) $O(n^{2+1/k+O(1/\sqrt{\log(n)})}) = O(n^{2+1/k+o(1)})$.*

New Work Published After our Results: Our result was the first to go beyond the $\widetilde{O}(mn)$ total update time barrier for decremental SSSP, at the cost of a $(1+\epsilon)$-approximation. Since the publication of our result in SODA 2011, there has been a series of further improvements, most of which were done by Henzinger, Krinninger, and Nanongkai. Recall that our result achieves total update time $O(n^{2+o(1)})$ in undirected unweighted graphs. For the same case of undirected and unweighted graphs, Henzinger *et al.*presented an algorithm with total update time $O(m^{1+o(1)} + n^{1.8+o(1)}$ [Henzinger et al., 2014c]. They later developed an algorithm with a close to optimal total update time of $O(m^{1+o(1)})$, which has the additional advantage of achieving the same update time for graphs with weights polynomial in n [Henzinger et al., 2014a]. The same authors also showed how to go beyond total update time $\widetilde{O}(mn)$ for *directed* graphs [Henzinger et al., 2014b; Henzinger et al., 2015a], although the state-of-the-art still only has total update time $\widetilde{O}(mn^{.9})$. Finally, in a very recent paper, Shiri Chechik and I showed how to achieve a total update time of $\widetilde{O}(n^2)$ with a *deterministic* algorithm, in contrast to all the other results mentioned here, which are randomized and non-adaptive. This deterministic algorithm is the subject of the next chapter in this thesis.

3.3 Techniques

Our approach relies on extending a common tool used in static algorithms to the dynamic setting. Many approximate shortest paths algorithms in undirected graphs start by constructing a sparse *emulator* (or spanner) of the graph – that is, another graph on the same vertex set with a similar shortest distance structure (see Definition 8 for a formal description). The algorithm then computes shortest paths in the *sparse* emulator rather than in the original graph (see e.g. [Awerbuch et al., 1998; Cohen, 1998; Aingworth et al., 1999; Dor et al., 2000; Cohen and Zwick, 2001; Elkin and Peleg, 2004; Thorup and Zwick, 2006; Pettie, 2009]). However, this approach has never been used in a decremental (or incremental) setting. The reason for this is that as edges in G are being deleted, we also have to change the edges in the emulator H, so that H remains a good emulator. But a deletion in G can lead to *insertions* into H. Thus, we cannot run decremental algorithms on our emulator, because from the perspective of H we are not in a decremental setting.

Our main contribution is an emulator that possesses several novel properties relating to how it

changes as edges in G are being deleted. The emulator itself is basically identical to one used by Bernstein [Bernstein, 2009], which is in turn a modification of a spanner developed by Thorup and Zwick [Thorup and Zwick, 2006]. However, the properties we prove are entirely *new to this result*.

In the original SODA 2011 paper in which Liam Roditty and I published these results [Bernstein and Roditty, 2011], we proved that as edges in G are being deleted, insertions into H can indeed occur, but they are rather well behaved. We then showed how to extend several existing decremental algorithms to handle such well behaved insertions.

However, since then, we realized that our algorithm can in fact be greatly simplified. We would like to thank Shiri Chechik and Sebastian Krinninger for bringing this to our attention. Recall that given any dynamic problem, there is always the following trivial algorithm: after every update, run a static (non-dynamic) algorithm for the problem on the current version of the graph. If the setting is decremental or incremental and the graph is unweighted, then the total number of updates is at most $m = \text{MAX-EDGES}(G)$. Now, for single source shortest paths in unweighted graphs, a single instance can be solved in $O(m)$ time using a single breadth first search: this yields a trivial dynamic algorithm with total update time $O(m^2)$. But what if we ran this algorithm on a sparse emulator of the graph, with only approximately n edges? Then in theory we could get total update time $O(n^2)$. There are two main difficulties that arise. The first is how to efficiently maintain the emulator as the graph changes. The second is that as the adversary performs m updates on the graph G, this could potentially lead to significantly more than n edge changes in the emulator.

Our main contribution is an emulator that is easy to maintain in a decremental (or incremental) setting, and that only changes a small number of times as G changes.

3.4 Framework

Our entire framework is grounded on the emulator presented in Theorem 4 below. We leave the proof of Theorem 4 for Section 3.5, and in this section instead focus on how we use this theorem to prove our two main results (Theorems 2 and 3).

Definition 8 *An emulator H of G is a graph with the same vertex set as G, but with different (possibly weighted) edges. We say that H is an (α, β)-emulator if for any pair of vertices x, y we have $\delta(x, y) \leq \delta_H(x, y) \leq \alpha\delta(x, y) + \beta$. We say that H is an α-emulator if it is an $(\alpha, 0)$-emulator.*

Remark: Many approximate graph algorithms use spanners, which are emulators whose edge set must be a subset of the original edges E. However, we stick to the more general definition of emulators.

Definition 9 *We say that an algorithm maintains an (α, β) emulator in a dynamic graph G if it maintains a graph H which is always guaranteed to a be a (α, β) emulator for the current version of G. (Note that the algorithm can change H as G changes.)*

Theorem 4 *Let G be an unweighted undirected graph subject to a decremental or incremental sequence of updates, and let ϵ be a fixed (constant) approximation parameter. Let n be the number of vertices in G, and let $m = \text{MAX-EDGES}(G)$. There exists an algorithm that maintains an emulator H with the following properties (note that the algorithm inserts, deletes, and changes the weights of edges in H as G changes):*

1. *The current version of H is always a $((1+\epsilon), n^{O(1\sqrt{\log(n)})}) = ((1+\epsilon), n^{o(1)})$ emulator for the current version of G (with high probability).*
2. *At all times H has at most $O(n^{1+O(1\sqrt{\log(n)})}) = O(n^{1+o(1)})$ edges (with high probability).*
3. *The algorithm has a total update time of $O(mn^{O(1\sqrt{\log(n)})}) = O(mn^{o(1)})$ over the entire sequence of edge updates to G (with high probability).*
4. *The total number of changes made to H (insertions, deletions, or weight changes) over the entire sequence of edge updates to G is $O(n^{1+O(1\sqrt{\log(n)})}) = O(n^{1+o(1)})$ (with high probability).*

It may seem problematic at first that we are maintaining an $((1+\epsilon), n^{o(1)})$ emulator, since we want our final distances to be $(1+\epsilon)$ approximate without an additive error. Note, however, that the additive error can be subsumed under the $(1+\epsilon)$ approximation for not-too-small distances, i.e. distances greater than $n^{o(1)}$. Distances smaller than $n^{o(1)}$ can then be handled separately because many dynamic algorithms run much faster for small distances.

Theorem 5 *Say that we want to solve decremental (resp. incremental) single source shortest paths or decremental (resp. incremental) all pairs shortest paths in an unweighted graph G subject to*

a sequence of edge deletions (resp. insertions). Let $\epsilon < 1$ *be some fixed constant approximation parameter. Let us say that there exists a static (i.e. non-dynamic) algorithm* A_s *that can solve a single instance of the problem in time* T_s *in a graph that has* n *vertices and* $O(n^{1+O(1\sqrt{\log(n)})}) = O(n^{1+o(1)})$ *edges, and in which all distances are at most* $O(n)$*. Let* α_s *be the approximation ratio of this static algorithm. Let us say that there exists a decremental (resp. incremental) algorithm* A_d *that can solve the problem up to distance* $d = O(n^{O(1\sqrt{\log(n)})}) = O(n^{o(1)})$ *(see Definition 6) on the sequence of edge deletions (resp. insertions), and that* A_d *has* <u>*total*</u> *update time* T_d *and approximation ratio* α_d*. Then, there exists a* randomized non-adaptive *algorithm* A *that solves the decremental problem with expected total update time* $O(T_d + n^{1+O(1\sqrt{\log(n)})} \cdot T_s) = O(T_d + n^{1+o(1)} \cdot T_s)$*, and with high probability has approximation ratio* $\max\{\alpha_d, \alpha_s(1+\epsilon)\}$.

Proof: The proof follows more or less directly from Theorem 4, but gets a bit technical simply because there are so many time bounds to combine. Our algorithm will maintain the emulator H in Theorem 4. Recall that by Property 1 of Theorem 4, H is a $((1+\epsilon), n^{O(1\sqrt{\log(n)})})$ emulator; we will use $\epsilon' = \epsilon/2$, resulting in a $(1+\epsilon/2, \text{AE})$ emulator, where $\text{AE} = n^{O(1\sqrt{\log(n)})}$ is the additive error of the emulator. Let $d = 2\text{AE}/\epsilon$, and note that $d = O(n^{O(1\sqrt{\log(n)})}) = O(n^{o(1)})$. For distances up to d we run the algorithm A_d on the original graph G; recall that A_d is dynamic, so it maintains α_d approximate distances (up to distance d) over the entire sequence of changes to G in total update time T_d. We also run the static algorithm A_s on the emulator H (this time up to arbitrary distances). Each time an edge in H changes, we have to rerun algorithm A_s on the new version of H, but by property 4 of Theorem 4, an edge change to H only occurs $O(n^{1+O(1\sqrt{\log(n)})})$ times over the entire sequence of changes to G, leading to a total update time of $O(n^{O(1\sqrt{\log(n)})} \cdot T_s)$, as desired.

We must now analyze the approximation guarantee. Given any query that asks for $\delta(v,w)$ (where v must be the source in the case of dynamic SSSP and can be any vertex in the case of dynamic APSP), our final algorithm A outputs the minimum returned by algorithms A_s and A_d. More formally, let $\delta_s(v,w)$ be the distance returned by A_s and $\delta_d(v,w)$ be the distance returned by A_d; our final algorithm A then returns $\delta^*(v,w) = \min\{\delta_s(v,w), \delta_d(v,w)\}$. Now, we know that A_d is α_d-approximate up to distance d, so $\delta(v,w) \le \delta_d(v,w)$, and $\delta_d(v,w) \le \alpha_d \delta(v,w)$ if $\delta(v,w) \le d$. We also know that the static algorithm A_s returns α_s approximate distances in H, and that H is a $(1+\epsilon/2, \text{AE}) = (1+\epsilon/2, \epsilon d/2)$ emulator so letting $\delta_H(v,w)$ be the shortest $v-w$

distance in H, we have that

$$\delta(v,w) \leq \delta_H(v,w) \leq \delta_s(v,w) \leq \alpha_s \delta_H(v,w) \leq \alpha_s((1+\epsilon/2)\delta(v,w) + \epsilon d/2) \tag{3.1}$$

But note that when $\delta(v,w) \geq d$, the above Equation 3.1 implies

$$\delta(v,w) \leq \delta_s(v,w) \leq (1+\epsilon)\alpha_s \delta(v,w) \tag{3.2}$$

On the other hand, when $\delta(v,w) \leq d$, we know that

$$\delta(v,w) \leq \delta_d(v,w) \leq \alpha_d \delta_d(v,w) \tag{3.3}$$

Recall that our final output is $\delta^*(v,w) = \min\{\delta_s(v,w), \delta_d(v,w)\}$; combining Equations 3.2 and 3.3 implies that

$$\delta(v,w) \leq \delta^*(v,w) \leq \max\{\alpha_d, \alpha_s(1+\epsilon)\}\, \delta(v,w)$$

as desired. □

Proof of Theorem 2 We directly apply Theorem 5 above. For algorithm A_s we use Dijkstra's algorithm; it is well known that Dijkstra's can easily be made to run in time $O(m)$ as long all distances are at most $O(n)$. This yields $T_s = O(n^{O(1\sqrt{\log(n)})}) = O(n^{o(1)})$ and $\alpha_s = 1$. For algorithm A_d we use the algorithm of Lemma 1. Since the algorithm A_d only runs up to distance $d = O(n^{O(1\sqrt{\log(n)})})$, this yields a total update time of $O(mn^{O(1\sqrt{\log(n)})}) = O(n^{2+O(1\sqrt{\log(n)})})$, and approximation ratio $\alpha_d = 1$ (the algorithm of Lemma 1 is exact). Theorem 5 then yields an algorithm with the bounds specified in Theorem 2. □

Proof of Theorem 3 We directly apply Theorem 5 above. Here though, we must be careful about the precise definition of "computing" all-pairs shortest paths. The standard method is to construct an n by n table where entry (i,j) contains some suitable approximation to $\delta(v_i, v_j)$. However, the precise nature of the data structure does not matter, and a table is not necessarily the right choice. All we need from A_d and A_s is that they compute a *distance oracle*, i.e. a data structure that given any pair of vertices (v,w) can return an approximation to $\delta(v,w)$ in $O(1)$ time.

For A_s, we use a well known distance oracle of Thorup and Zwick [Thorup and Zwick, 2005]. For any fixed integer $k > 2$, the distance oracle requires $O(mn^{1/k})$ time to construct (in expectation), and given any pair of vertices (v,w) can return a $2k-1$ approximation to $\delta(v,w)$ in $O(1)$ time.

For A_d, we use an existing algorithm of Roditty and Zwick [Roditty and Zwick, 2012] which shows how to maintain the above oracle of Thorup and Zwick in a decremental or incremental setting. For any fixed integer k and any distance bound d, the oracle returns a $2k-1$ approximation to any $\delta(v,w)$ in $O(1)$ time, as long as $\delta(v,w) \leq d$ (otherwise it returns an arbitrary number $\geq \delta(v,w)$); the oracle requires a total update time $O(mdn^{1/k})$ over the entire sequence of updates, where $m = \text{MAX-EDGES}(G)$ (recall Definition 5).

Combining these algorithms using Theorem 5 yields an algorithm with the bounds specified in Theorem 3. □

3.5 The Emulator

3.5.1 The Techniques of Thorup and Zwick

Our emulator relies on a clustering technique introduced by Thorup and Zwick [Thorup and Zwick, 2005], which we now review.

Definition 1 *Let $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{k-1} \supseteq A_k = \emptyset$ be sets of vertices ($2 \leq k \leq \log(n)$ is a parameter of our choosing). In particular, we start with $A_0 = V$, and for $1 \leq i \leq k-1$ every vertex in A_i is independently sampled and put into A_{i+1} with probability $1/n^{1/k}$. We refer to the indices $1, 2, ..., k$ as vertex priorities, where the priority of v is i if and only if $v \in A_i \setminus A_{i+1}$.*

Definition 2 *Define the i-witness of v, or $p_i(v)$, to be the vertex in A_i that is nearest to v: $p_i(v) =$ argmin$_{w \in A_i}(\delta(v,w))$. To break ties, we pick the $p_i(v)$ that survives to the set A_j of largest index (equivalently, it is contained in the most sets A_j). Define $\delta(v, A_i)$ to be $\delta(v, p_i(v))$.*

Definition 3 *Given a vertex $v \in A_i - A_{i+1}$, we define the cluster of v to be $C(v) = \{w \in V \mid \delta(w,v) < \delta(w, A_{i+1})\}$. We define the bunch of v to be $B(v) = \{w \in V \mid v \in C(w)\}$.*

Lemma 2 *[Thorup and Zwick, 2005] With high probability, the size of every bunch is $O(kn^{1/k}\log(n))$. Thus, the total size of all the bunches is $O(kn^{1+1/k}\log(n))$. Note that v is in the bunch of w if and only if w is in the cluster of v, so the total size of all the clusters must also be $O(kn^{1+1/k}\log(n))$.*

Thorup showed how to efficiently compute all the clusters $C(v)$, as well as all the witnesses $p_i(v)$ ($i \leq k-1$). Roditty and Zwick [Roditty and Zwick, 2012] later showed that we can efficiently maintain this oracle in a decremental or incremental setting up to a distance threshold d.

Theorem 6 *[Roditty and Zwick, 2012] Let G be an undirected graph with positive edge weights, subject to a decremental or incremental sequence of updates. Given any distance d, we can maintain the following information as G changes:*

1. *For every vertex v, we maintain a set that contains all vertices w such that $w \in C(v)$ AND $\delta(v,w) \leq d$.*

2. *For every vertex v and every priority level $1 \leq i \leq k$, we maintain every $p_i(v)$ for which $\delta(v, p_i(v)) \leq d$; for every i for which $\delta(v, p_i(v)) > d$, our algorithm need not maintain anything.*

With high probability, the algorithm can maintain all of the above information in total update time $O(mdn^{1/k})$ over the entire sequence of updates.

3.5.2 Defining the Emulator

The emulator we use is basically identical to an emulator used by Bernstein [Bernstein, 2009], which is in turn a modification of one developed by Thorup and Zwick [Thorup and Zwick, 2006]. Both are based upon the techniques covered in Section 3.5.1. Note that although the emulator itself is not new, we prove several new properties relating to how it changes as the original graph G changes.

We start with an emulator of Thorup and Zwick [Thorup and Zwick, 2006] (they actually used a spanner, but for simplicity, we express it as an emulator).

Theorem 7 *[Thorup and Zwick, 2006] Let H be the following undirected emulator: for every vertex v, and every $w \in C(v)$, H contains an edge of weight $\delta(v,w)$ from v to w. Also, for every vertex v and every vertex priority i, H contains an edge of weight $\delta(v, p_i(v))$ from v to $p_i(v)$. Then, H contains $O(kn^{1+1/k})$ edges and is a $((1+\epsilon/2), \zeta)$ emulator, where $\zeta = O((6/\epsilon)^k)$. (Technical note: Lemma 2.3 of Thorup and Zwick works for* any ϵ*, so plugging $\epsilon/2$ into their lemma we get our Theorem. In particular, note that $2 + 2/(\epsilon/2)$ – the term in their lemma – is $\leq 6/\epsilon$ because $\epsilon < 1$).*

Corollary 2 *If $\delta(x,y) \geq (2/\epsilon)\zeta$ (recall: $\zeta = O((6/\epsilon)^k)$) then $\delta_H(x,y) \leq (1+\epsilon)\delta(x,y)$.*

Proof: $\delta_H(x,y) \leq (1+\epsilon/2)\delta(x,y) + \zeta \leq (1+\epsilon/2)\delta(x,y) + (\epsilon/2)\delta(x,y) = (1+\epsilon)\delta(x,y)$. □

Bernstein [Bernstein, 2009] showed that the properties of H still hold if we remove all heavy edges. We take advantage of this, although for different reasons.

Definition 4 *Let $\gamma = (24/\epsilon)\zeta$, where $\zeta = O((6/\epsilon)^k)$ is the additive error of H, and k is the maximum priority level used in our clustering.*

Theorem 8 *[Bernstein, 2009] Let H be the same emulator as in Theorem 7, except with all edges of weight $\geq \gamma$ removed. Then, H has $O(kn^{1+1/k})$ edges and is a $((1+\epsilon), O((6/\epsilon)^k))$ emulator. Also, if $\delta(x,y) \geq \gamma/2$ then $\delta_H(x,y) \leq (1+\epsilon)\delta(x,y)$ (no additive error).*

Proof: The proof is almost identical to one in [Bernstein, 2009], and is given here for the sake of completeness. To bound the approximation error of H, we let H' be the original emulator without heavy edges removed. We now break our proof into two possible cases. If $\delta(x,y) \leq \gamma/3$, then we know that $\delta_{H'}(x,y) \leq (1+\epsilon)\gamma/3 + \zeta \leq \gamma$, so the path from x to y in H' never uses edges of length greater than γ, so this path must also exist in H.

If $\delta(x,y) > \gamma/3$ then we split $\pi(x,y)$ into paths of length $\lceil \gamma/12 \rceil$. That is, we let $y_1 = x$, we let y_2 be the vertex on $\pi(x,y)$ that is at distance $\lceil \gamma/12 \rceil$ from x, we let y_3 be the vertex at distance $\lceil \gamma/12 \rceil$ from y_2 and so on up to some y_r. We define $y_{r+1} = y$. We then let π_i be the subpath of $\pi(x,y)$ from y_i to y_{i+1}.

(Technical note: we choose y_r in such a way that $\lceil \gamma/12 \rceil \leq w(\pi_r) \leq \lceil \gamma/6 \rceil + 1$. This can always be done because if we break $\pi(x,y)$ into paths of length $\lceil \gamma/12 \rceil$, then the remainder left over will have length $\leq \lceil \gamma/12 \rceil$, so we can just concatenate this remainder to the last path of length $\lceil \gamma/12 \rceil$, thus yielding a path of length $\leq 2\lceil \gamma/12 \rceil \leq \lceil \gamma/6 \rceil + 1$.)

Note that for any i we have $w(\pi_i) \geq \gamma/12 \geq (2/\epsilon)\zeta$, so by the corollary of Theorem 7 there exists a $(1+\epsilon)$ approximate path p_i in H' (see Figure 1). But all of the p_i have length less than $(1+\epsilon)(\lceil \gamma/6 \rceil + 1) < \gamma/3$, so they must also be in H (not just H'). Thus, we consider the path $p = p_1 \circ p_2 \circ ... \circ p_r$; this is an $x-y$ path in H of length $\leq (1+\epsilon)\delta(x,y)$, which completes the proof. □

Recall that the whole purpose of constructing a sparse emulator H was to run all of our algorithms on H instead of G. But as we delete edges from G we also need to modify H so that it remains a $((1+\epsilon), O((6/\epsilon)^k))$ emulator. In particular, the clusters and witnesses in G change as we delete edges, so we need to maintain all of the $C(v)$ and $p_i(v)$. The following lemma stems directly from Theorem 6.

Lemma 3 *We can decrementally maintain the truncated emulator in Theorem 8 in a total of* $O(m\gamma n^{1/k})$ *time over all deletions in* G*.*

3.5.3 Proving Theorem 4

Although we use an already existing emulator, we use it in an entirely novel way; all of the analysis in this section was new to the paper under discussion [Bernstein and Roditty, 2011]. In particular, we will show how the emulator H is affected by changes to the original graph G.

Our end goal is to prove Theorem 4. The first three properties of this theorem will follow directly from our discussion so far. The fourth property, however, requires some additional analysis. We need to upper bound the total number of edge changes to H as the original graph G undergoes either a decremental or incremental sequence of updates. Although the basic idea behind the proof is the same in the incremental and decremental case, the details are a bit different, so we prove a separate lemma for each case.

Lemma 4 *Let* H *be the emulator in Theorem 8, and say that the original graph* G *is subject to a decremental (only deletions) sequence of updates. Then the number of edges* inserted *into* H *over all deletions in* G *is* $O(k^2\gamma n^{1+1/k}\log(n))$ *(with high probability).*

Proof: By the definition of clusters, the only way a deletion in G can cause an edge (v, w) to be inserted into H, is if the deletion causes $\delta(w, A_i)$ to increase, which in turn causes w to join $C(v)$ (here, v must have priority $i-1$). Also, since all edges in H have weight less than γ, $\delta(w, A_i)$ must have been less than γ before the deletion in G.

But note that since all edges in G have weight 1, $\delta(w, A_i)$ can increase at most γ times before it exceeds γ. Moreover, every time $\delta(w, A_i)$ increases, at most $O(kn^{1/k}\log(n))$ edges (v, w) are inserted into H; this is because (v, w) can only exist in H if $w \in C(v)$, and at any time, w is contained in $O(kn^{1/k}\log(n))$ clusters (with high probability) – see Lemma 2. Thus, for any vertex w and any vertex priority i, the total number of edges inserted into H on account of $\delta(w, A_i)$ increasing is $O(\gamma kn^{1/k}\log(n))$. But there are only $O(kn)$ pairs (w, i), which leads to an $O(k^2\gamma n^{1+1/k}\log(n))$ upper bound on the total number of edges inserted into H.

□

Lemma 5 *Let H be the emulator in Theorem 8, and say that the original graph G is subject to an incremental (only insertions) sequence of updates. Then the number of edges* deleted *from H over all deletions in G is $O(k^2 \gamma n^{1+1/k} \log(n))$ (with high probability).*

Proof: By the definition of clusters, the only way an insertion into G can cause an edge (v, w) to be deleted from H, is if the insertion causes $\delta(w, A_i)$ to decrease for some i, which in turn causes w to leave $C(v)$ (here, v must have a priority $(i-1)$). Also, since all edges in H have weight less than γ, $\delta(w, A_i)$ must have been less than γ before the insertion in G.

But note that since all edges in G have weight 1, once $\delta(w, A_i)$ is less than γ, it can decrease at most γ more times. Note also that for any given vertex w, the number of edges (v, w) that are in H because $w \in C(v)$ is at most $O(kn^{1/k} \log(n))$, because with high probability, that is how many clusters w is contained in (see Lemma 2). Thus, even if all of these edges are deleted due to $\delta(w, A_i)$ decreasing, we still have that the number of edges (v, w) deleted from H as a result of $\delta(w, A_i)$ decreasing is $O(kn^{1/k} \log(n))$. Since $\delta(w, A_i)$ can decrease at most γ times we have that the total number of edges deleted from H on account of $\delta(w, A_i)$ decreasing is $O(\gamma kn^{1/k} \log(n))$. But there are only $O(kn)$ pairs (w, i), which leads to an $O(k^2 \gamma n^{1+1/k} \log(n))$ upper bound on the total number of edges deleted in H.

□

We now have all the tools required to prove our main theorem.

Proof of Theorem 4 Note that given a fixed graph G, the only parameter we choose in constructing the emulator is the number of priorities k: once k is set, the sets $A_1, A_2, ..., A_k$ are chosen entirely at random, and the rest of the emulator follows from there. Given some constant approximation parameter $\epsilon < 1$, the emulator we construct is the emulator H in Theorem 8, with $k = \sqrt{\log(n))}$. Turning to Definition 4, this yields $\gamma = n^{O(1/\sqrt{\log(n)}} = n^{o(1)}$. We also have $n^{1+1/k} = n^{O(1/\sqrt{\log(n)})} = n^{o(1)}$.

Let us now focus on properties 1 and 2. Note that if we had a *fixed* (non-dynamic) graph G, and constructed the emulator H in Theorem 8, then H would satisfy properties 1 and 2 by Theorem 8. However, we need to show that as G changes (and H with it), properties 1 and 2 will hold for *all* versions of G. The key observation is that for any given version of G, H is defined entirely in terms of the sets $A_1, A_2, ..., A_k$ (see Definition 1), which are chosen entirely at random. Since we assume the dynamic adversary is oblivious and non-adaptive, we can treat the update sequence

as fixed in advance; that is, the update sequence is completely independent both of our random choices (oblivious), and by our answers to queries (non-adaptive). This implies that the sets the sets $A_1, A_2, ..., A_k$ will be random from the perspective of *every* version of the graph. This in term means that Theorem 8 always holds with high probability for the current version of G, and so by the union bound it holds with high probability for *all* versions of G, completing the proof of properties 1 and 2. (More formally, Theorem 8 holds with high probability, which means that it can be made to hold with probability $1 - n^{1/c}$ for any constant c, while only multiplying the running time by $O(c)$. On the other hand, since each update deletes an edge, there are at most $O(m) = O(n^2)$ version over which we need to union bound.)

Property 3 follows directly from Lemma 3.

Property 4 follows from Lemmas 4 and 5. We need to handle two the cases separately: the first where G is subject to a decremental sequence of updates, and second an incremental sequence. In the *decremental* case, we know by Lemma 4 that the total number of insertions into H over the entire sequence of deletions to G is $O(k^2 \gamma n^{1+1/k} \log(n)) = n^{O(1/\sqrt{\log(n)})}$. But we know by Theorem 8 that H originally only has $n^{O(1/\sqrt{\log(n)})}$ edges, so since that is also the total number of insertions, it is clear that the total number of insertions AND deletions to H is at most $n^{O(1/\sqrt{\log(n)})}$. We must finally consider edge weight changes. Note that the weight of an edge (v, w) in H always corresponds to $\delta(v, w)$ in G; thus, while edges can enter and leave H, edge weights can only increase. Since we delete all edges of weight greater than γ, we have that the total number of edge weight changes is $\gamma \cdot n^{O(1/\sqrt{\log(n)})} = n^{O(1/\sqrt{\log(n)})} = n^{1+o(1)}$ as desired. The argument for the case where G is subject to an *incremental* is exactly symmetrical. □

3.6 Conclusions

We presented the first partially dynamic algorithm for single source shortest paths to go beyond the $O(mn)$ total update time bound. Our approximation error is only $(1 + \epsilon)$: the conditional lower bounds of [Roditty and Zwick, 2004b] and [Henzinger et al., 2015c] suggest that an exact algorithm would not be possible. The main open question of course is whether one can achieve total update time less than $O(n^2)$. In particular, can one do better for sparse graphs?

Since our result, there have been several improvements by other authors [Henzinger et al.,

2014c; Henzinger et al., 2014a; Henzinger et al., 2014b; Henzinger et al., 2015a]. For undirected graphs, this progress culminated in a result of Henzinger *et al.*[Henzinger et al., 2014a], which maintains $(1+\epsilon)$ approximate shortest paths with total update time $O(m^{1+o(1)})$. Note that this is optimal up to a $m^{o(1)}$ factor because $O(m)$ time is clearly necessary. There have also been several algorithms to go beyond the $O(mn)$ bound for directed graphs [Henzinger et al., 2014b; Henzinger et al., 2015a], though the state of the art in [Henzinger et al., 2015a] still only achieves total update time $O(mn^{.9})$.

There remain a few big open questions in this area. The first is whether we can do better for directed graphs. The second is whether for undirected graphs we can get rid of the $O(m^{o(1)})$ factor, and achieve something truly optimal up to log factors. Finally, the last open question is whether we can break the $O(mn)$ bound deterministically, as *all* the algorithms discussed here are randomized and non-adaptive. Going beyond the $O(mn)$ bound deterministically is the topic of the next chapter.

Chapter 4

Single Source Shortest Paths: Deterministic

Publication History: This chapter presents a result done in collaboration with Shiri Chechik that has been accepted to STOC 2016.

As discussed in Section 3.1, the state of the art partially dynamic algorithm for maintaining *exact* single source shortest distances has total update time $O(mn)$, and there are conditional lower bounds suggesting this is the best possible [Roditty and Zwick, 2004b; Henzinger et al., 2015c]. In Chapter 3 we presented the first algorithm to go beyond the $O(mn)$ bound by allowing a $(1+\epsilon)$ approximation [Bernstein and Roditty, 2011]. The algorithm is randomized and non-adaptive, in that it assumes the update sequence is fixed in advance, and so does not depend on the approximate shortest distances (or paths) returned by the algorithm. Since then, there have been several other algorithms to go beyond this $O(mn)$ total update time barrier [Henzinger et al., 2014c; Henzinger et al., 2014a; Henzinger et al., 2014b; Henzinger et al., 2015a] (see Section 3.6), but all of these algorithms are also randomized and non-adaptive. As discussed in Section 1.1, the assumption of a non-adaptive adversary makes these algorithms inadequate for certain settings, and in particular prevents them from being used as black-box data structures. Thus an important open question remains: is it possible to beat the $O(mn)$ bound with a deterministic algorithm?

This gap between randomized and deterministic algorithms is extremely common in dynamic

shortest paths problems. For partially dynamic shortest paths in particular, the large majority of the state of the art algorithms are randomized and non-adaptive. To see why such algorithms tend to be more efficient in the dynamic setting, consider the following simple clustering problem. The goal is to maintain a set of $\backsim \sqrt{n}$ vertices called centers in an unweighted undirected graph, such that all other vertices are at distance at most $\sqrt{n}$ from one of the centers. (Assume for simplicity that the graph is always connected). This basic clustering tool is used in a huge number of approximate shortest path algorithms, both static and dynamic (e.g. [Peleg and Schäffer, 1989; Peleg and Ullman, 1989; Thorup and Zwick, 2001; Thorup and Zwick, 2005; Baswana and Kavitha, 2006; Bernstein and Roditty, 2011; Roditty and Zwick, 2012; Chechik, 2014; Henzinger et al., 2014b; Henzinger et al., 2014a], and many many others). In the static setting, there is an obvious randomized algorithm: sample $O(\sqrt{n}\log(n))$ centers uniformly at random. A simple greedy deterministic algorithm also exists in the static setting, but in the dynamic setting, it is easy to see that an efficient deterministic algorithm cannot exist: whatever $\sqrt{n}$ centers we choose, the adversary can disconnect them while leaving the rest of graph intact, forcing us to restart from scratch. With randomization and an oblivious adversary, however, the problem becomes easy: once again we sample $O(\sqrt{n}\log(n))$ centers uniformly at random. The adversary then proceeds to change the graph, but since the updates are oblivious to our random choices we can argue that these centers are uniformly random in *all* versions of the graph, and so with high probability will form a valid clustering throughout the entire update sequence. The extra assumption of a non-adaptive adversary is often necessary to prevent the adversary from gaining information about our randomly chosen centers from the algorithm's answers to shortest path/distance queries.

Essentially every randomized algorithm for dynamic shortest paths (all pairs or single source) uses some generalization of the above clustering, and so is difficult to match with a deterministic algorithm. As far as we know the only exception is the decremental $(1+\epsilon)$-approximate all pairs shortest path algorithm of Henzinger *et al*. (for unweighted undirected graphs) [Henzinger et al., 2013], which succeeds in derandomizing the simple clustering-based algorithm of Roditty and Zwick [Roditty and Zwick, 2012] that achieves the same bounds. For decremental SSSP in particular, all the approximation algorithms that break through the $O(mn)$ barrier rely on clustering using randomization, and all require an oblivious non-adaptive adversary.

Shiri Chechik and I have developed a different approach that does not rely on any clustering

scheme, and is the first to break through the $O(mn)$ barrier deterministically. The paper has been accepted to STOC 2016.

4.1 Our Results

Theorem 9 *Given an undirected unweighted graph G and a fixed source s, there is a* deterministic *decremental SSSP algorithm with total update time $O(m\log^3(n) + n^2\log(n)\epsilon^{-1})$. The query time is $O(1)$.*

We can easily extend our algorithm to work for graphs with small positive integer weights, at the cost of multiplying the update time by $O(W)$, where W is the largest weight in the graph. We can also extend the algorithm to work in the *incremental* settings (see Section 4.6).

Our algorithm does not match the randomized state of the art of total update time $O(m^{1+o(1)})$, but it is optimal up to log factors for dense graphs, and is the first deterministic algorithm to go beyond the $O(mn)$ barrier. In addition to being deterministic, our algorithm has several secondary advantages. The first is that it is much simpler than the three randomized algorithms for the problem [Bernstein and Roditty, 2011; Henzinger et al., 2014b; Henzinger et al., 2014a]. The reason for this is that those algorithms all relied on variations of the same clustering technique of Thorup and Zwick [Thorup and Zwick, 2005] which is somewhat involved, especially in the dynamic setting. We develop an entirely different approach which is much simpler, and could potentially be of use in other deterministic algorithms for dynamic shortest paths. The simplicity also allows us to avoid the extra $m^{O(1/\sqrt{\log(n)})} = m^{o(1)}$ term incurred by all the randomized algorithms, which again arose from the Thorup and Zwick clustering. Thus, in addition to being deterministic, our algorithm is simpler and faster than the $O(n^{2+o(1)})$ randomized algorithm of Bernstein and Roditty, and is in fact faster than *all* existing randomized algorithms for the problem in dense graphs where $m = \Omega(n^{2-1/\sqrt{\log(n)}})$.

4.2 High Level Overview

To highlight the simplicity of our approach, and to make the technical details easier to follow, we start with a high level description of how to solve $(1+\epsilon)$-approximate decremental SSSP in total

update time $O(n^{2.5})$, ignoring log factors that arise from some of the technical details. We assume for this section that ϵ is a fixed constant. Note that the Even and Shiloach tree of Lemma 1 already provides a method for maintaining *short* distances. In particular, running $ES(G, s, 5\sqrt{n}\epsilon^{-1})$ only requires $O(m\sqrt{n}) = O(n^{2.5})$ time and maintains all distances $\delta(s, v)$ for which $\delta(s, v) \leq 5\sqrt{n}\epsilon^{-1}$ (see Definition 7).

To maintain *long* distances, we will sparsify the graph G. Let us say that a vertex $v \in V$ is *heavy* if it has degree at least $\sqrt{n}$, and *light* otherwise. Our main observation is that any shortest path π in G can contain at most $3\sqrt{n}$ heavy vertices: intuitively, this is because no two heavy vertices on π can share a common neighbor because then there would be a very short path between them, which we could use to obtain a path shorter than the shortest path π. (The formal proof is only slightly complicated by the fact that two heavy vertices on the path *can* share a common neighbor if the two heavy vertices are already right next to each other on the shortest path π; however if they are at distance 3 away from each other in π, they cannot have a common neighbor.) Thus, since the neighborhood of a single heavy vertex contains $\sqrt{n}$ vertices, and there are only n vertices in total, there can only be $\sqrt{n}$ heavy vertices on the shortest path π.

Since we are only concerned with vertices v for which $\delta(s, v) >> \sqrt{n}$, we know that the shortest path from s to v will contain far more light vertices than heavy vertices. Thus, if we are only seeking an approximate distance, we can effectively ignore the heavy vertices and thus reduce the number of edges in the graph. More specifically, our sparsification works as follows. Let HEAVY be the set of heavy vertices in G, and let $G[\text{HEAVY}]$ be the subgraph of G induced by the heavy vertices. Consider the (still unweighted) graph G' which contains all the edges of G, as well as an edge between all pairs of vertices v, w such that v and w are both heavy and are in the same connected component in $G[\text{HEAVY}]$. (Of course these connected components will change as edges in G are deleted, but they are easy to maintain because dynamic connectivity is easy to do efficiently in undirected graphs.) As described, G' in fact contains more edges than G, but it is easy to exactly mimic G' with a sparse graph; include only the edges of G incident to at least one light vertex, and then for every component C in $G[\text{HEAVY}]$ create a new vertex c in G' and add an edge of weight $1/2$ from c to every vertex in the component C. The resulting graph has at most $O(n^{1.5})$ edges: $O(\sqrt{n})$ per light vertex, and a single extra edge of weight $1/2$ per heavy vertex.

Of course distances in G' differ from those in G. In fact they are *shorter* because G' allows us to

skip over whole components of heavy vertices. But intuitively, as long as $\delta(s, v)$ is large, $\delta_{G'}(s, v)$ will not be too much smaller than $\delta(s, v)$, because $\delta(s, v)$ is dominated by light vertices anyway, so skipping over the heavy ones does not change much. In particular, we will prove in that

$$\delta_{G'}(s, v) \leq \delta(s, v) < \delta_{G'}(s, v) + 5\sqrt{n} \ .$$

Thus, as long as $\delta(s, v) >> \sqrt{n}$, $\delta_{G'}(s, v) + 5\sqrt{n}$ will be a good approximation to $\delta(s, v)$.

All in all, our algorithm runs two Even and Shiloach trees; $ES(G, s, 5\sqrt{n}\epsilon^{-1})$ handles short distances, while $ES(G', s, n)$ handles long ones. The first part runs in time $O(m\sqrt{n}) = O(n^{2.5})$ because we have bounded depth, while the second runs in time $O(n^{2.5})$ because G' is sparse. To answer queries for a vertex v, the algorithm simply takes the minimum of the subroutine for short distances, and the subroutine for long ones. Using these ideas it is not hard to reduce the total update time to $\widetilde{O}(n^2)$ by using $O(\log(n))$ different heaviness thresholds to handle $O(\log(n))$ ranges of $\delta(s, v)$, and taking the minimum of the $O(\log(n))$ subroutines.

4.3 Preliminaries

Our algorithm will make use of several graphs that are different from G. Given any subset of vertices $V' \subseteq V$, we define the induced graph $G[V']$ to contain all the vertices V', and all edges $(u, v) \in E$ such that u and v are both in V'. Given any two sets S, T we define the set difference $S \setminus T$ to contain all elements s such that $s \in S$ but $s \notin T$.

Throughout our algorithms, we will often compute distances up to a certain threshold d (see Definition 6). To this end, we define the following auxiliary definition

Definition 10 *Given any number d, the function* $\text{BOUND}_d(x)$ *is equal to x if $x \leq d$, and to ∞ otherwise.*

Our algorithm uses the Even and Shiloach tree from Lemma 1 as its basic building block. Note that running the Even and Shiloach tree from source s up to distance d (Definition 6) is precisely equivalent to maintaining $\text{BOUND}_d(\delta(s, v))$ for all vertices v. To fit this building block cleanly into our algorithm, we introduce a few simple variations on Lemma 1.

Lemma 6 *[Even and Shiloach, 1981] Let $G = (V, E)$ be a dynamic graph with positive integer weights, let s be a fixed source, and say that for every vertex v we are guaranteed that the distance*

$\delta(s, v)$ never decreases due to an edge insertion into G. Then, the total *update time of* $\mathbf{ES}(G, s, d)$ *over the entire sequence of edge updates is $O(m \cdot d + \Delta)$, where $m =$ MAX-EDGES(G), and Δ is the total number of edge changes.*

Remark 1 *The typical guarantee given for the ES-tree is the one in Lemma 1: $\mathbf{ES}(G, s, d)$ has total update time $O(md)$ as long as the graph is subject to* only *deletions or* only *insertions. But recall that in the proof of Lemma 1, the only-deletions assumption was only necessary to guarantee that distances do not decrease, so the same bound holds as long as this monotonicity is separately guaranteed for the insertions as well, as it is in the statement of this theorem. The existence of insertions leads to the extra $O(\Delta)$ term because the algorithm needs to spend $O(1)$ time per edge change. (In the case of only deletions, we have $\Delta \leq m$, so we can ignore the Δ term).*

Corollary 3 *If a dynamic graph G and a source s satisfy all the assumptions of Lemma 6 except that weights in G are not necessarily integral, but all the weights are positive and integer multiples of some number x, then the* total *running time of* $\mathbf{ES}(G, s, d)$ *is $O(md/x + \Delta)$, where $m =$ MAX-EDGES(G). (We simply divide all weights by x and apply Lemma 6.)*

4.4 The Threshold Graph

Definition 11 *Given a graph G and a positive integer threshold τ, we say that a vertex v in G is τ-heavy if v has degree at least τ, and we say that v is τ-light otherwise. Let* HEAVY(τ) *be the set of all τ-heavy vertices in G; note that when we say that a vertex v is τ-heavy or τ-light, this is* always *with respect to the main graph G, never with respect to any other graph the algorithm relies on.*

Definition 12 *Given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, and an integer $\tau \in [1, n]$, define the threshold graph $G_\tau = (V_\tau, E_\tau)$ as follows (note that although G is unweighted, the edges in G_τ have weights 1 and $1/2$):*

- *V_τ contains every vertex $v \in V$.*
- *V_τ also contains an additional vertex c for each connected component C in the induced subgraph $G[$HEAVY$(\tau)]$.*
- *E_τ contains all edges incident to τ-light vertices $v \in V$. All such edges are given weight 1.*

- *For every τ-heavy vertex $v \in V$, E_τ contains an edge from v to c of weight $1/2$, where c is the component vertex in $V_\tau \setminus V$ that corresponds to the component C in $G[\textsc{heavy}(\tau)]$ that contains v.*

For any pair of vertices $s, t \in V$ define $\pi_\tau(s,t)$ to be the shortest path from s to t in G_τ, and define $\delta_\tau(s,t)$ to be the weight of this path.

Lemma 7 *The number of edges in the threshold graph G_τ is always $O(n\tau)$.*

Proof: This follows directly from the definition of G_τ. E_τ contains the $O(n\tau)$ edges incident to τ-light vertices in V, as well as a single additional edge for every τ-heavy vertex in V. □

Lemma 8 *For any graph $G = (V, E)$, any positive integer threshold τ, and any pair of vertices $s, t \in V$:*

$$\delta_\tau(s,t) \leq \delta(s,t) < \delta_\tau(s,t) + \frac{5n}{\tau}.$$

Proof: We first show the simpler claim that $\delta_\tau(s,t) \leq \delta(s,t)$. Consider the shortest $s - t$ path $\pi(s,t) \in G$. We will exhibit a (not necessarily simple) path $P_\tau(s,t) \in G_\tau$ with weight exactly $\delta(s,t)$. $P_\tau(s,t)$ contains all the edges of $\pi(s,t)$ that are incident to some τ-light vertex: these edges have weight 1 in both G and G_τ. The only edges that remain are edges $(v, w) \in \pi(s,t)$ where v and w are both τ-heavy. Since v and w are neighbors in G, they are part of the same connected component C in $G[\textsc{heavy}(\tau)]$, and so G_τ contains a path of length 1 from v to w; namely, the path $(v, c) \circ (c, w)$. We thus replace every edge $(v, w) \in \pi(s,t)$ with a path of length 1 in G_τ, which results in a path $P_\tau(s,t) \in G_\tau$ with weight exactly $\delta(s,t)$, as needed.

We now prove that $\delta(s,t) \leq \delta_\tau(s,t) + \frac{5n}{\tau}$. Let $\pi_\tau(s,t)$ be the shortest $s - t$ path in G_τ. Let L_{π_τ} be the set of τ-light vertices $v \in V \bigcap \pi_\tau(s,t)$. Now, let $V^* \subseteq V$ be the set of vertices containing

- All the vertices in L_{π_τ}
- All the τ-heavy vertices in G
- All the neighbors of τ-heavy vertices in G.

Let G^* be the subgraph of G induced by V^*. We first show that there must exist an $s - t$ path in G^*. We construct this path by looking at $\pi_\tau(s,t)$. $\pi_\tau(s,t)$ contains edges incident to τ-light vertices in

G^*, as well as subpaths of length 2 of the form $(v,c) \circ (c,w)$, where v and w are τ-heavy vertices that are in the same connected component C in $G[\textsc{heavy}(\tau)]$. The edges on $\pi_\tau(s,t)$ incident to light vertices exist in G^* as well, so we can follow those directly. For every subpath $(v,c) \circ (c,w)$, since v and w are both in the same connected component in $G[\textsc{heavy}(\tau)]$, there is a $v-w$ path in G using only heavy vertices, so that path is in G^* as well. We have thus shown that there exists an $s-t$ path in G^*.

Now, let $\pi^*(s,t)$ be the *shortest* $s-t$ path in G^*. Let $\delta^*(s,t)$ be the length of $\pi^*(s,t)$. Since G^* is a subgraph of G, we know that $\delta(s,t) \leq \delta^*(s,t)$. We now show that

$$\delta^*(s,t) < \delta_\tau(s,t) + \frac{5n}{\tau} \tag{4.1}$$

which completes the proof of Lemma 8. Let X^* contain all vertices in $\pi^*(s,t)$ that are NOT in L_{π_τ}: observe that by definition of V^*, every vertex $v \in X^*$ is either τ-heavy, or adjacent in G^* to a τ-heavy vertex. Note that $\delta^*(s,t) \leq |L_{\pi_\tau}| + |X^*|$, while $\delta_\tau(s,t) \geq |L_{\pi_\tau}|$ because all the vertices in L_{π_τ} are on $\pi_\tau(s,t)$. We thus have that

$$\delta^*(s,t) \leq \delta_\tau(s,t) + |X^*|,$$

so to prove Inequality 4.1, it suffices to show that

$$|X^*| < \frac{5n}{\tau}\ . \tag{4.2}$$

Let Y^* be the set containing every 5th vertex in X^*: that is, Y^* contains the vertex in X^* that is closest to s in G^*, the one that is 6th closest to s, the one that is 11th closest to s, and so on. Clearly, $|Y^*| \geq |X^*|/5$. We complete the proof of Equation 4.2, and hence of Lemma 8 as a whole, by arguing that

$$|Y^*| < \frac{n}{\tau}\ . \tag{4.3}$$

To prove Equation 4.3 above, for any vertex $v \in V_\tau$ we define $\textsc{ball}(G^*, v, 2) \subseteq V^*$ to be the set containing v itself, the neighbors of v in G^*, and all vertices at distance 2 from v in G^*. Now, on the one hand, for every $v \in Y^*$ we have that $|\textsc{ball}(G^*, v, 2)| > \tau$ because since $v \in Y^* \subset X^*$ we know that v is either itself τ-heavy or adjacent in G^* to a τ-heavy vertex, and so $\textsc{ball}(G^*, v, 2)$ must contain that τ-heavy vertex as well as its $\geq \tau$ neighbors. On the other hand, if v and w are both in Y^* then $\textsc{ball}(G^*, v, 2)$ and $\textsc{ball}(G^*, w, 2)$ must be disjoint because otherwise there would be

a path of length at most 4 between v and w in G^*, which contradicts the fact that the subpath of the shortest path $\pi^*(s,t)$ between v and w is of length at least 5. Thus, the total number of vertices among all of the $\text{BALL}(G^*, v, 2)$ for $v \in Y^*$ is strictly greater than $\tau|Y^*|$, but since there are only n vertices in the graph, we have $|Y^*| < n/\tau$, as desired. □

Now that we have shown that distances in G_τ are a close approximation to those in G, we show that these distances, as well as the graph G_τ itself, can be easily maintained.

Lemma 9 *Given a graph G subject to a sequence of edge deletions, and a positive integer threshold τ, we can maintain the graph G_τ in* total *time $O(m \log^2(n))$.*
Moreover, $\text{MAX-EDGES}(G_\tau) = O(n\tau + n\log(n))$.

Proof: G_τ contains two types of edges: those incident to τ-light vertices, and those from a τ-heavy vertex to its component vertex c. The first types of edges are trivial to maintain, since they are simply a subset of the edges in G; when a deletion causes a τ-heavy vertex to become τ-light we must add all of its incident $O(\tau)$ edges to G_τ, but this transition can only happen a single time per vertex over the entire sequence of deletions because vertex degrees are only decreasing. This first type of edges thus requires $O(m)$ time to maintain, and only leads to $O(\min\{n\tau, m\})$ edge insertions into G_τ.

We now show how to maintain the edge from each τ-heavy vertex v to its component vertex c, where c corresponds to the connected component C in $G[\text{HEAVY}(\tau)]$ that contains v. First off, note that $G[\text{HEAVY}(\tau)]$ itself is easy to maintain because it is simply a subgraph of G. We can maintain connected components in $G[\text{HEAVY}(\tau)]$ by using a dynamic *c*onnectivity data structure (CDS) on the graph $G[\text{HEAVY}(\tau)]$. We use the CDS of Holm *et al.*[Holm et al., 2001], which is based on top trees. This CDS can process insertions and deletions into the graph with amortized update time of $O(\log^2(n))$. It is not hard to check that the top trees used by their algorithm can be augmented to support more than just basic connectivity queries. In particular, their CDS can answer the following queries:

- **connected(u,v):** determines whether u and v are in the same connected component in the current graph. The query time is $O(\log(n))$.
- **size(v)**: returns the size of the connected component of v. The query time is $O(\log(n))$.

- **component(v)**: Returns a list of all the vertices in the same connected component as v. The query time is $O(\log(n)$ + number of vertices returned).

To maintain the graph G_τ as G changes, we will run the above CDS on the graph $G[\text{HEAVY}(\tau)]$. Note that $G[\text{HEAVY}(\tau)]$, like G, is only subject to edge deletions. When the adversary deletes an edge (u, v) in G where both u and v are τ-heavy, this edge must be deleted from $G[\text{HEAVY}(\tau)]$ as well, and this deletion is processed by the CDS in time $O(\log^2(n))$. Similarly, when a vertex $v \in V$ transitions from τ-heavy to τ-light, all of its incident edges must be deleted from $G[\text{HEAVY}(\tau)]$, and processed by the CDS. Each edge is deleted from $G[\text{HEAVY}(\tau)]$ at most once, so the total update time of the CDS is $O(m \log^2(n))$.

We must now show how to use the connectivity information maintained by the CDS to maintain the graph G_τ; in particular, how to maintain the edges from a τ-heavy vertex to its component vertex $c \in V_\tau \setminus V$. Whenever an edge $(u, v) \in G[\text{HEAVY}(\tau)]$ is deleted, we first query the CDS in $O(\log(n))$ time to check whether u and v are still part of the same connected component in $G[\text{HEAVY}(\tau)]$; if yes, the edges of G_τ do not change, and we are done. Otherwise, the deletion of (u, v) has caused the component to split into two. We now query CDS.size(u) and CDS.size(v) to determine in $O(\log(n))$ time which of the two parts is smaller. Say, wlog, that CDS.size(v) $\leq$ CDS.size(u). Let C be the original component that contained both u and v before the deletion of (u, v). Let C_v be the component containing v after the deletion. We can use CDS.component(v) to find all the vertices in C_v in time $O(\log(n) + |C_v|)$. Before the deletion, G_τ contained an edge from every vertex in C to the component vertex $c \in V_\tau \setminus V$. After the deletion, we add a new component vertex c_v to G_τ, and for every $w \in C_v$ we remove the edge (w, c) and add the edge (w, c_v). This takes time $O(|C_v|)$ and makes $O(|C_v|)$ edge changes to $G[\text{HEAVY}(\tau)]$. Amortized over all edge deletions in $G[\text{HEAVY}(\tau)]$ we have $\sum |C_v| \leq n \log(n)$ because edges in $G[\text{HEAVY}(\tau)]$ are only being deleted, so components are only splitting apart, and each time a vertex w is part of the smaller component C_v in a component split, we know that its component has shrunk by a factor of at least two.

Thus, the total time to process a deletion in $G[\text{HEAVY}(\tau)]$ is dominated by the $O(\log^2(n))$ update time of the CDS, yielding the desired $O(m \log^2(n))$ total update time. Moreover, from the bounds above, we see that at most $O(n\tau + n \log(n))$ edges are inserted into the graph; $O(n\tau)$ of the first type (edges incident to a τ-light vertex), and $O(n \log(n))$ of the second (edges from a τ-heavy

vertex to its component vertex c). By Lemma 7, the number of edges in the initial G_τ is $O(n\tau)$, so $\text{MAX-EDGES}(G_\tau) = O(n\tau + n\log(n))$.

$\square$

Lemma 10 *Given a graph G subject to a sequence of edge deletions, a positive integer threshold τ, and a pair of vertices u, v in G, the distance $\delta_\tau(u, v)$ never decreases as edges in G are deleted.*

Proof: Say that the adversary deleted edge (x, y) in G. Note that any path in G_τ consists of a concatenation of subpaths of length 1 between vertices in V; each subpath is either an edge of weight 1 incident to a τ-light vertex, or two edges of weight $1/2$ through a component vertex $c \in V_\tau \setminus V$. Thus, to show that distances in G_τ do not decrease, we show that for any pair of vertices $a, b \in V$ such that $\delta_\tau(a, b) = 1$ *after* the deletion of (x, y), we also had $\delta_\tau(a, b) = 1$ *before* the deletion of (x, y). We know that $\delta_\tau(a, b) = 1$ after the deletion if and only if edge (a, b) is in E, AND/OR a and b are in the same connected component in $G[\text{HEAVY}(\tau)]$. But either of these cases would clearly hold before the deletion of an edge as well, so we had $\delta_\tau(a, b) = 1$ before the deletion. $\square$

Lemma 11 *Given a graph $G = (V, E)$ subject to a sequence of deletions, a fixed source s, a positive integer threshold τ, and a depth bound d, we can maintain the distance $\text{BOUND}_d(\delta_\tau(s, v))$ for all vertices v in a* total *update time of $O(m\log^2(n) + n \cdot d \cdot (\tau + \log(n))$.*

Proof: We simply maintain the graph G_τ as edges in G are deleted, and run **ES**(G_τ, s, d) (See Definition 7). By Lemma 9, we can maintain the graph G_τ in time $O(m\log^2(n))$. Moreover, by Lemma 9 we have $\text{MAX-EDGES}(G_\tau) = O(n\tau + n\log(n))$. This bound on $\text{MAX-EDGES}(G_\tau)$ implies that the total number of changes made to G_τ is $\Delta = O(n\tau + n\log(n))$. By Lemma 10 distances in G_τ never decrease, and all weights in G_τ are either $1/2$ or 1, so by Lemma 6 and Corollary 3, the total running time of the ES-tree is $O(\text{MAX-EDGES}(G_\tau) \cdot d + \Delta)$, which is $O(n \cdot d \cdot (\tau + \log(n)))$. $\square$

4.5 The Decremental SSSP Algorithm

Our goal is to maintain approximate distances from a fixed source s. For every integer $i \in [1, \lfloor \log(n) \rfloor]$, let $\tau_i = \frac{n}{2^i}$, let $d_i = 2^i \cdot \frac{10}{\epsilon}$, and for every vertex v let

$$\widehat{\delta}_i(v) = \text{BOUND}_{d_i}(\delta_{\tau_i}(s,v)).$$

Let $\widehat{\delta}(v) = \min_i\{\widehat{\delta}_i(v) + 5 \cdot 2^i\}$. When the adversary queries the distance to a vertex v, our algorithm returns $\widehat{\delta}(v)$.

The execution of the algorithm is simple. By Lemma 11, for any i we maintain $\widehat{\delta}_i(v)$ for all vertices v in total update time $O(m \log^2(n) + n \cdot d_i \cdot (\tau_i + \log(n)) = O(m \log^2(n) + n^2 \epsilon^{-1} + n \log(n) d_i)$. Doing this for every i yields total update time $O(m \log^3(n) + n^2 \log(n) \epsilon^{-1})$ because $\sum_i d_i = O(n\epsilon^{-1})$. To maintain all the $\widehat{\delta}(v)$, for each vertex v we create a min-heap HEAP_v containing $\widehat{\delta}_i(v)$ for every i. The algorithm can access any $\widehat{\delta}(v)$ in $O(1)$ time by looking at the minimum of the heap, thus leading to an $O(1)$ query time. Maintaining the heaps is easy: each $\widehat{\delta}_i(v)$ can change at most $d_i = O(2^i \epsilon^{-1})$ times, so there will be at most $\sum_i d_i = n\epsilon^{-1}$ changes to each HEAP_v, and since each heap contain $O(\log(n))$ elements, a change requires $O(\log\log(n))$ time to process. Maintaining HEAP_v for all v thus requires total update time only $O(n^2 \log\log(n) \epsilon^{-1})$.

We now turn to proving that for any vertex v, $\delta(s,v) \leq \widehat{\delta}(s,v) \leq (1+\epsilon)\delta(s,v)$. The fact that $\delta(s,v) \leq \widehat{\delta}(s,v)$ follows directly from Lemma 8, because for every i

$$\begin{aligned} \delta(s,v) &\leq \delta_{\tau_i}(s,v) + \frac{5n}{\tau_i} \\ &\leq \widehat{\delta}_i(s,v) + \frac{5n}{\tau_i} \\ &= \widehat{\delta}_i(s,v) + 5 \cdot 2^i \ . \end{aligned}$$

To prove that $\widehat{\delta}(s,v) \leq (1+\epsilon)\delta(s,v)$, we need to show that for *some* i, we have $\widehat{\delta}_i(s,v) + 5 \cdot 2^i \leq (1+\epsilon)\delta(s,v)$. Let k be the largest index for which $\delta(s,v) \geq 2^k \cdot \frac{5}{\epsilon}$, so

$$2^k \cdot \frac{10}{\epsilon} \geq \delta(s,v) \geq 2^k \cdot \frac{5}{\epsilon} \ . \tag{4.4}$$

Now, by Lemma 8, $\delta_{\tau_k}(s,v) \leq \delta(s,v) \leq 2^k \cdot \frac{10}{\epsilon} = d_k$, so we have

$$\widehat{\delta}_k(s,v) = \text{BOUND}_{d_k}(\delta_{\tau_k}(s,v)) = \delta_{\tau_k}(s,v) \leq \delta(s,v) \ .$$

Thus,

$$\widehat{\delta}_k(s,v) + 5 \cdot 2^k \leq \delta(s,v) + 5 \cdot 2^k \leq (1+\epsilon)\delta(s,v) \ .$$

(The last inequality follows from $\delta(s,v) \geq 2^k \cdot \frac{5}{\epsilon}$ in Equation 4.4.)

4.6 From Decremental to Incremental SSSP

In this section we sketch the modifications needed to make our algorithm incremental rather than decremental.

We first remind the reader that the Even-Shiloach algorithm works with the same bounds in the incremental setting. In particular, we have the following analogy to Lemma 6:

Lemma 12 *[Even and Shiloach, 1981] Let $G = (V, E)$ be a dynamic graph with positive integer weights, let s be a fixed source, and say that for every vertex v we are guaranteed that the distance $\delta(s, v)$ never increases due to an edge deletion in G. Then, the* total *update time of* **ES**(G, s, d) *over the entire sequence of edge updates is $O(m \cdot d + \Delta)$, where $m = \text{MAX-EDGES}(G)$, and Δ is the total number of edge changes.*

In our incremental algorithm we invoke the incremental version of the Even-Shiloach algorithm described above. We will therefore need to make sure that distances never increases during edge deletions.

Similarly to the decremental case, we maintain the threshold graph G_τ described in Section 4.4. The only difference is that we need to maintain G_τ incrementally. Initially the graph is empty and all nodes are τ-light. As edges are added to G, some of these nodes may become τ-heavy (note that once a node becomes τ-heavy it will always remain τ-heavy as edges may only be added to G). We maintain the connected components of the τ-heavy nodes using again the result of Holm *et al.*[Holm et al., 2001] that is fully dynamic and so works in the incremental setting as well (with the same time bounds).

The graph G_τ is the same as in the decremental algorithm: for every connected component C in $G[\text{HEAVY}]$, G_τ contains a new node c that corresponds to this heavy connected component, with edges of weight $1/2$ from c to every vertex in C. As new nodes in G become τ-heavy, components in $G[\text{HEAVY}]$ can merge; once two connected components in $G[\text{HEAVY}]$ merge, their corresponding

nodes c and c' are also merged by picking the smaller connected component, say c', and replacing all edges (c', v) with edges (c, v). Similarly to the analysis in Section 4.4 a node may belong to the smaller component at most $\log n$ times.

Since we maintain exactly the same threshold graph G_τ as in Section 4.4, the rest of the algorithm and correctness analysis are identical to the decremental case. Moreover, using arguments analogous to those in Lemma 10, it is not hard to see that distances in G_τ never increase. The asymptotic bound of the total update time of our incremental algorithm is the same as that of the decremental one.

4.7 Conclusions

We presented the first *deterministic* partially dynamic algorithm for single source shortest distances that goes beyond the $O(mn)$ total update time time barrier. There remain many open problems concerning the gap between randomized and deterministic algorithms for this problem. For starters, the total update time of our deterministic algorithm is $O(n^2)$, while the best randomized algorithm of Henzinger *et al*.[Henzinger et al., 2014a] has an almost optimal total update time of $\widetilde{O}(m^{1+o(1)})$. The most salient open question is thus whether we can match this bound with a deterministic algorithm. As a less ambitious starting point, we need a result that improves over $O(mn)$ for *sparse* graphs, i.e. an algorithm with total update time $O(mn^{1-c})$ for some constant c.

Another limitation of our algorithm is that it only works for unweighted undirected graphs. For weighted graphs, the randomized algorithm of [Henzinger et al., 2014a] achieves the same total update time of $\widetilde{O}(m^{1+o(1)})$ as long as weights are polynomial in n. There are also several randomized algorithms that manage to go (slightly) beyond the $O(mn)$ bound in *directed* graphs, with the state of art achieving $O(mn^{.9})$. The existence of deterministic algorithm with total update time $o(mn)$ for either directed or weighted graphs remains open.

Finally, the gap between randomized and deterministic algorithms exists for many other variations of dynamic shortest paths (both exact and approximate), and closing this gap as much as possible remains an important goal in the field.

Chapter 5

All Pairs Shortest Paths in Directed Weighted Graphs

Publication History: This chapter presents a result that was originally published in STOC 2013 [Bernstein, 2013], where it received the best student paper award. A full version was then published in the SICOMP special issue for STOC 2013 [Bernstein, 2016]

As discussed in Section 1.2, fully dynamic all pairs shortest paths (APSP) has proven to be quite a difficult problem. The trivial algorithm simply recomputes APSP after every update, which yields update time $\widetilde{O}(mn)$ and constant query time. The state of the art improves this to update time $\widetilde{O}(n^2)$. This is the best that is known even for undirected, unweighted graphs; all existing algorithm to go beyond $\widetilde{O}(n^2)$ update time either have a polynomial query time, or an approximation ratio of at least 2.

For this reason, just as with dynamic single source shortest paths, researchers turned to developing more efficient algorithms for the *partially* dynamic setting (decremental or incremental). In this setting, Baswana *et al.* [Baswana et al., 2007] presented an algorithm with amortized update time $\widetilde{O}(n^3/m)$ and constant query time in directed unweighted graphs. This remains the state of the art for exact shortest distances. In the same paper, they showed that the amortized update time can be reduced to $\widetilde{O}(n^2/\sqrt{m})$ if we allow a $(1+\epsilon)$ approximation. In *undirected* unweighted graphs, Roditty and Zwick showed how to reduce the amortized update time to $\widetilde{O}(n)$, again with a $(1+\epsilon)$

approximation. All of the above algorithms are randomized and non-adaptive, but very recently Henzinger *et al.* presented a *deterministic* version of the earlier result of Roditty and Zwick: $\widetilde{O}(n)$ update and constant query time in unweighted undirected graphs.

Note that all the above results for this problem are tending towards a natural $\widetilde{O}(n)$ amortized update time barrier. The reason for this is that if a decremental (resp. incremental) algorithm spends $\widetilde{O}(n)$ time per update, then it is spending $\widetilde{O}(mn)$ total update time over the entire sequence of deletions (resp. insertions), where $m = \text{MAX-EDGES}(G)$ is the number of edges in the original (resp. final) graph. But barring fast matrix multiplication, there is no $o(mn)$ algorithm for computing even a single instance of APSP (modulo sub-polynomial improvements), even if the graph is undirected and unweighted and we allow $(1+\epsilon)$ approximate distances. In other words, a partially dynamic algorithm with $\widetilde{O}(mn)$ total update time manages to solve APSP on *every* instance of the evolving graph in the same total time (up to log factors) as solving just a single instance. Henzinger *et al.* [Henzinger et al., 2015c] formalize this intuition and prove a $\Omega(mn)$ conditional lower bound on the total update time, even for unweighted undirected graphs and a $(1+\epsilon)$ approximation.

Thus, the algorithms of Roditty and Zwick[Roditty and Zwick, 2012] and Henzinger *et al.* [Henzinger et al., 2013] achieve the best update time we can hope for with a $(1+\epsilon)$ approximation. However, both these results only work in undirected unweighted graphs. This raises the question of whether we can achieve $\widetilde{O}(mn)$ total update time for a more general case. Directed graphs? Weighted graphs? Can we remove the $(1+\epsilon)$ approximation? I developed an algorithm that answers the first two questions affirmatively.

5.1 Our Results

Recall that in weighted graphs, a decremental update sequence can delete edges and increase edge weights, while an incremental update sequence can insert edges or decrease edge weights (See Definition 1).

Theorem 1 *Let G be a directed graph with real positive weights. There is an algorithm for $(1+\epsilon)$ approximate decremental or incremental APSP with the following bounds: query time $O(1)$ (a single table look up), and* total *update time $O(mn\log^4(n)\log(nR)/\epsilon + \Delta)$, where $m =$ $\text{MAX-EDGES}(G)$, Δ is the total number of update operations, and R is the ratio of the heaviest*

edge weight to appear in G at any point in the update sequence to the lightest such edge weight. For unweighted graphs the running time is slightly smaller: $O(mn\log^4(n)\log\log(n))$. The update procedure is randomized (Monte Carlo), and assumes an oblivious non-adaptive adversary.

Note that the $O(\Delta)$ factor in our total update time has nothing to do with the particularities of our algorithm, but is simply an unavoidable constant time per update (no matter what we do, we cannot avoid looking at every update). The only reason $O(\Delta)$ did not come up in the other decremental (resp. incremental) algorithms mentioned above is because they only worked for unweighted graphs, in which we always have $\Delta \leq m$ because every update deletes (resp. inserts) an edge. But in weighted graphs, we can no longer bound the number of updates. It may thus appear strange that we continue to analyze our algorithm in terms of *total* update time, but the basic idea is that our algorithm in fact spends a total of only $\widetilde{O}(mn\log R/\epsilon)$ time processing updates that might actually be relevant (*i.e.* might actually change some approximate shortest distance); for example in the decremental setting, it is not hard to see that since distances only increase, any given $x-y$ distance can increase by a $(1+\epsilon)$ factor at most $\log_{(1+\epsilon)}(nR) = O(\log(nR)/\epsilon)$ times, so there are at most $O(n^2\log(nR)/\epsilon)$ relevant updates that require processing. (Of course we do not know ahead of time which updates are relevant, but this difficulty is not hard to deal with.) The additional $O(1)$ time per update then corresponds to merely throwing away the irrelevant updates.

Note that $\log(nR) = O(\log(n))$ as long as weights are polynomial in n. Thus, our algorithm achieves the desired $\widetilde{O}(mn/\epsilon)$ total update time for directed graphs with weights polynomial in n, as compared to the previous state of the art of Roditty and Zwick [Roditty and Zwick, 2004a], which only achieved this bound for undirected unweighted graphs. In fact, ours is the first non-trivial algorithm for decremental or incremental APSP in weighted graphs (though previous ones could handle small integer weights). Another advantage of our algorithm over the one of Roditty and Zwick is that although their query time was constant, it still required an involved process, whereas ours is simply a table lookup. One obvious benefit of this is that in real world scenarios, one often wants to keep query time as small as possible. Another benefit is that in many settings, when an update occurs, we want to quickly find out all pairs that were affected by it. The involved query procedure of Roditty and Zwick would require them to separately check each pair, so that even if only a few pairs were affected by the update, the algorithm would still require a prohibitive $\widetilde{O}(n^2)$ time to find those pairs. Our algorithm, however, could just return all changed entries of our distance matrix;

this requires time O(number of changed entries), and so does not increase the asymptotic update time.

There is a standard reduction from decremental APSP algorithms to fully dynamic APSP algorithms with a query-update trade off. Thus, our improved decremental algorithm leads to a new fully dynamic one. For any $T \leq \sqrt{n}$, we can maintain $(1+\epsilon)$-APSP in directed graphs with amortized update time $\widetilde{O}(\frac{mn \log R}{T\epsilon})$ and query time $O(T)$. This trade-off was previously only possible for undirected unweighted graphs. The decremental to fully dynamic reduction was originally introduced by Henzinger and King [Henzinger and King, 1995], and has since been used in several dynamic shortest paths papers ([Roditty and Zwick, 2004b; Roditty and Zwick, 2004a]). Our use of this reduction is more or less identical to previous ones, but a few of the details differ, so we offer a more detailed discussion at the end of this chapter (Section 5.8.3).

Theorem 1 applies to both a decremental and an incremental update sequence; the algorithms for these two cases are basically the same, but with all the relevant parameters flipped (insertions instead of deletions, distance decreases instead of increases, and so on). *For simplicity, in the main body of the paper we only prove Theorem 1 for a decremental update sequence*. We then argue that the same approach works in the incremental setting at the very end of Section 5.8.2.

Section 5.4 outlines the basic approach of our algorithm, but first we present notation and existing work in Sections 5.2 and 5.3. Section 5.5 introduces a simpler version of our algorithm for the sake of intuition, and Section 5.6 presents our final algorithm, which proves Theorem 1 above. Section 5.7 presents in full detail an existing result that we rely heavily upon, as well a new improvement on this result that reduces the algorithm's dependence on Δ. Finally, Section 5.8 touches upon some final details, including applications of our algorithm to the incremental and full dynamic setting.

New Work Published After our Results: Our algorithm remains the state of the art for decremental $(1+\epsilon)$-approximate APSP. For the special case of undirected unweighted graphs, Henzinger *et al.* published a paper in FOCS 2013 [Henzinger et al., 2013] that achieves similar bounds with a *deterministic* algorithm: the total update time is again $\widetilde{O}(mn/\epsilon)$, while the query time is $O(\log\log(n))$.

5.2 Preliminaries

Let $G = (V, E)$ be a directed graph with real positive weights subject to a decremental update sequence. As we process our updates, G always refers to the *current* version of the graph. Let $m = \text{MAX-EDGES}(G)$ be the number of edges in the *initial* graph, and let n be the number of vertices (which does not change); we assume that $m = \Omega(n)$. Given any vertices x, y, let (x, y) be the edge between them (if it exists), and let $w(x, y)$ be the weight of this edge. Let $\pi(x, y)$ be the shortest $x - y$ path in G (if one exists); if there are multiple shortest paths from x to y we can use any tie breaking strategy which ensures that any subpath of a shortest path is itself a shortest path (For an example, see section 3.4 of [Demetrescu and Italiano, 2004]). Define $\delta(x, y)$ to be the length of $\pi(x, y)$, or ∞ if no $x - y$ path exists. We assume that all edge weights are positive. We define the *hop-length* of a path P, denoted $h(P)$, to be the number of edges on P, and we let $h(x, y) = h(\pi(x, y))$. For any h, we define $\pi^h(x, y)$ to be the shortest path from x to y that uses at most h edges (if one exists), and we define $\delta^h(x, y)$ to be the length of $\pi^h(x, y)$, or ∞ if this path does not exist. Given a graph G' different from G, we define $\pi_{G'}(x, y), \delta_{G'}(x, y)$, $\pi^h_{G'}(x, y)$, and $\delta^h_{G'}(x, y)$ to be the corresponding paths and distances in G'.

Many of our running times are expressed in terms of the variables Δ and R. We define Δ to be the total number of updates made to the graph over the course of the entire dynamic sequence. We define R to be the ratio of the largest weight in the graph *at any point in the update sequence* to the smallest such weight. More formally, we define C, c, and R as follows:

- $C = \max_\tau \max_{(u,v) \in E} \{w(u, v) \text{ at time } \tau\}$
- $c = \min_\tau \min_{(u,v) \in E} \{w(u, v) \text{ at time } \tau\}$
- $R = C/c$

Note that as long as weights are polynomial in n, $\log R = O(\log(n))$. Finally, we say that output $\delta'(x, y)$ is α-approximate if $\delta(x, y) \leq \delta'(x, y) \leq \alpha\delta(x, y)$. We say that a path $P(x, y)$ is α-approximate if its weight is an α-approximation of $\delta(x, y)$.

For the sake of simplicity, we make a few minor assumptions about the graph and the update sequence.

- We assume that R is known in advance; in Section 5.8.1, we show that this assumption can easily be removed by continually updating an approximate guess for R.
- We model the deletion of an edge by increasing its weight to ∞. This is not quite rigorous, as then $\log(R)$ also becomes infinite. To resolve this, we model the deletion of an edge by raising its weight to large number U^*. We ensure that at all times $U^* \geq 2nC^*$, where C^* is the largest non-infinite weight in the current graph. Thus, if a query ever returns a shortest $x - y$ distance $\geq U^*$, this clearly corresponds to there being no path from x to y in the graph. As edge weights in the graph increase, U^* might come to be less than $2nC^*$, in which case we repeatedly double it until it is large enough. It is not hard to see that modeling deletions in this way does not affect the asymptotic running time. Firstly, repeated doubling can never cause U^* to be greater than $4nC$, so adding edges of weight U^* increases R by only an $O(n)$ factor, so the $\log(nR)$ term is not affected. Secondly, U^* doubles at most $O(\log(4nC/c)) = O(\log(nR))$ times, so the dummy weight on each deleted edge increases at most $O(\log(nR))$ times, and the total number of additional updates is at most $O(m\log(nR))$; the change to $O(\Delta)$ is thus well within the $\widetilde{O}(mn\log(nR))$ total update time.
- We assume the graph is connected at all times. Our algorithm doesn't actually rely on this, but it obviates the need for an analysis of edge cases. We can ensure this by adding a super source s^* with an edge of weight U^* to and from every vertex; as above, U^* might increase as edge weights in the original graph increase. A shortest distance $\geq U^*$ once again indicates that no path exists. The number of edges is still $O(m)$, and the number of new updates is only $O(n\log(nR))$.

5.3 Hop Distances and the Even and Shiloach Tree

As with our results on dynamic SSSP, the basic building block of our algorithm is the Even and Shiloach tree, which decrementally maintains SSSP from a fixed source s up to distance d (see Lemma 1).

In an earlier paper [Bernstein, 2009], we developed a simple but powerful generalization of the above Even and Shiloach tree, though at the cost of a $(1+\epsilon)$ approximation error. Loosely speak-

ing, we showed that instead of maintaining a shortest path tree up to distance d, we can efficiently maintain it up to a hop length h. We refer to this algorithm for decrementally maintaining approximate single source shortest distances as the *h*-SSSP algorithm. We now formally present the result achieved by *h*-SSSP, which we use a black box throughout most of the paper, leaving the details of *h*-SSSP itself for Section 5.7.

Theorem 2 *[Bernstein, 2009] Given a source s and a hop distance h,* h-*SSSP decrementally maintains distances $\delta'(s,v)$ to each vertex v, such that we always have $\delta(s,v) \leq \delta'(s,v) \leq (1+\epsilon)\delta^h(s,v)$. Moreover, after every update* h-*SSSP can return a list of all vertices v for which $\delta'(s,v)$ changed due to that update, without affecting the asymptotic update time. The total update time of* h-*SSSP over all deletions and weight-increases is $O(mh\log(n)\log(nR)/\epsilon + \Delta)$ for weighted graphs and a slightly faster $O(mh\log(n)\log\log(n)/\epsilon)$ for unweighted ones.*

Remark: Note that in the theorem above $\delta'(s,v)$ itself may correspond to a path with more than h edges; the algorithm is not concerned with the length of the output path. The only guarantee is merely that $\delta'(s,v)$ is a good approximation to $\delta^h(s,v)$, which is equal to $\delta(s,v)$ as long as $h(s,v) \leq h$. Thus, we can think of our algorithm as maintaining $(1+\epsilon)$-distances from s up to hop-length h.

When we say that we "run" the *h*-SSSP algorithm from (to) vertex s, this refers not merely to an initialization step, but rather to the whole dynamic procedure. In other words, it means that we maintain approximate distances $\delta'(s,v)$ to (from) each vertex v over all deletions to come. By Theorem 2 the total cost of running the *h*-SSSP algorithm is $\widetilde{O}(mh\log R/\epsilon)$.

5.4 The Basic Approach

The basic outline of our approach is similar to one Bernstein used in two earlier papers [Bernstein, 2009; Bernstein, 2012], though except for the *h*-SSSP algorithm essentially all the details differ. The advantage of *h*-SSSP over King's $O(md)$ algorithm algorithm [King, 1999] (see Lemma 1) is that the latter maintains a shortest path tree up a to certain distance, whereas *h*-SSSP maintains it up to a certain hop-length, and is hence barely dependent on the weights of the edges. (The running time of *h*-SSSP does depend on $\log R$, but this is only a logarithmic dependence on the weight, as compared to King's linear dependence.) This change of focus from weighted distance to hop length

is obviously crucial for weighted graphs, but it is in fact equally important in unweighted graphs. Both d and h can initially be as large as $n-1$, but whereas the distance between a pair vertices is inherent to the graph, the *hop* distance can easily be decreased by adding shortcuts to the graph.

Suppose that we already knew the shortest distance $\delta(v, w)$. We could then add a new edge (v, w) of weight $\delta(v, w)$; this would not change any of the distances in the graph but it would reduce $h(v, w)$ to one. It would also reduce the hop-length of any path that used $\pi(v, w)$ as a subpath. This observation suggests the following approach: we construct a large number of shortcut edges to reduce hop-distances all across the graph, which would allow us to efficiently run the *h*-SSSP algorithm. The problem is that in order to create shortcut edges we need to have already computed $\delta(v, w)$; moreover, as the graph changes so do the shortest distances in the graph, so dynamically maintaining correct weights on the shortcut edges requires maintaining the distances $\delta(v, w)$.This leaves us in the position of trying to maintain shortest paths by first maintaining other shortest paths.

Bernstein previously applied the idea of creating shortcut edges to reduce the hop-distances in two papers on *undirected* graphs [Bernstein, 2009; Bernstein, 2012]. (Note that these papers were not actually on the problem of decremental shortest paths; they just used the same basic approach of shortcutting edges and then running an algorithm for small hop distances.) The main idea was to apply results from the rich field of spanners and emulators, which shows that one can approximate all distances in a graph with a small number of edges. Bernstein modified this result to show that a small number of shortcut edges can approximate all distances while effectively maintaining short hop-lengths as well: that is, for any pair (x, y), one can always patch together an approximate shortest path from x to y using just a small number of these shortcut edges. One still had to maintain the distances of the shortcut edges directly, but these shortcut distances were only a small subset of all distances.

Applying shortcuts to *directed* graphs, however, is significantly harder because in this case it is essentially impossible to approximate all-pairs shortest distances with a sparse spanner or emulator. The key feature of undirected graphs is that if u and v are nearby, then shortest paths from u are approximately the same as those from v, and we can therefore handle a whole cluster of nearby vertices with a single representative. Such clustering does not work in directed graphs because a short $u-v$ path does not imply a short $v-u$ path.

Shortcuts are thus much more difficult to apply in directed graphs, and as far as we know, this is the first paper to do so in the dynamic setting. (In the static setting, Thorup's algorithm for distance oracles in *planar* graphs [Thorup, 2004] uses them extensively, and there are several papers that use them to achieve faster running times in practice – see [Abraham et al., 2010] for an overview.) Because directed graphs do not allow for clustering, we end up having to maintain shortcut edges for essentially all pairs, which seems to bring us back to the original predicament of trying to maintain all pairs shortest paths by first maintaining all pairs shortest paths. The key lies in doing the computation in the proper order. The *h*-SSSP algorithm already provides an efficient way to maintain shortest paths of small hop length, so we start by shortcutting those. This reduces the hop-length of the other paths because shortcutting a subpath of some path $\pi(x, y)$ reduces $h(x, y)$. The reduced hop-lengths allow *h*-SSSP to efficiently maintain a larger set of distances, which in turn leads to more shortcut edges and a further reduction in hop-lengths. In iterating this process, we continually shortcut the small-hop subpaths of large-hop paths, to the point where the latter themselves become small-hop and easy to shortcut.

This paper is substantially less technical than the papers which applied the shortcut edge approach to undirected graphs, because we do not rely on the heavy machinery of emulators and clustering. One advantage of this is that it forces us to explore the limits of the core approach itself; directed graphs do not seem to offer much structure, so all we can really do is iteratively use the basic *h*-SSSP algorithm to create more and more shortcut edges. We now present some of the key lemmas and definitions used throughout our algorithm.

Definition 13 *We say that an edge (u, v) with weight $w(u, v)$ is an exact shortcut edge if $w(u, v) = \delta(u, v)$. We say that it is an α-shortcut if $\delta(u, v) \leq w(u, v) \leq (1 + \epsilon)^{\alpha}\delta(u, v)$.*

Definition 14 *Let G^* be the graph G with some shortcut edges added. Given some $x - y$ path P, we define the G^*-α-reduction of P to be the $x - y$ path of smallest hop length whose edges are either part of P itself, or α-shortcuts of sub-paths of P. We refer to the hop-length of this reduction as the G^*-α-reduced hop-length of P.*

It is easy to see that the G^*-α-reduction of a shortest path $\pi(x, y)$ is a $(1+\epsilon)^{\alpha}$-approximate shortest path. If all the shortcut edges were exact, then the reduced path would have the same length as the

original one. As is, all the shortcut edges are off by a factor of at most $(1+\epsilon)^\alpha$, so the overall weight is off by at most $(1+\epsilon)^\alpha$.

Lemma 13 *If the G^*-α-reduction of some path $\pi(x,y)$ has fewer than h edges, then running the* h-*SSSP algorithm on G^* up to hop-length h yields a $(1+\epsilon)^{\alpha+1}$-approximation to $\delta(x,y)$.*

Proof: Since the G^*-α-reduction of $\pi(x,y)$ has fewer than h edges, we know that $\delta^h_{G^*}(x,y) \leq \delta(x,y)(1+\epsilon)^\alpha$. But by Theorem 2, the *h*-SSSP algorithm yields a $1+\epsilon$ approximation to $\delta^h_{G^*}(x,y)$, which is a $(1+\epsilon)^{\alpha+1}$ approximation to $\delta(x,y)$. □

We now present a well known sampling lemma that is used throughout our algorithm.

Lemma 14 *Let S be a set of r vertices chosen uniformly at random from V, and let P be some path in G with at least $cn\ln(n)/r$ vertices (c is a constant of our choosing). Then, with probability at least 1 - n^{-c}, the path P contains at least one vertex in S.*

Proof: For any particular vertex $v \in P$, we have that $Pr[v \in S] = r/n$. Thus,

$$Pr[S \bigcap P = \emptyset] \leq (1-r/n)^{|P|} \leq (1-r/n)^{cn\ln(n)/r} < n^{-c}$$

□

For simplicity of presentation, we will fix $c = 9$. The following corollary is then a direct consequence of the union bound:

Corollary 4 *Let S be a set of r vertices chosen uniformly at random from V, and let $\mathcal{P}$ be a set of $\leq n^4$ paths in G, each of which contains at least $9n\ln(n)/r$ vertices. Then, with probability at least $1-n^{-5}$, every path in $\mathcal{P}$ contains at least one vertex in S.*

Remark: Our algorithm only requires the above sampling lemma to hold for $O(n^3\log(n))$ paths (n^2 shortest paths in $n\log(n)$ different graphs used by our algorithm), so we can use the above corollary. Now, note that since we are assuming an adversary that is oblivious to our random choices, the set of r sampled vertices will be random from the perspective of each version of the graph throughout the update sequence. Thus, Corollary 4 holds with probability at least $1-n^{-5}$ for each version of the graph, so by the union bound, it will hold with high probability for *all* versions within the first $O(n^4)$ updates.

Now, as we discuss in the next few sections, since we are looking for a $(1+\epsilon)$ approximation we only need to register a change to an edge weight when it has increased by at least a $(1+\epsilon)$ factor. The total number of updates that our algorithm actually registers (instead of simply throwing away) is thus $O(m \log(nR))$. On the reasonable assumption that $\log R \leq n^3/m$, the number of updates is $O(n^4)$, so by the above discussion setting $c = 9$ will ensure that the corollary holds with high probability throughout all versions of the graph. More generally, it is not hard to see that if $\log R = O(n^x)$ then setting $c = x + 5$ is sufficient. In the extremely unlikely case that $\log R$ is not polynomial in n we would need to set c to be $O(\log\log(R)/\log(n))$, and the running time of the whole algorithm would be multiplied by this factor. In this case, however, our running time is already super-polynomial, and we would likely be better of using an algorithm that works for general weights. All in all: *we assume for the rest of the paper that Corollary 4 holds throughout all versions of the graph.*

5.5 A Simplified Not Quite O(mn) Algorithm

We now present an algorithm for decrementally maintaining $(1+\epsilon)$-approximate shortest paths in directed graphs with a total update time of $\widetilde{O}(mn^{4/3} \log R/\epsilon)$. We later improve this to $\widetilde{O}(mn \log R/\epsilon)$, but even this preliminary approach already yields the first efficient decremental algorithm for polynomial weights, and for sparse m it even beats the previous state of the art of $\widetilde{O}(n^2\sqrt{m}/\epsilon)$ for *unweighted* directed graphs.

We maintain approximate APSP by separately maintaining distances from different sources using the *h*-SSSP algorithm. The *h*-SSSP algorithms that we use are grouped into three distinct layers. The first layer of *h*-SSSP algorithms runs on the main graph G, and maintains approximate distances from only a small number of sources, up to a limited h. We use these distances to construct shortcut edges, which reduce hop-lengths in G. Our second layer runs on this new graph with shortcut edges added, and is thus able to efficiently maintain a larger subset of approximate shortest distances. We use these distances to create even more shortcut edges, which further reduce hop-lengths. Our third and final layer computes all-pairs shortest distances by running *h*-SSSP from every vertex v; this remains efficient because thanks to the shortcut edges from the second layer, we only have to run *h*-SSSP up to a small hop length h.

Recall that the h-SSSP algorithm is not a one-time computation, but rather maintains distances dynamically over *all* updates, so all our overall algorithm needs to do is set up the necessary h-SSSP algorithms in the very beginning, and let them run. Each of the three layers is responsible for maintaining its own distance matrix, which is simply an aggregate of the distances maintained by all of the h-SSSP algorithms in that layer. As we process our updates, the distances in these matrices will increase, which will lead to weight-increases in the corresponding shortcut edges.

Dependence on Δ: Our primary concern with respect to running time is to maintain distances in total time $\widetilde{O}(mn\log(R))$; the whole apparatus of shortcut edges was developed for this purpose. But a secondary concern is ensuring that every update incurs an additional overhead of only $O(1)$, i.e., that the dependence on Δ is only $O(\Delta)$. We show in Section 5.7.1 that the h-SSSP building block incurs this optimal overhead of $O(1)$ time per update. The problem is that our *all*-pairs shortest path algorithms runs h-SSSP algorithms from $O(n)$ different sources, which seems to lead to an overhead of $O(n)$ per update. We resolve this problem by noting that although the total number of updates can be arbitrarily large, most of them will only increase weights by a small amount. Any such insignificant update can simply be ignored in $O(1)$ time, i.e. not processed by any of the h-SSSP algorithms. To this end, we define a function which allows us to only register updates that increase the weight by a $(1+\epsilon)$ factor.

Definition 15 *For any number x, let $Round_{(1+\epsilon)}(x)$ be $(1+\epsilon)^{\lceil \log_{(1+\epsilon)}(x) \rceil}$ (the smallest power of $(1+\epsilon)$ that is $\geq x$).*

5.5.1 The Algorithm

Main Setup:

1. Use Dijkstra to compute shortest paths from all $v \in V$ in the original graph, before any updates occur.
2. Sample $n^{2/3}$ vertices uniformly at random, and let A be the resulting set.
3. For all $a \in A$, run the h-SSSP algorithm from a up to hop-length $10n^{2/3}$. Recall that this maintains distances from a over all deletions to come. Store the distances maintained in a

matrix $D_{A\times A}$, which is initialized with the distances from Step 1. ($D_{A\times A}$ stores approximate distances between nearby vertices in A)

4. Let G^* be the graph G plus a shortcut edge added for each pair $(a,b) \in A \times A$. Set shortcut (a,b) to have weight $Round_{(1+\epsilon)}(D_{A\times A}[a,b])$.

5. For each $a \in A$, run the *h*-SSSP algorithm *to* a in G^* up to hop-length $10n^{1/3}\ln(n)$. Store the results in a matrix $D_{V\times A}$, which is initialized with the distances from Step 1. ($D_{V\times A}$ stores approximate distances to all vertices in A)

6. For each $v \in V$, let G_v be the graph G with a shortcut edge added from v to every vertex in $a \in A$. Set shortcut (v,a) to have weight $Round_{(1+\epsilon)}(D_{V\times A}[v,a])$.

7. For each $v \in V$, run the *h*-SSSP algorithm from v up to hop-length $10n^{1/3}\ln(n)$ in G_v. Store the results in a matrix $D_{V\times V}$, which is initialized with the distances from Step 1. This is our final distance matrix.

Query(v,w): To approximate $\delta(v,w)$, simply return $D_{V\times V}[v,w]$.

Update Step: Our whole algorithm is essentially contained in the *h*-SSSP algorithms of the main setup. The only catch is that many of these algorithms are not running on the main graph G, but on a graph that also contains some shortcut edges. It is crucial for correctness that we dynamically maintain correct distances for these shortcuts (as $\delta(x,y)$ changes, the weight of an $x-y$ shortcut should also change). Here is the order in which we process an update increase-weight(x,y) : $w_{\text{old}}(x,y) \to w_{\text{new}}(x,y)$ (an edge deletion can be modeled as increasing the weight to ∞).

- If $Round_{(1+\epsilon)}(w_{\text{new}}(x,y)) = Round_{(1+\epsilon)}(w_{\text{old}}(x,y))$, the algorithm simply throws away the update in $O(1)$ time and does not move on to the steps below. Note that because of this, all edge weights in G are effectively only $(1+\epsilon)$-approximate.

- Else, if $Round_{(1+\epsilon)}(w_{\text{new}}(x,y)) > Round_{(1+\epsilon)}(w_{\text{old}}(x,y))$, input the update increase-weight(x,y) into all of the *h*-SSSP algorithms from Step 3, which might cause some of the distances maintained in $D_{A\times A}$ to change.

- For all entries for which $Round_{(1+\epsilon)}(D_{A\times A}[a,b])$ has increased, we increase the weight of corresponding shortcut edge (a,b) in G^* (Step 4) to the new $Round_{(1+\epsilon)}(D_{A\times A}[a,b])$.

- Input the original increase-weight(x, y), as well as all the shortcut-edge weight increases in G^* from the previous step into the h-SSSP algorithms of Step 5. This might cause changes in $D_{V \times A}$.
- For all $v \in V$, for all entries for which $Round_{(1+\epsilon)}(D_{V \times A}[v, a])$ has increased, we increase the weight of corresponding shortcut edge (v, a) in G_v (Step 6) to the new $Round_{(1+\epsilon)}(D_{V \times A}[v, a])$.
- Input increase-weight(x, y), as well as all the shortcut-edge weight increases from the previous step into the h-SSSP algorithms of Step 7. This might cause some changes to $D_{V \times V}$, which makes sense, since our final distance matrix should be changing over time.

5.5.2 Running Time Analysis

The key observation is that the running time for a single update step is just the time to update the distance matrices $D_{A \times A}, D_{V \times A}$, and $D_{V \times V}$ via the h-SSSP algorithms of the main setup. We also have to increase shortcut edge weights, but every such increase corresponds to a change in $D_{A \times A}, D_{V \times A}$, or $D_{V \times V}$, and so can be charged to the h-SSSP algorithms. Thus, the total update time of our algorithm is simply the sum of the total update times of all of its constituent h-SSSP algorithms. We now proceed to analyze this sum.

Note that the rate at which edge-weights are increased may vary greatly depending on whether we are dealing with G, G^*, or G_v; a single update only increases one edge weight in the main graph G, but this can lead to a large number of shortcut-edge weight increases in G_v. All that matters, however, is that since G only sees weight *increases* (by definition of this being a decremental algorithm), G^* and G_v will also only see weight increases; shortcut edge weights are based on distances in G, so since the latter are only getting larger, the same is true of the former. Thus, the algorithm is decremental from the perspective of each graph involved, which allows us to side step the analysis of how many updates occurs in each graph; all that matters is that in each graph, the h-SSSP algorithm will always have *total* update time $\widetilde{O}(m'h \log R/\epsilon)$, where m' is the number of edges on the graph in question (m' is larger than m when we add shortcut edges).

(Technical note: the running time of h-SSSP depends on $\log(nR)$, but the h-SSSP algorithms run on graphs with weighted shortcut edges, and R might be slightly larger in these shortcutted graphs than in the original graph. But since every shortcut edge correspond to a $(1 + \epsilon)$ approximate

distance in the original graph, no shortcut will have weight larger than $R' = (1+\epsilon)nR$, so the running time will not be affected because $\log(nR') = O(\log(nR))$. Thus, we just assume the same R throughout.)

- Step 3 runs the h-SSSP algorithm from $n^{2/3}$ vertices up to $h = O(n^{2/3})$, which by Theorem 2 yields a total update time of $\widetilde{O}(n^{2/3}mn^{2/3}\log R/\epsilon) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.
- Step 5 runs the h-SSSP algorithm to $n^{2/3}$ vertices up to $h = \widetilde{O}(n^{1/3})$. Note however that the graph G^* has $(m + n^{4/3})$ edges because of the shortcut edges for $A \times A$. This yields a total update time of $\widetilde{O}(n^{2/3}(m + n^{4/3})n^{1/3}\log R/\epsilon) = \widetilde{O}(mn + n^{7/3}) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.
- Step 7 runs the h-SSSP algorithm from n vertices up to $h = \widetilde{O}(n^{1/3})$. Each graph G_v has $m+n^{2/3} = O(m)$ edges. Thus, the total update time is $\widetilde{O}(nmn^{1/3}\log R/\epsilon) = \widetilde{O}(mn^{4/3}\log R/\epsilon)$.

Dependence on Δ*:* Our algorithm is built on many h-SSSP algorithms that process many difference edge weight increases: increases to weights of edges in G but also to the various shortcut edges. Let us define any such weight increase: $w_{\text{old}}(x, y) \to w_{\text{new}}(x, y)$ to be *significant* if $Round_{(1+\epsilon)}(w_{\text{new}}(x, y)) > Round_{(1+\epsilon)}(w_{\text{old}}(x, y))$. It is clear from the description of our update step that any weight increase which is *not* significant is thrown away in $O(1)$ time and not processed by any h-SSSP algorithms; this is where the $O(\Delta)$ term comes from. We now show that that the total number of times an h-SSSP algorithm processes a *significant* update is $O(mn\log(nR)/\epsilon)$, which is within our desired update bounds.

Recall the definitions of c, C, and R from Section 5.2. The weight of any edge in G ranges from c to C. Every shortcut edge weight is a $(1+\epsilon)$ approximation to some shortest distance in G, so it ranges from c to at most $(1+\epsilon)nC \leq 2nC$. Thus, since edge weights only increase, the number of significant updates on any given edge (shortcut edges included) is at most $O(\log_{1+\epsilon}(2nC/c)) = O(\log(nR)/\epsilon)$.

In particular, since the main algorithm runs $O(n)$ h-SSSP algorithms, and every edge in G (i.e. every non-shortcut edge) registers $O(\log(nR)/\epsilon)$ significant updates, the total number of times that an h-SSSP algorithm processes a significant update on an edge in G is $O(mn\log(nR)/\epsilon)$. G^* contains $O(n^{4/3})$ shortcut edges between vertices in A, and any significant update to one of these is processed by the $O(n^{2/3})$ h-SSSP algorithms running on G^*, leading to an additional $O(n^2\log(nR)/\epsilon)$

significant updates processed by an h-SSSP algorithm. Finally, for each vertex v the graph G_v contains $n^{2/3}$ shortcut edges but has only a single h-SSSP algorithm running on it, so the total number of times an h-SSSP algorithm processed a significant update to a shortcut edge in some graph G_v is only $O(n^{5/3}\log(nR)/\epsilon)$. Thus the total number of times that an h-SSSP algorithm processes a significant update is only $O(mn\log(nR)/\epsilon)$; all other updates are insignificant and thrown away in $O(1)$ time.

5.5.3 Approximation Error Analysis

Before proceeding, we observe a basic mathematical fact

Lemma 15 *For any positive real number $\epsilon < 1$, and any non-negative integer a, we have $(1+\frac{\epsilon}{2a})^a < 1+\epsilon$*

Proof: $(1+\frac{\epsilon}{2a})^a < (e^{\frac{\epsilon}{2a}})^a = e^{\epsilon/2} < 1+\epsilon$. □

We now prove that the distances stored in $D_{V\times V}$ are $(1+\epsilon)^6$-approximate. Using $\epsilon' = \epsilon/12$, by Lemma 15 we get a $(1+\epsilon/12)^6 \leq (1+\epsilon)$ approximation. Recall that $h(a,b)$, $\delta(a,b)$ and so on always refer to the *current* version of G. Recall also that G^* is the graph from Step 4 of the initialization phase, and G_v are the graphs from Step 6.

Lemma 16 *For any pair $(a,b) \in A \times A$, if $h(a,b) \leq 10n^{2/3}$ then there is a 3-shortcut from a to b in G^* (see Definition 13 for 3-shortcut).*

Proof: Recall that the weight of the shortcut edge is $Round_{(1+\epsilon)}(D_{A\times A}[a,b])$. By Theorem 2 the h-SSSP algorithm in Step 3 yields a $(1+\epsilon)$ approximation to $\delta^{10n^{2/3}}(a,b) = \delta(a,b)$; however since edge weights in G are only a $(1+\epsilon)$-approximation to their actual weight (because we only register updates that increase $Round_{(1+\epsilon)}(w(x,y))$), the total approximation factor is $(1+\epsilon)^2$. Thus, $D_{A\times A}[a,b]$ is $(1+\epsilon)^2$ approximate. The $Round_{(1+\epsilon)}$ function on the shortcut weights adds another $(1+\epsilon)$ approximation, leading to $(1+\epsilon)^3$ in total. □

Lemma 17 *For any pair of vertices $(v,a) \in V \times A$, the G^*-3-reduced hop-length of $\pi(v,a)$ is $\leq 10n^{1/3}\ln(n)$ (see Definition 14).*

Proof: We prove this by exhibiting a path of hop-length $\leq 10n^{1/3}\log(n)$ that only uses edges of $\pi(v,a)$ and 3-shortcuts of its subpaths. Let b be the first vertex in A on $\pi(v,a)$. By Lemma 14, there are at most $9n^{1/3}\ln(n)$ edges between v and b, so we just follow these directly. We now need to find a path from b to a.

We prove the following by induction: for any vertex $a' \in A$, there is a path from a' to a consisting of at most $\lceil h(a',a)/n^{2/3}\rceil$ 3-shortcuts of subpaths of $\pi(a',a)$.

- *Base Case:* if $h(a,a') \leq 10n^{2/3}$, then by Lemma 16 there is a 3-shortcut from a' to a.

- *Induction Step:* we now assume that the claim is true for all vertices $a' \in A$ for which $h(a',a) \leq i$, and prove that it then also holds when $h(a',a) = i+1$. If $h(a',a) \leq 10n^{2/3}$ we use the base case; otherwise, by Lemma 14 there is some vertex $a'' \in A$ on $\pi(a',a)$ that is between $n^{2/3}$ and $10n^{2/3}$ vertices away from a' (because this interval contains $9n^{2/3} \geq 9n^{1/3}\ln(n)$ vertices). Since $h(a',a'') \leq 10n^{2/3}$, there exists a 3-shortcut (a',a''); combining this 3-shortcut with the path of at most $\lceil h(a'',a)/n^{2/3}\rceil \leq (\lceil h(a',a)/n^{2/3}\rceil - 1)$ 3-shortcuts from a'' to a guaranteed by the induction hypothesis yields the desired path of 3-shortcuts from a' to a.

We can now prove the main Lemma: to get from v to a we first use $9n^{1/3}\ln(n)$ non-shortcut edges to get from v to the first vertex b on $\pi(v,A)$ that is in A; since $h(b,a)$ is trivially $\leq n$, we now take a path of at most $\lceil n/n^{2/3}\rceil = \lceil n^{1/3}\rceil$ 3-shortcuts from b to a. □

Corollary 5 *All the entries in $D_{V\times A}$ are $(1+\epsilon)^4$ approximate distances. This follows directly from Lemma 17 and Lemma 13.*

Lemma 18 *All the entries in $D_{V\times V}$ are $(1+\epsilon)^6$-approximate distances.*

Proof: We maintain $D_{V\times V}$ by running the *h*-SSSP algorithm on each G_v. We know that all the shortcuts in G_v are 5-shortcuts because their weights are obtained from $D_{V\times A}$; we get a $(1+\epsilon)^4$ error from $D_{V\times A}$, and another $(1+\epsilon)$ from the $Round_{(1+\epsilon)}$ function.

We can now show that the G_v-5-reduced hop-length of any $\pi(v,w)$ is $\leq 10n^{1/3}\log(n)$. If $h(v,w) \leq 10n^{1/3}\log(n)$ then this is trivially true. Otherwise, by Lemma 14 there must be a vertex in A on $\pi(v,w)$. Let a be the *last* such vertex, and note that again by Lemma 14, $h(a,w) \leq$

$9n^{1/3}\log(n)$. Thus, our 5-reduced path is just the 5-shortcut from v to a (created in Step 6), followed by the path $\pi(a, w)$. By Lemma 13, running h-SSSP up to $h = 10n^{1/3}\log(n)$ yields a $(1+\epsilon)(1+\epsilon)^5 = (1+\epsilon)^6$ approximation. □

5.6 The Final O(mn) Algorithm

Our $\widetilde{O}(mn \log R/\epsilon)$ algorithm is a direct extension of the $\widetilde{O}(mn^{4/3} \log R/\epsilon)$ algorithm above. The basic idea remains the same; we maintain some subset of the shortest distances, and use these to construct shortcut edges which lower hop-lengths and hence allow us to efficiently maintain more shortest distances, and so on. Once again the h-SSSP algorithms dynamically maintain distances from a source over *all* updates, so we just set them up in the beginning and let them run. In this version, however, instead of using three layers of h-SSSP algorithms, we use $\log(n)$ layers. We assume for simplicity that n is a power of 4.

Definition 16 *For the rest of this paper, we define q to be* $log(n)/2$.

Definition 17 *Define A_0 to be V. Construct A_1 by picking exactly half of the vertices from $A_0 = V$ uniformly at random (we round up if $|A_0|$ is odd). Construct A_2 by picking half the vertices from A_1 uniformly at random. Keep doing this until we reach $A_q = A_{(\log(n)/2)}$. Note that A_k contains $n/2^k$ random vertices from V, and that $|A_q| = \sqrt{n}$*

5.6.1 The Algorithm

<u>Main Setup:</u>

1. Use Dijkstra to compute, for all vertices v, shortest paths from v in the original graph G before any updates.

2. Construct the sets $A_0, A_1, ..., A_q$ as in Definition 17.

3. For each $v \in A_q$, run the h-SSSP algorithm to and from v up to $h = 10\sqrt{n}\log(n)$ on the main graph G. Store the results in $D_{A_q \times A_q}$. We initialize this matrix with the distances from Step 1. ($D_{A_q \times A_q}$ contains approximate distances between nearby vertices in A_q).

4. Let G_q be the graph G with a shortcut edge (v, w) added for each pair $(v, w) \in A_q \times A_q$. Set the weight of shortcut (v, w) to be $Round_{(1+\epsilon)}(D_{A_q \times A_q}[v, w])$.

5. For each $v \in A_q$, run the h-SSSP algorithm *to and from* v on the graph G_q up to $h = 10\sqrt{n}\log(n)$. Store the results in matrix D_q. As before, initialize D_q with the distances from Step 1 (D_q contains approximate distances to and from vertices in A_q).

6. For $k = q - 1$ down to 0

   - For each $v \in A_k$, we create a graph $G_{v,k}$, which is the graph G with shortcut edges (v, w) and (w, v) added for every $w \in A_{k+1}$. Set the weight of shortcut (v, w) to be $Round_{(1+\epsilon)}(D_{k+1}[v, w])$, and do the same for (w, v).
   - For each $v \in A_k$, run the h-SSSP algorithm *to and from* v in $G_{v,k}$ up to $h = 10\log(n)2^{k+1}$. Store the combined results for all $v \in A_k$ in matrix D_k (D_k contains approximate distances to and from vertices in A_k).

Query(v,w): To find an approximation to any $\delta(v, w)$, simply look up $D_0[v, w]$.

Update step: As in Section 5.5, all the work of our algorithm is done by the various h-SSSP algorithms of the main setup. All we need to describe here is the order in which we process some update increase-weight(x, y) : $w_{\text{old}}(x, y) \to w_{\text{new}}(x, y)$ (an edge deletion can be modeled as increasing the weight to ∞).

- If $Round_{(1+\epsilon)}(w_{\text{new}}(x, y)) = Round_{(1+\epsilon)}(w_{\text{old}}(x, y))$, the algorithm deems the update insignificant and throws it away in $O(1)$ time; i.e., it skips the steps below and does not process the update in any h-SSSP algorithm.

- Else, if $Round_{(1+\epsilon)}(w_{\text{new}}(x, y)) > Round_{(1+\epsilon)}(w_{\text{old}}(x, y))$, input the update increase-weight(x, y) into all of the h-SSSP algorithms from Step 3. This might cause some of the distances maintained in $D_{A_q \times A_q}$ to change.

- For all $(v, w) \in A_q \times A_q$ for which we have that $Round_{(1+\epsilon)}(D_{A_q \times A_q}[v, w])$ has increased, we increase the weight of shortcut edge (v, w) in G_q (Step 4) to the new $Round_{(1+\epsilon)}(D_{A_q \times A_q}[v, w])$.

- Input the original increase-weight(x, y), as well as all the shortcut-edge weight increases in G_q from the previous step into the h-SSSP algorithms of Step 5. This might cause changes to D_q.

- For $k = q - 1$ down to 0

  - For all pairs $(v, w) \in A_k \times A_{k+1}$ for which we have that $Round_{(1+\epsilon)}(D_{k+1}[v, w])$ or $Round_{(1+\epsilon)}(D_{k+1}[w, v])$ has increased, we increase the weight of corresponding shortcut edge (v, w) or (w, v) in $G_{v,k}$ to the new value of $Round_{(1+\epsilon)}(D_{k+1}[v, w])$ or $Round_{(1+\epsilon)}(D_{k+1}[w, v])$.

  - For each $v \in A_k$, input the original increase-weight(x, y), as well all the shortcut-edge weight increases from the previous step into the h-SSSP algorithm to and from v in $G_{v,k}$. Record the changed distances for each v into the matrix D_k.

5.6.2 Running Time Analysis

As in Section 5.5.2, we have various graphs $G_{v,k}$ whose edges are changing at different rates, but the algorithm is nonetheless decremental from the point of view of each of these graphs. Thus, we simply need to analyze the total update times of all the h-SSSP algorithms, and the time to maintain G_q and all the $G_{v,k}$. Recall the definitions of c, C, and R from Section 5.2. (Technical note: The value of R might vary slightly among the h-SSSP algorithms because of the shortcut weights, but as discussed in section 5.5.2, it never gets so big as to asymptotically affect the running time.)

- To maintain $D_{A_q \times A_q}$ in Step 3, we run the h-SSSP algorithm from $\sqrt{n}$ vertices up to $h = O(\sqrt{n}\log(n))$, which results in total update time $O((\sqrt{n}\log(n))(m\sqrt{n}\log(n)\log(nR)/\epsilon)) = O(mn\log^2(n)\log(nR)/\epsilon)$.

- For all k, we maintain D_k by running the h-SSSP algorithm from $|A_k| = n/2^k$ vertices up to $h = O(2^k \log(n))$. Each graph $G_{v,k}$ has $O(m + |A_{k+1}|) = O(m)$ edges, so the total update time corresponding to any given k is $O((n/2^k)\cdot(2^k\log(n))\cdot(m\log(n)\log(nR)/\epsilon)) = O(mn\log^2(n)\log(nR)/\epsilon)$. There are $O(\log(n))$ values of k, which yields another log factor. Finally, as we discuss at the beginning of the next section, in order for the algorithm to yield the desired $(1 + \epsilon)$ approximation it must internally use $\epsilon' = \epsilon/\log(n)$, which incurs an

additional $\log(n)$ factor. The final running time is thus $O(mn\log^4(n)\log(nR)/\epsilon)$. The same analysis yields a slightly faster running time of $O(mn\log^4(n)\log\log(n)/\epsilon)$ in unweighted graphs because the h-SSSP algorithm is slightly faster in that case (see Theorem 2).

Dependence on Δ*:* The analysis of the overhead per update is almost identical to that of Section 5.5.2. As before, the number of significant updates on any given edge (shortcut edges included) is at most $O(\log_{1+\epsilon}(2nC/c)) = O(\log(nR)/\epsilon)$. The total number of h-SSSP algorithms is: $|A_q| = \sqrt{n}$ running on the original graph G, $|A_q| = \sqrt{n}$ running on G_q, and one more for each graph $G_{v,k}$, for a total of $2\sqrt{n} + \sum_{k=0}^{q}(n/2^k) = O(n)$. Thus, the total number of times that an h-SSSP algorithm processed a significant update on an edge in G (i.e. a non-shortcut edge) is $O(mn\log(nR)/\epsilon)$.

We now bound the number of times an h-SSSP algorithm processed a significant update on a shortcut edge. The graph G_q has $O(n)$ shortcut edges and there are $O(\sqrt{n})$ h-SSSP algorithms running on it, for a total of $O(n^{1.5}\log(nR)/\epsilon)$ significant updates processed. Each graph $G_{v,k}$ has $|A_{k+1}| = n/2^{k+1} = O(n)$ shortcut edges (see Section 5.6.2 for details), each of which is processed by one single h-SSSP algorithm; the one running to and from v in $G_{v,k}$. For any fixed k, there are $|A_k| = O(n/2^k)$ graphs $G_{v,k}$ so the total number of shortcut edges over all the $G_{v,k}$ graphs is $O(n\sum_{k=1}^{q} n/2^k) = O(n^2)$. Since each edge registers only $O(\log(nR)/\epsilon)$ significant updates, the total number of significant shortcut weight increases processed by our algorithm – *i.e.* the number of times that a h-SSSP algorithm must handle a shortcut weight increase – is only $O(n^2\log(nR)/\epsilon)$. All in all, combining shortcut and non-shortcut edges, the total number of times that an h-SSSP algorithm processes a significant update is only $O(mn\log(nR)/\epsilon)$; all other updates are insignificant and thrown away in $O(1)$ time.

5.6.3 Approximation Error Analysis

We will prove that our final distance matrix $D_0[v,w]$ contains a $(1+\epsilon)^{(4+\log(n))}$ approximation to all shortest distances. Using $\epsilon' = \epsilon/(4\log(n))$, we get a $(1+\frac{\epsilon}{4\log(n)})^{(4+\log(n))} \leq (1+\frac{\epsilon}{4\log(n)})^{2\log(n)} \leq (1+\epsilon)$ approximation (see Lemma 15) while only multiplying the running time by $O(\log(n))$. Generally speaking, each layer of the algorithm incurs an $(1+\epsilon)^2$ approximation: one $(1+\epsilon)$ factor comes from the h-SSSPalgorithm, while the other comes from applying the $Round_{(1+\epsilon)}$ function to the shortcut weights of the graphs in that layer.

The graphs G_q and $G_{v,k}$ refer to the graphs created during the main setup (see Section 5.6.1). Recall that $\delta(x, y), h(x, y)$, and $\pi(x, y)$ are changing over time, and refer to the *current* graph.

Lemma 19 *For any pair $(x, y) \in V \times A_q$, the entries $D_q[x, y]$ and $D_q[y, x]$ are $(1+\epsilon)^4$ approximations to $\delta(x, y), \delta(y, x)$ respectively.*

Proof: First note that $D_{A_q \times A_q}$is maintained by running the *h*-SSSP algorithm on the main graph G, so it is $(1 + \epsilon)$ approximate up to $h = 10\sqrt{n}\log(n)$. However, since edge weights in G are themselves only $(1 + \epsilon)$-approximate (because we only register an update to $w(x, y)$ if it increases $Round_{(1+\epsilon)}(w(x, y))$), *h*-SSSP returns $(1 + \epsilon)^2$-approximate distances. Thus, for any pair $(a, b) \in A_q \times A_q$ with $h(a, b) \leq 10\sqrt{n}\log(n)$, there is a 3-shortcut edge (a, b) in G_q; it is 3-shortcut rather than a 2-shortcut because of the extra $(1+\epsilon)$ error that comes from applying the $Round_{(1+\epsilon)}$ function to the shortcut weight.

We now prove that the G_q-3-reduction of $\pi(x, y)$ has at most $10\sqrt{n}\log(n)$ edges. Since we run each *h*-SSSP algorithm to and from its source, the proof for $\pi(y, x)$ is exactly the same. More generally the proof is completely analogous to that of Lemma 17. The goal is to exhibit an $x - y$ path P of hop-length $\leq 10\sqrt{n}\log(n)$ that uses only edges of $\pi(x, y)$ and 3-shortcuts of its subpaths. Recall that $y \in A_q$, and let x_2 be the first vertex in A_q on $\pi(x, y)$. By Lemma 14, x_2 is at most $9\sqrt{n}\log(n)$ vertices away from x, so our path P will just directly take the subpath $\pi(x, x_2)$.

To get from x_2 to y, we prove the following by induction: given any $y' \in A_q$, there is a path from y' to y consisting of at most $\lceil h(x, y)/\sqrt{n} \rceil$ 3-shortcuts of subpaths of $\pi(y', y)$.

- *Base Case:* If $h(y', y) \leq 10\sqrt{n}\log(n)$ then G_q contains a 3-shortcut between them, so we just use that.

- *Induction Step:* We now assume that the claim holds for all vertices $y' \in A_q$ for which $h(y', y) \leq i$, and prove it for the case that $h(y', y) = i + 1$. If $h(y', y) \leq 10\sqrt{n}\log(n)$ we simply use the base case; otherwise, again by Lemma 14, $\pi(y', y)$ contains some vertex $y'' \in A_q$ that is between $\sqrt{n}$ and $10\sqrt{n}\log(n)$ vertices away from y' (this interval of vertices is a shortest path with more than $9\sqrt{n}\log(n)$ vertices, so it must contain a vertex in A_q). Since $h(y', y'') \leq 10\sqrt{n}\log(n)$, G_q contains a 3-shortcut (y', y''); combining this 3-shortcut with the path of $\lceil h(y'', y)/\sqrt{n} \rceil \leq (\lceil h(y', y)/\sqrt{n} \rceil - 1)$ 3-shortcuts from y'' to y guaranteed by the induction hypothesis yields the desired path of 3-shortcuts from y' to y.

Our final path from x to y consists of the at most $9\sqrt{n}\log(n)$ non-shortcut edges from x to x_2, followed by the $\lceil h(x_2, y)/\sqrt{n}\rceil \leq \lceil n/\sqrt{n}\rceil = \lceil\sqrt{n}\rceil$ 3-shortcuts from x_2 to y, yielding less than $10\sqrt{n}\log(n)$ edges in total. By Lemma 13 this implies that the *h*-SSSP algorithm of Step 5, which runs up to hop length $h = 10\sqrt{n}\log(n)$, returns a $(1+\epsilon)(1+\epsilon)^3 = (1+\epsilon)^4$ approximation, as desired. $\square$

Lemma 20 *Given any non-negative integer $k \leq q = \log(n)/2$, and any pair $(u, v) \in A_k \times V$, we have that $D_k[u, v]$ and $D_k[v, u]$ are $(1+\epsilon)^{4+2(q-k)}$ approximations to $\delta(u, v)$, $\delta(v, u)$. In particular, for any pair of vertices $u, v \in V$, $D_0[u, v]$ is a $(1+\epsilon)^{\log(n)+4}$ approximation, as desired.*

Proof: (by induction) We proved the base case of $k = q$ in Lemma 19, so only the induction step is left. We assume the lemma is true for some k and prove that it also holds for $k-1$.

For any $v \in A_{k-1}$, all the shortcut edges in $G_{v,k-1}$ come from D_k, so since the lemma holds for D_k, these must all be $(5+2(q-k))$-shortcuts; the extra $(1+\epsilon)$ factor (from 4 to 5) comes from the application of the $Round_{(1+\epsilon)}$ function to the shortcut weights in $G_{v,k-1}$. We now show that for any pair $(u, v) \in V \times A_{k-1}$, the $G_{v,k-1}$-$(5+2(q-k))$-reduction of $\pi(u, v)$ has hop-length $\leq 10\log(n)2^k$. Let u_2 be the first vertex in A_k on $\pi(u, v)$ (if u_2 does not exist then by Lemma 14 $h(u, v) \leq 9\log(n)2^k$, so we are done). We know from Lemma 14 that the subpath $\pi(u, u_2)$ of $\pi(u, v)$ contains at most $9\log(n)2^k$ edges. Moreover, because of how we constructed $G_{v,k-1}$, there must be a $(5+2(q-k))$-shortcut (u_2, v). We have thus exhibited a $u-v$ path with $\leq 9\log(n)2^k + 1 \leq 10\log(n)2^k$ edges, as desired. It is not hard to see that by symmetry, the same holds for the reverse direction: for any $(u, v) \in V \times A_{k-1}$ the $G_{v,k-1}$-$(5+2(q-k))$-reduction of $\pi(v, u)$ has hop-length $\leq 10\log(n)2^k$.

Thus, by Corollary 13, the *h*-SSSP algorithm to and from v on $G_{v,k-1}$ up to $h = 10\log(n)2^k$ incurs an additional $(1+\epsilon)$ approximation and returns a $(1+\epsilon)^{6+2(q-k)} = (1+\epsilon)^{4+2(q-(k-1))}$ approximation to both $\delta(u, v)$ and $\delta(v, u)$. Our argument holds for all pairs $(u, v) \in V \times A_{k-1}$, so we are done. $\square$

5.7 The *h*-SSSP Algorithm

We now present the *h*-SSSP algorithm for decrementally maintaining an approximate shortest path tree up to hop-length h. This algorithm was not new to the paper under discussion [Bernstein, 2013],

and was used as a subroutine in Bernstein's FOCS 2009 paper [Bernstein, 2009]. Recall the main theorem we are trying to prove.

Theorem 3 *[Bernstein, 2009] Given a source s and a hop distance h, we can decrementally maintain distances $\delta'(s, v)$ to every vertex v such that we always have $\delta(s, v) \leq \delta'(s, v) \leq (1 + \epsilon)\delta^h(s, v)$. The total update time over all deletions and weight-increases is $O(mh \log(n) \log(nR)/\epsilon + \Delta)$ in weighted graphs, and $O(mh \log(n) \log\log(n))$ in unweighted ones.*

Recall from Theorem 1 that the main idea behind King's $O(md)$ algorithm was to only explore the edges of a vertex v when the distance to v from s changed. The basic idea of our $(1 + \epsilon)$ approximation is to only explore the edges of v when $\delta(s, v)$ changes by a significant amount. The *h*-SSSP algorithm is actually broken up into many smaller algorithms, each of which handles different ranges of $\delta^h(s, v)$.

Definition 18 *Given a source vertex s, a hop-length h, and an integer k, we say that algorithm A_k maintains* h-*SSSP*$_k$ *if it decrementally maintains distances $\delta'_k(s, v)$ with the following properties:*

- *If $2^k \leq \delta^h(s, v) \leq 2^{k+1}$ then $\delta(s, v) \leq \delta'_k(s, v) \leq (1 + \epsilon)\delta^h(s, v)$.*
- *Otherwise, our only guarantee is that that $\delta(s, v) \leq \delta'_k(s, v)$.*

Lemma 21 *Assuming $0 < \epsilon < 1$, we can maintain any* h-*SSSP*$_k$ *in total update time $O(mh/\epsilon + \Delta)$.*

Proof: (of Lemma 21) Recall that we are here only concerned with approximating distances $\delta^h(s, v)$ for which $2^k \leq \delta^h(s, v) \leq 2^{k+1}$. For the rest of this proof, let $\alpha = \frac{\epsilon 2^k}{h}$. We start by scaling the edge-weights of graph G to obtain a new graph G_k in the following way.

- Delete all edges of weight $> 2^{k+1}$ from G
- Round all remaining edge weights up to the nearest integer multiple of α
- Divide all edge weights by α

Note that the scaled weights in G_k are positive and integral. Our algorithm maintains a shortest path tree from s in G_k by simply running King's $O(md)$ decremental SSSP algorithm (see Section 5.3) up to distance $d = \lceil 4h/\epsilon \rceil$. More precisely, if an update deletes (u, v) in the original graph we

simply delete (u, v) in the scaled graph; if the update is increase-weight (u, v), we first scale the new weight according to the three steps above, and then change the weight of $w(u, v)$ to this new scaled weight. We then output $\delta'_k(s, v) = \alpha \cdot \delta_{G_k}(s, v)$. By Lemma 1, the total update time of King's algorithm is just $O(md) = O(mh/\epsilon)$, as desired. The final $O(\Delta)$ term arises from weight increases in G that do not change the scaled weights in G_k (*i.e.*, the old weight and the new weight scale up to the same nearest multiple of α), and so are discarded in $O(1)$ time. We now do an approximation analysis.

Let G^*_k be the graph G_k before dividing the edge weights by α (but after scaling up to a multiple of α), and note that since G_k and G^*_k are the same up to a scaling factor, our output is precisely $\delta'_k(s, v) = \alpha \cdot \delta_{G_k}(s, v) = \delta_{G^*_k}(s, v)$. All edge weights in G^*_k are greater than those in G, so it is clear that $\delta'_k(s, v) = \delta_{G^*_k}(s, v) \geq \delta(s, v)$. We now need to show that if $2^k \leq \delta^h(s, v) \leq 2^{k+1}$, then $\delta_{G^*_k}(s, v) \leq (1 + \epsilon)\delta^h(s, v)$. To see this, let us examine how the weight changes $G \to G^*_k$ affect $\pi^h(s, v)$. Since $\delta^h(s, v) \leq 2^{k+1}$, $\pi^h(s, v)$ does not contain any edges of weight $> 2^{k+1}$, so the first set of changes does not affect it at all. The second set of changes adds up to α weight to every edge on $\pi^h(s, v)$, so the weight of the path in G^*_k is at most $\delta^h(s, v) + h\alpha = \delta^h(s, v) + \epsilon 2^k \leq (1 + \epsilon)\delta^h(s, v)$ (the last inequality follows from $2^k \leq \delta^h(s, v)$). Thus, we have exhibited an $s - v$ path in G^*_k of weight $\leq (1 + \epsilon)\delta^h(s, v)$, so certainly the *shortest* $s - v$ path in G^*_k will have weight $\leq (1 + \epsilon)\delta^h(s, v)$, as desired. Finally, note that the weight of this path in G_k is at most $(1 + \epsilon)\delta^h(s, v)/\alpha \leq \frac{2 \cdot 2^{k+1}}{\epsilon 2^k / h} = 4h/\epsilon$, so running King's algorithm up to $d = \lceil 4h/\epsilon \rceil$ in G_k will in fact find this path. □

We now show how to obtain an algorithm for *h*-SSSP by simply combining *h*-SSSP$_k$ algorithms for different values of k. The most natural way to do this, however, achieves a slightly worse dependence on $O(\Delta)$ than the one promised in Theorem 3: $O(\Delta \log(nR) \log\log(nR))$. For the sake of intuition, we show in this section a simple method for achieving this worse update time, and then show in the next section how to reduce the dependence on Δ to $O(\Delta)$. Recall the definitions of c, C, and R from Section 5.2.

Lemma 22 *If for any k we could maintain* h-*SSSP$_k$ in total update time T, then we would have an algorithm for* h-*SSSP with total update time $O(T \log(nR) \log\log(nR))$.*

Proof: All we do is maintain *h*-SSSP$_k$ for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$. The crux is that if we then set $\delta'(s, v) = \min\{\delta'_k(s, v)\}$ we have the desired property $\delta(s, v) \leq \delta'(s, v) \leq (1 + \epsilon)\delta^h(s, v)$.

To see this, note that $\delta(s,v)$ can never be smaller than c or larger than nC, so in particular, there is some k between $\lfloor \log(c) \rfloor$ and $\lceil \log(nC) \rceil$ for which $2^k \leq \delta^h(s,v) \leq 2^{k+1}$. For this value of k, we know that h-SSSP$_k$ outputs $\delta'_k(s,v) \leq (1+\epsilon)\delta^h(s,v)$, so it is certainly true that $\delta'(s,v) = \min_k\{\delta'_k(s,v)\} \leq (1+\epsilon)\delta^h(s,v)$. That $\delta'(s,v) \geq \delta(s,v)$ follows from the fact that every $\delta'_k(s,v)$ is $\geq \delta(s,v)$.

Thus, for each vertex v, our algorithm maintains a min-heap of all $\delta'_k(s,v)$ for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$. A query operation for $\delta'(s,v)$ simply returns the minimum of this heap in $O(1)$ time. An update operation on edge (x,y) is inputted into every h-SSSP$_k$, and whenever some $\delta'_k(s,v)$ changes we update the min-heap for $\delta'(s,v)$ in $O(\log\log(nR))$ time. We maintain $O(\log(nR))$ different h-SSSP$_k$, so the total time spent processing updates in the h-SSSP$_k$ is $O(T\log(nR))$. This is also the bound on how often some $\delta'_k(s,v)$ can change, so the total time updating the min-heap is $O(T\log(nR)\log\log(nR))$. Together the two add to $O(T\log(nR)\log\log(nR))$, as desired. □

Corollary 6 *We can maintain* h*-SSSP in total time* $O(mh\log(nR)\log\log(nR)/\epsilon + \Delta\log(nR)\log\log(nR))$. *This follows directly from the two preceding lemmas. Note that in unweighted graphs* $R = 1$ *and* $\Delta \leq m$*, so the running time is* $O(mh\log(n)\log\log(n)/\epsilon)$*, as promised in Theorems 2 and 3.*

5.7.1 Limiting the dependence on Delta to O(Delta)

In this section, we improve the dependence on Δ in Corollary 6 to $O(\Delta)$, thus achieving the total update time for unweighted graphs promised in Theorem 3. Note that although the h-SSSP algorithm itself was used in Bernstein's FOCS 2009 paper [Bernstein, 2009], in that paper the implicit dependence on Δ was $O(\Delta\log(nR)\log\log(nR))$. *Thus, the reduction to an* $O(\Delta)$ *dependence on* Δ *is in fact new to the paper under discussion [Bernstein, 2013].*

That being said, the improvement in this subsection is extremely technical and not particularly interesting from a conceptual perspective. Moreover, it is not all that important: using the simple h-SSSP algorithm presented in Lemma 22 and Corollary 6 would yield a decremental APSP algorithm with total update time $\widetilde{O}(mn\log R/\epsilon) + O(\Delta\log(nR)\log\log(nR))$. The $O(\Delta\log(nR)\log\log(nR))$ term is very unlikely to affect the asymptotic running time, especially as if it ever came to dominate that would imply that we were achieving an amortized update time of $O(\log(nR)\log\log(nR))$, which is already very good. The reader would thus not lose much in simply skipping the current section.

We now continue with the improvement to h-SSSP. Say that we are running the h-SSSP algorithm up to hop-length h. Let us focus on processing a particular update increase-weight(u, v), and let $w_{\text{old}}(u, v)$ and $w_{\text{new}}(u, v)$ respectively correspond to the weights of (u, v) before and after update (a deletion can be modeled by setting $w_{\text{new}}(u, v) = \infty$). Since we are in a decremental setting, we know that $w_{\text{old}}(u, v) < w_{\text{new}}(u, v)$. The naive way for h-SSSP to process increase-weight(u, v) is to process this update in each of the h-SSSP$_k$ algorithms: there are $O(\log(nR))$ different values for k, so this would require a minimum of $O(\log(nR))$ time. But note that there is no reason to process this update in some particular h-SSSP$_k$ if we know that increase-weight(u, v) has no chance of affecting $\delta'_k(s, x)$ for any vertex x. Thus, our basic approach is to only update increase-weight(u, v) in those h-SSSP$_k$ for which it might be relevant.

Before proceeding, let us carefully pinpoint the different steps taken by the h-SSSP algorithm, so that we can analyze the parts separately. The algorithm for handling an update increase-weight(u,v) can be thought of as consisting of the three operations below:

Running Time Breakdown:

1. Figure out for which h-SSSP$_k$ the update might be relevant, and register the update in those h-SSSP$_k$ only.

2. Process the update in the chosen h-SSSP$_k$, thus potentially changing various $\delta'_k(s, v)$. This is where the "real work" occurs.

3. Now that some of the $\delta'_k(s, v)$ have changed, we must update $\delta'(s, v) = \min_k\{\delta'_k(s, v)\}$.

We have already analyzed the total time spent in Step 2 over all updates: each h-SSSP$_k$ spends a total of $O(mh/\epsilon)$ time processing its updates, so since there are $O(\log(nR))$ possible values of k, among all h-SSSP$_k$ we have total update time $O(mh\log(nR)/\epsilon)$, as desired. For Steps 1 and 3 to be efficient however, we must modify the h-SSSP algorithm.

We start with Step 3. First let us bound how often the $\delta'_k(s, v)$ might change. Recall that h-SSSP$_k$ runs on a scaled graph G_k, where it only stores distances up to distance $\lceil 4h/\epsilon \rceil$. Thus, since distances only increase, it is clear that for any particular v, $\delta'_k(s, v)$ can change at most $\lceil 4h/\epsilon \rceil$ times. Summing over all vertices v, and all the h-SSSP$_k$, we see that in total over all

updates there are $O(nh\log(R)/\epsilon)$ changes to the $\delta'_k(s,v)$. The total time spent in Step 3 is thus $O([nh\log(nR)/\epsilon]\cdot$ [the time to update $\delta'(s,v) = \min_k\{\delta'_k(s,v)\}$ when some $\delta'_k(s,v)$ changes)].

The second term of course depends on our data structure for updating $\delta'(s,v) = \min_k\{\delta'_k(s,v)\}$. The most natural option would be, for any particular v, to store all the $\delta'_k(s,v)$ in a min-heap. This heap would have $O(\log(nR))$ elements, and so update time $O(\log\log(nR))$. This is not quite good enough, as it would imply a total time of $O(nh\log(nR)\log\log(nR)/\epsilon)$ spent in Step 3, which is not strictly contained in our desired bound of $O(mh\log(n)\log(nR)/\epsilon)$. We now present a different data structure.

Lemma 23 *Given any vertex v, and assuming that $(1/\epsilon)$ is at most polynomial in n, we can build a data structure on the $\delta'_k(s,v)$ that returns $\delta'(s,v) = \min_k\{\delta'_k(s,v)\}$ in $O(1)$ time and processes an increase to some $\delta'_k(s,v)$ in $O(\log(n))$ time.*

Proof: The data structure is based on the following observation:

Observation: Let k' be some index for which $\delta'_{k'}(s,v) \neq \infty$. Then, for any index $k^* > k' + 2 + \log(h/\epsilon)$ we always have that $\delta'(s,v) \neq \delta'_{k^*}(s,v)$.

This observation relies on the details of the h-SSSP$_k$ algorithm presented in the proof of Lemma 21. For our index k', h-SSSP$_{k'}$ only runs up to distance $\lceil 4h/\epsilon \rceil$ on the scaled graph $G_{k'}$, so any path it finds will have length at most $\lceil 4h/\epsilon \rceil$ in $G_{k'}$. It is clear from how h-SSSP$_k$ performs the scaling that any edge weight in G'_k is no larger than the corresponding edge weight in G divided by $\epsilon 2^{k'}/h$; thus, the unscaled length in G of any path found by h-SSSP$_k$ is at most $(\lceil 4h/\epsilon \rceil)\cdot(\epsilon 2^{k'}/h) \leq 2^{k'+2} + 2^{k'} < 2^{k'+3}$, so $\delta'_{k'}(s,v) \neq \infty$ implies that $\delta'(s,v) \leq \delta'_{k'}(s,v) < 2^{k'+3}$. But now, looking at k^*, we see that we always have $\delta'_{k^*}(s,v) \geq \epsilon 2^{k^*}/h$ because we scale each edge weight up to the nearest multiple of $\epsilon 2^{k^*}/h$. Thus, if $k^* > k' + 2 + \log(h/\epsilon)$, then since k^* is an integral index we have $k^* \geq k' + 3 + \log(h/\epsilon)$, so $\delta'_{k^*}(s,v) \geq \epsilon 2^{k^*}/h \geq 2^{k'+3} > \delta'(s,v)$, so $\delta'(s,v) \neq \delta'_{k^*}(s,v)$, as desired.

Data Structure: The data structure is very simple: we keep the $\delta'_k(s,v)$ in a linked list, sorted in increasing order of k (*not* in increasing order of $\delta'_k(s,v)$). We also maintain a pointer to the minimum $\delta'_k(s,v)$ in the list. We can return $\delta'(s,v)$ in $O(1)$ time by following the min-value pointer, so we now focus on updates to the $\delta'_k(s,v)$. We will always throw away any $\delta'_k(s,v)$

for which $\delta'_k(s,v) = \infty$ so the head of the list will be the first entry for which $\delta'_k(s,v) \neq \infty$. Thus, by the observation above we know that $\delta'(s,v) = \min_k\{\delta'_k(s,v)\}$ will always be within $k' + 2 + \log(h/\epsilon) = O(\log(n))$ entries from the head (we are assuming that $(1/\epsilon)$ is at most polynomial in n). After some increase-weight(x,y), some of the $\delta'_k(s,v)$ values may be increased. To process these increases, we first update the $\delta'_k(s,v)$ in our list (we can trivially maintain pointers that will allow us to do this). We then go through the list, starting from the head, and delete any $\delta'_k(s,v)$ that has come to equal ∞; since distances only increase, once $\delta'_k(s,v) = \infty$ for some index k, it will continue to be ∞ in the future, so we can safely throw it away. We stop when we reach the first $\delta'_k(s,v) \neq \infty$. By the observation above we can now find the minimum by comparing this first entry and the $k' + 2 + \log(h/\epsilon) = O(\log(n))$ entries that come after it, and then update our min-value pointer. It is clear that the update time per $\delta'_k(s,v)$-increase is just $O(\log(n) + [$ number of $\delta'_k(s,v)$ deleted$])$. For a fixed v, the second term amounts to a total of O($\log(nR)$) over all updates, since that the number of possibilities for k. Summed over all vertices v, this yields an extra $O(n\log(nR))$ total time for h-SSSP, which is well within our $O(mh\log(n)\log(nR)/\epsilon)$ time bound. We can thus maintain $\delta'(s,v)$ in time $O(\log(n))$ time per change to $\delta'_k(s,v)$ □

The total time spent in Step 3 (see running time breakdown above) is thus $O([nh\log(nR)/\epsilon] \cdot$ [the time to update $\delta'(s,v)$]$) = O([nh\log(nR)/\epsilon]\cdot[\log(n)]) = O(nh\log(n)\log(nR)/\epsilon)$, which is within our desired $O(mh\log(n)\log(nR)/\epsilon)$ time bound.

All we have left is to analyze the total time spent in Step 1 of the running time breakdown above. This will take some work. Let us focus on h-SSSP$_k$ for some particular value of k. Recall that given update increase-weight(u,v), h-SSSP$_k$ starts by scaling the weight of (u,v) in G_k. The first two steps are as follows:

- If $w_{\text{new}}(u,v) > 2^{k+1}$, h-SSSP$_k$ deletes it from G_k.
- Scale $w_{\text{new}}(u,v)$ up to the nearest multiple of $\epsilon 2^k/h$.

Thus, it is easy to see that if $w_{\text{old}}(u,v) > 2^{k+1}$ or if $w_{\text{old}}(u,v)$ and $w_{\text{new}}(u,v)$ both scale up to the same multiple of $\epsilon 2^k/h$, then there is no need to process increase-weight(u,v) in h-SSSP$_k$ as it will not affect G_k in any way. This motivates the following definitions. Recall that h is the hop-length

to which we are running the h-SSSP algorithm in question.

Definition 19 *Let $\alpha = \epsilon/h$. Given an integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$, we say that a positive real number ζ is k-*marked *if both of the following properties hold:*

1. *ζ is an integer multiple of $2^k\alpha = \epsilon 2^k/h$*
2. $\zeta < 2^{k+1}$

We say that a number is marked *if it is k-marked for at least one value of k.*

Lemma 24 *Say that we are given an update increase-weight(u, v): $w_{old}(u, v) \rightarrow w_{new}(u, v)$ (an edge deletion just increases the weight to ∞). This update affects an* h-*SSSP$_k$ algorithm only if there is a k-marked number in the half-open interval $[w_{old}(u, v), w_{new}(u, v))$. We will refer to such an update as* crossing *the k-marked number.*

Proof: This lemma stems directly from our previous discussion. The only way for increase-weight(u, v) to change a weight in G_k is if $w_{\text{old}}(u, v) \leq 2^{k+1}$ and if $w_{\text{old}}(u, v)$ and $w_{\text{new}}(u, v)$ scale up to different multiples of $\epsilon 2^k/h = \alpha 2^k$. This is equivalent to the requirement that there is a k-marked number in the half-open interval $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$ □

Lemma 25 *For any integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$ there are $2h/\epsilon = 2/\alpha$* k-*marked numbers, for a total of $2h \log(nR)/\epsilon$ marked numbers. (Recall that we are focusing on a particular* h-*SSSP algorithm running up to hop-length h, so h if fixed.)*

Proof: For any integer $k \in [\lfloor \log(c) \rfloor, \lceil \log(nC) \rceil]$, the k-marked numbers are all the positive multiples of $\alpha 2^k$ that are $\leq 2^{k+1}$. It is easy to see that there are exactly $2^{k+1}/(\alpha 2^k) = 2/\alpha = 2h/\epsilon$ of these. There are $\log(nC) - \log(c) = \log(nR)$ different possible values for k, yielding a total of $2h \log(nR)/\epsilon$ marked numbers. □

Lemma 26 *Given a marked number ζ, there are $O(\log(n))$ values of k for which ζ is* k-*marked, and we can find all of them in $O(\log(n))$ time.*

Proof: If ζ is marked, it must be k-marked for at least one k, so there must be some k such that ζ is an integer multiple of $2^k\alpha$ that is less than 2^{k+1}. In particular, for that value of k we must have

$2^k\alpha \le \zeta \le 2^{k+1}$. Taking logs yields: $k + \log(\alpha) \le \log(\zeta)$ and $\log(\zeta) \le k+1$; the first inequality then yields $k \le \log(\zeta) - \log(\alpha)$, while the second yields $\log(\zeta) - 1 \le k$. All in all we thus have:

$$\log(\zeta) - 1 \le k \le \log(\zeta) - \log(\alpha)$$

so we only have to consider integers k in interval $[\log(\zeta) - 1, \log(\zeta) - \log(\alpha)]$ (note that $\log(\alpha) = \log(\epsilon/h)$ is negative). This interval contains at most $1 - \log(\alpha) \le 1 + \log(h/\epsilon) = O(\log(n))$ integers k, and for each such k we can check in $O(1)$ time if ζ is an integer multiple of $2^k\alpha$. □

We can now give an intuition for our algorithm. Recall from Lemma 24 that an update increase-weight(u, v) only needs to be processed by some h-SSSP$_k$ if it crosses a k-marked number. By Lemma 26 each marked number is k-marked for only $O(\log(n))$ values of k. Thus, an update increase-weight(u, v) must be processed by $O(\log(n))$ h-SSSP$_k$ algorithms for every marked number that it crosses. But since weights only increase, they go through each marked number exactly once, so by Lemma 25, all the updates on a single edge (u, v) go through a total of $O(h \log(nR)/\epsilon)$ marked numbers. Thus, over all weight increases on a single edge (u, v), the total number of updates to the h-SSSP$_k$ algorithms is $O(\log(n)) \cdot O(h \log(nR)/\epsilon) = O(h \log(n) \log(nR)/\epsilon)$. Thus, over all edges, the total number of times that an update affects some h-SSSP$_k$ and must be further processed is $O(mh \log(n) \log(nR)/\epsilon)$; since each h-SSSP$_k$ algorithm only needs an additional $O(1)$ time per update (the $O(\Delta)$ term in Lemma 21), this does not exceed our overall $O(mh \log(n) \log(nR)/\epsilon)$ time bound for h-SSSP. We now have to show that it only takes us $O(1)$ per update to determine which h-SSSP$_k$ an update should be processed in.

Lemma 27 *Given an update increase-weight(u, v): $w_{old}(u, v) \to w_{new}(u, v)$ we can find the* smallest *marked number of $[w_{old}(u, v), w_{new}(u, v))$ (if it exists) in $O(1)$ time.*

Proof: By the definition of k-marked (Definition 19) it is clear that if there is to be a k-marked number in $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$ (for some k), then we must have that $w_{\text{old}}(u, v) \le 2^{k+1}$. Thus, $k_0 = \lceil \log(w_{\text{old}}(u, v)) \rceil$ is the very smallest value of k for which $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$ might contain a k-marked number. Moreover, for any $k' > k_0$, an integer multiple of $2^{k'}\alpha$ is obviously an integer multiple of $2^k\alpha$. Thus, the smallest marked number of $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$ is simply $w_{\text{old}}(u, v)$ rounded up to the nearest multiple of $2^{k_0}\alpha$, which we can find in $O(1)$ time (if this nearest multiple of $w_{\text{old}}(u, v)$ is $\ge w_{\text{new}}(u, v)$ then there is no k-marked number in interval $[w_{\text{old}}(u, v), w_{\text{new}}(u, v))$). □

The Improved h-SSSP Algorithm:

We now present our algorithm for processing an update increase-weight(u,v): $w_{\text{old}}(u,v) \to w_{\text{new}}(u,v)$ in such a way as to only update the relevant h-SSSP$_k$. Note that the actual processing of an update in a particular h-SSSP$_k$ is no different than before, so we simply follow the algorithm presented in Lemma 21. The set S will end up containing all indices k for which increase-weight(u,v) affects h-SSSP$_k$.

1. While(True)

   (a) Use Lemma 27 to find the smallest marked number ζ in $[w_{\text{old}}(u,v), w_{\text{new}}(u,v))$. If no marked number exists in this interval, terminate loop, go to Step 2.

   (b) Use Lemma 26 to find all k for which ζ is k-marked, and add them to S.

   (c) Start over from Step 1, but this time process increase-weight(u,v): $\zeta \to w_{\text{new}}(u,v)$.

2. For all $k \in S$, process the update increase-weight(u,v): $w_{\text{old}}(u,v) \to w_{\text{new}}(u,v)$ in h-SSSP$_k$.

3. Update the $\delta'(s,x) = \min\{\delta'_k(s,x)\}$ for all affected vertices x.

Analysis: Correctness follows directly from Lemma 24. Let us examine the time to process a particular update increase-weight(u,v). Recall that everything written in this section is about determining which h-SSSP$_k$ are affected by the update – once we decide to process the update in a particular h-SSSP$_k$, it is processed in exactly the same way as in Section 5.7. Thus, as in the running time breakdown above, the overall running time of our algorithm consists of three components:

1. The time spent determining which updates increase-weight(u,v) should be processed by which h-SSSP$_k$– *i.e.* the loop of Step 1 above.

2. The time spent actually processing the updates – *i.e.* Step 2 above.

3. The time spent updating $\delta'(s,v) = \min\{\delta'_k(s,v)\}$.

We showed at the beginning of this section that Steps 2 and 3 are both within our $\widetilde{O}(mh \log R/\epsilon)$ time bound (see "running time breakdown" above). We now bound the time spent on the while loop of Step 1. Running the algorithm of Lemma 27 in (1a) to find a marked number only takes O(1) time, but then running the algorithm of Lemma 26 in (1b) takes $O(\log(n))$ time, even though in the end we may discover that the marked number ζ is in fact only k-marked for a single value of k. Thus, the running time of the loop for a single update increase-weight(u, v) is $O(1)$, plus an additional $\log(n)$ for every marked number crossed by increase-weight(u,v). Because we are in a decremental setting, all the weight increases of any particular edge (u, v) only go through each marked number once, so by Lemma 25, all of the different edge updates combined cross a *total* of only $O(mh \log(nR)/\epsilon)$ marked numbers, so the total spent in the while loop over *all* updates to the h-SSSP algorithm is $O(mh \log(n) \log(nR)/\epsilon + \Delta)$, as desired. We have thus proved Theorem 2.

5.8 Final Touches

5.8.1 Removing the Assumption that We Know R in Advance

Recall from Section 5.2 that we define c to be the lightest edge weight to appear in the graph at any point in the update sequence, and C to be the heaviest such edge weight. We define $R = C/c$. Since edge weights only increase, c is just the lightest edge weight in the original graph, before any updates occur, so we know it from the start. C, however, can keep increasing, and as presented, our algorithm requires an a-priori upper bound on C in order to run the right h-SSSP$_k$ algorithms: each h-SSSP algorithm consists of running an h-SSSP$_k$ algorithm for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$ (see Definition 18 and Lemma 22 in Section 5.7 for a description of h-SSSP$_k$). We show in this section that we do not actually need to know C ahead of time, and can instead just continually update our current bound on C.

At any point in the update sequence, define C^* to be the largest edge weight *seen so far*, and note that $C^* \leq C$. Recall from Section 5.7 that we create algorithm h-SSSP$_k$ to handle distances between 2^k and 2^{k+1}, and that h-SSSP returns $\delta'(s, v) = \min\{\delta'_k(s, v)\}$, where $\delta'_k(s, v)$ is the (s, v)-distance returned by h-SSSP$_k$. But all distances in the current graph are less than nC^*, so we have no need for h-SSSP$_k$ as long as $2^k > nC^*$ – that is, for $k > \lceil \log(nC^*) \rceil$, we can think of $\delta'_k(s, v) = \infty$. Thus, instead of immediately creating h-SSSP$_k$ for all $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$,

we just create them for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC^*) \rceil$. Then, as C^* increases with updates, we start running h-SSSP$_k$ as soon as k becomes $\leq \lceil \log(nC^*) \rceil$, and we add $\delta'_k(s, v)$ to the heap for $\delta'(s, v)$.

It is easy to see that the running time of this new method is no worse than if we knew C in advance and set up all the h-SSSP$_k$ from the beginning (for $\lfloor \log(c) \rfloor \leq k \leq \lceil \log(nC) \rceil$). It is in fact slightly faster, as we avoid processing updates that occur while $k > \lceil \log(nC^*) \rceil$.

5.8.2 The Incremental Setting

As presented, our algorithm works in the decremental setting, where we have only deletions and weight increases. However, like many other decremental algorithms, our algorithm can be made to run in the incremental setting with only the smallest of modifications. That is, it can process either a sequence of deletions/weight-increases or a sequence of insertions/weight-decreases, though certainly not a sequence of both.

Most of the description of our algorithm deals with the various graphs and shortcut edges that we construct. Yet when it comes to dynamically processing the updates, all the work is done by the various h-SSSP algorithms running on these different graphs. The h-SSSP algorithm is in turn composed of h-SSSP$_k$ algorithms. So in the end, our algorithm is merely a large collection of different h-SSSP$_k$ algorithms. But the h-SSSP$_k$ algorithm simply runs King's algorithm for maintaining a shortest path tree, which by Lemma 1 runs equally well in the incremental setting.

Thus, our algorithm can be made to run in the incremental setting by simply switching all of the constituent h-SSSP$_k$ algorithms to run in the incremental setting. Everything else remains unchanged: the various graphs $G_{v,k}$, the shortcut edges, the approximation analysis, *etc.*. There are only two minor points worth noting:

- Recall that our algorithm constructs many different graphs, and so an update in G can proliferate into multiple updates on multiple graphs. We argued that we could nonetheless run h-SSSP on these different graphs because although the update sequence differs from the perspective of each graph, it is always decremental: shortcut weights correspond to distances in the original graph, so since distances only increase, shortcut weights only increase. A symmetric claim is true of the incremental setting: since distances in the original graph only decrease, shortcut weights also only decrease, so the update sequence is incremental from the perspective of all the different graphs.

- In Section 5.8.1 we argued that while c remains fixed, C increases over time, so our decremental algorithm kept an upper bound on C in order to have an estimate on R. In the incremental setting, it is C that is fixed and c that changes, so we instead keep a lower bound on c.

5.8.3 A Fully Dynamic Algorithm

There is a standard technique for transforming any decremental algorithm for all pairs shortest paths or directed transitive closure into a fully dynamic algorithm with query-update trade offs. The fastest update times we know how to achieve in the fully dynamic setting often stem from this technique, though at the often unacceptable cost of a polynomial query time. The technique was first introduced in a paper by Henzinger and King [Henzinger and King, 1995] on dynamic transitive closure, and has since been used in several papers on dynamic shortest paths and dynamic transitive closure (see *e.g* [Roditty and Zwick, 2008b; Roditty and Zwick, 2004a; King, 1999]). Our application of the technique is completely identical to the one used in the cited papers, but as far as we know, none of those papers presented the result in quite sufficient enough generality for it to apply directly to our case. We thus reconstruct the technique from scratch, stating it in the most general terms possible (note that transitive closure is a special case of α-approximate all-pairs shortest paths). Recall that $\pi(x, y)$ is the shortest path from x to y in the current version of the graph and that $\delta(x, y)$ is the length of this path.

Theorem 10 ([Henzinger and King, 1995]) *Given a decremental algorithm D for α-approximate all pairs shortest paths with* total *update time Λ over all updates and query time $O(q)$, for any positive integer t we can construct a fully dynamic algorithm for α-approximate APSP with* amortized *update time $O(\Lambda/t+(m+n\log(n))t)$ and query time $O(t+q)$. The fully dynamic algorithm admits the following batch updates in the same $O(\Lambda/t + (m + n\log(n))t)$ amortized time per update:*

- Batch Delete: *Delete or increase the weight of an arbitrary set of edges E'*
- Centered Batch Insert: *Insert or decrease the weight of a group of edges E_v that are all incident to some vertex v*

Corollary 7 *Given our (randomized) $\widetilde{O}(mn\log R/\epsilon)$ decremental algorithm for $(1+\epsilon)$-approximate APSP in directed weighted graphs, we can build a fully dynamic (randomized) algorithm with amor-*

tized update time $\widetilde{O}(\frac{mn \log R}{\epsilon t})$ and query time $O(t)$ for any $1 \leq t \leq \sqrt{n}$. Previously such a result was known only for undirected unweighted graphs.

Proof: (of theorem) We use the decremental algorithm D as a black box to build the desired fully dynamic algorithm. Let $D(u, v)$ be the α-approximate shortest distance from u to v returned by algorithm D.

Initialization:

- Initialize the decremental algorithm D on the starting graph G. This computes α-approximate APSP in G, and maintains this information over all deletions to come.
- Create an empty list I which will contain the vertices affected by insertions.
- Create a counter C for the number of updates so far. Start with $C = 0$.

Batch-Insertion(E_v):

- If $C > t$, restart the entire algorithm. That is, set $C = 0$, delete all elements from I, and reinitialize the decremental algorithm D on the current version of the graph. Else, set $C = C + 1$ and continue.
- Add v to I.
- For every vertex $x \in I$ use Dijkstra's algorithm to compute single source shortest distances to and from x.

Batch-Delete(E'):

- If $C > t$, restart the entire algorithm. That is, set $C = 0$, delete all elements from I, and reinitialize the decremental algorithm D on the current version of the graph. Else, set $C = C + 1$ and continue.
- Process all the deletions in the decremental algorithm D. This of course changes the various $D(u, v)$.
- For every vertex $x \in I$ use Dijkstra's algorithm to compute single source shortest distances to and from x.

Query(u,v):

- Let $I(u,v) = \min_{x \in I}(\delta(u,x) + \delta(x,v))$
- Return $\delta'(u,v) = \min\{I(u,v), D(u,v)\}$

Correctness Proof: Note that when we compute $I(u,v) = \min_{x \in I}(\delta(u,x) + \delta(x,v))$ we know both $\delta(u,x)$ and $\delta(x,v)$ because after each update we compute shortest paths to and from each vertex in I. Thus, $I(u,v)$ contains the length of the shortest path from u to v that goes through one of the vertices in I. If $\pi(u,v)$ contains a vertex from I, our query algorithm outputs $\delta'(u,v) = I(u,v) = \delta(u,v)$. If $\pi(u,v)$ does not use any vertex in I, then the entire path $\pi(u,v)$ exists in the graph G_D, which is the graph G subjected to all the deletions in our update sequence and none of the insertions. Since the decremental algorithm D is running precisely on G_D, $D(u,v)$ is guaranteed to return an α-approximation to $\delta(u,v)$. In either case, we have that $\delta'(u,v) = \min\{I(u,v), D(u,v)\}$ is a α-approximation to $\delta(u,v)$, as desired.

Running Time Analysis: The fully dynamic algorithm runs for exactly t updates before restarting. Since the *total* update time of the decremental algorithm D is Λ, the amortized time for a single update is thus $O(\Lambda/t)$. Each update inserts exactly one vertex into I so we always have $|I| \leq t$. Each update also requires us to run Dijkstra's algorithm to and from each vertex in I, which takes time $O((m + n\log(n))|I|) = O((m + n\log(n))t)$. The amortized update time is thus $O(\Lambda/t + (m + n\log(n))t)$, as desired.

Each query requires us to compute $\delta(u,x) + \delta(x,v)$ for every $x \in I$. Each such value can be compute in $O(1)$ since we already know $\delta(u,x)$ and $\delta(x,v)$. The query algorithm also spends $O(q)$ time querying $D(u,v)$. The query time is thus $O(|I| + q) = O(t + q)$, as desired.

□

5.9 Conclusions

For the problem of partially dynamic all-pairs shortest paths (APSP), a natural goal is total update time $\widetilde{O}(mn)$. This matches the time it takes to compute even a single instance of APSP, and a conditional lower bound of [Henzinger et al., 2015c] shows that total update time $O(mn)$ is optimal even for unweighted undirected graphs and a $(1+\epsilon)$ approximation. Previously, this $\widetilde{O}(mn)$ bound

was only achieved for unweighted, undirected graphs [Roditty and Zwick, 2012; Henzinger et al., 2013]. In this chapter, we showed how to achieve the same bound for directed graphs with weights polynomial in n.

There remain a great deal of open questions for dynamic all pairs shortest paths. The question that most directly concerns our result is whether it would be possible to achieve the same bounds with a *deterministic* algorithm. Perhaps the two most important related open questions are: 1) Is it possible to achieve the same bounds for *exact* distances. Anything better than total update time $O(n^3)$ constitute a breakthrough, even for undirected unweighted graphs. 2) For the fully dynamic case, can one beat the $O(n^2)$ amortized update time in directed graphs, even with a constant approximation?

Part II

Fully Dynamic Maximum Matching

Chapter 6

Dynamic Matching Introduction

Computing a maximum matching is one of the fundamental problems in graph algorithms, and has a variety of applications (see e.g. [Mehta et al., 2005; Feldman et al., 2010]). In the fully dynamic version of the problem, the graph is subject to an online sequence of edge insertions and deletions, and the goal is to design an algorithm that always maintains a maximum (or approximately maximum) matching for the current graph. We could handle each insertion or deletion by recomputing a matching from scratch in $O(\sqrt{n}m)$ time [Micali and Vazirani, 1980a; Vazirani, 2012], but it is more efficient to take advantage of the fact that only one edge has changed. For example, in an unweighted graph, it would suffice to compute at most one augmenting path to update the matching, which yields a simple algorithm with $O(m)$ update time.

There is a great deal of existing work on the maximum matching problem. We describe previous work on dynamic matching in detail below (Section 6.2), but first we briefly mention the related problems of finding approximate and online matchings. Duan and Pettie showed how to find a $(1 + \epsilon)$-approximate weighted matching in nearly linear time [Duan and Pettie, 2014]; their paper also contains an excellent summary of the history of matching algorithms. There are many papers on "online matching" (e.g. [Mehta et al., 2005; Feldman et al., 2010]), both exact and approximate. This model also involves edges being added to the graph, but is in other ways very different from the dynamic model: in online matching there are no deletions, the matching cannot be altered, and the quality of the algorithm is judged by its competitiveness to the optimal offline algorithm. A related model measures the number of changes needed to maintain a matching [Chaudhuri et al., 2009; Gupta et al., 2014; Bosek et al., 2014].

6.1 Preliminaries

Unlike with dynamic shortest paths, almost all existing dynamic matching algorithms focus on the fully dynamic case, where the adversary can insert an edge, delete an edge, or change an edge weight (if the graph is weighted). It is most natural to think of dynamic matching algorithms as not having a query: the goal is to simply at all times maintain a matching that is a maximum (or approximately maximum) matching for the current graph. Most of the work on dynamic matching, including all of the new results presented in this thesis, are for the specific case of *unweighted* matching. Thus, for the sake of simplicity, all of our definitions will be for an unweighted graph.

Let $G = (V, E)$ be an undirected, unweighted graph. Let $n = |V|$ refer to the number of vertices in G (which never changes), and let $m = |E|$ refer to the number of edges in the *current* graph. A matching in G is a set of vertex-disjoint edges. A matching is sometimes referred to as an *integral* matching to distinguish it from fractional matchings. A vertex is called *matched* if it is incident to one of the edges in the matching, and *free* or *unmatched* otherwise. We let $|M|$ denote the size (number of edges) of a matching M, and $\mu(G)$ denote the size of the maximum matching in G. We say that a matching M is maximum if $|M| = \mu(G)$, and α-approximate ($\alpha \geq 1$) if $|M| \geq \mu(G)/\alpha$.

A fractional matching is an assignment of values to edges such that the total value of edges incident to any vertex is at most 1. We let $\text{VAL}(u, v)$ denote the value of edge (u, v) in a fractional matching. Given a vertex v, $\text{VAL}(v)$ will denote the sum of the values of all edges incident to v. We say a fractional matching is *feasible* if $\text{VAL}(v) \leq 1 \quad \forall v \in V$. Given some fractional matching M_f, $\text{VAL}(M_f)$ will denote the sum of all edge values in M_f. We let $\mu_f(G)$ denote the size of the maximum-valued fractional matching in G. Given a fractional matching $M_f(G)$, we let $\text{SUPPORT}(M_f(G))$ be the set of edges (u, v) for which $\text{VAL}(u, v) > 0$ in $M_f(G)$.

A graph $G = (V, E)$ is said to be bipartite if V can be partitioned into two sets L, R such that every edge $e \in E$ goes from L to R. We now state a well known theorem about the relationship between maximum integral matchings and maximum fractional matchings.

Theorem 11 *Let $G = (V, E)$ be an unweighted graph. Then $\mu(G) \geq \frac{2}{3}\mu_f(G)$, and given any maximum fractional matching $M_f(G)$, we have that $\mu(\text{SUPPORT}(M_f(G))) \geq \frac{2}{3}\mu(G)$. If G is bipartite, then $\mu(G) = \mu_f(G)$, and given any maximum fractional matching $M_f(G)$, we have that*

$\mu(\text{SUPPORT}(M_f(G))) = \mu(G)$.

Another important property of matching is *maximality*. A matching M of G is said to be *maximal* if there is no way to add an edge from E to M and still have a matching. In other words, a matching M is maximal for every edge $(u, v) \in E$, either u or v are matched in M. We now state a well known lemma about maximal matchings.

Lemma 28 *If a matching M in G is maximal, then it is 2-approximate. That is, $|M| \geq \mu(G)/2$. This inequality is tight.*

Several papers on dynamic matching focus on *small arboricity* graphs, which we now define.

Definition 20 *The* arboricity *of a graph $G = (V, E)$, denoted by $\alpha(G)$, is $\max_J \frac{|E(J)|}{|V(J)|-1}$ where $J = (V(J), E(J))$ is any subgraph of G induced by at least two vertices. Many classes of graphs in practice have constant arboricity, including planar graphs, graphs with bounded genus and graphs with bounded tree width. Every graph has arboricity $O(\sqrt{m})$.*

6.2 Previous Work

Fully dynamic matching algorithms can be classified by update time, approximation ratio, whether they are randomized or deterministic and whether they have a worst-case or amortized update time. As with dynamic shortest paths and most other dynamic problems, the distinction between deterministic and randomized is particularly important as all of the existing randomized algorithms for dynamic matching assume a substantially weaker adversary. The key idea behind the randomized matching algorithms is that if the algorithm chooses a *random* matching in a dense graph then the probability of an update deleting an edge in the matching is very small. This only works if the adversary is non-adaptive, that is if the update sequence must be fixed in advance. Put otherwise, the updates made by the adversary must be completely independent of the matching maintained by the algorithm. This is quite a big assumption, and in particular means that randomized (non-adaptive) dynamic matching algorithms cannot be used as black-box subroutines inside a larger data structure.

For maintaining an *exact* maximum matching in the fully dynamic setting, there is a simple algorithm that achieves $O(m)$ update time by taking advantage of the fact that inserting or deleting

a single edge can only change the matching by one augmenting path. The only other result for exact dynamic matching achieves update time $O(n^{1.495})$ (Sankowski [Sankowski, 2007]) , which is faster in dense graphs.

Most work on dynamic matching shows how to obtain faster update times than $O(m)$ and $O(n^{1.495})$ if we allow approximation. One relevant line of work focuses on approximation ratios 2 or worse. A straightforward greedy algorithm maintains a *maximal* matching in $O(n)$ time per update, which by Lemma 28 gives a 2-approximation to the *maximum* matching. Ivkovic and Lloyd [Ivkovic and Lloyd, 1994] showed how to improve the update time to $O((m+n)^{\sqrt{2}/2})$. Onak and Rubinfeld [Onak and Rubinfeld, 2010b] were the first to achieve to achieve truly fast update times, with a randomized algorithm that maintains a $O(1)$-approximate matching in amortized update time $O(\log^2 n)$ time (with high probability). Baswana *et al.* [Baswana et al., 2011b] presented an improved randomized algorithm that maintains a maximal matching (2-approximation) in $O(\log(n))$ amortized time per update. Both of these algorithms are extremely fast, but the techniques they contain are unlikely to break through the barrier of a 2-approximation, because these algorithms both compare themselves to a *maximal* (2-approximate) matching, not a maximum one.

Both of the algorithms with poly-log update time are also inherently randomized and non-adaptive, and developing a fast deterministic algorithm seems much harder. The fastest deterministic algorithm comes from a very recent paper of Bhattacharya, Henzinger, and Italiano [Bhattacharya et al., 2015b], and achieves a $(3+\epsilon)$ approximation with update time $O(m^{1/3})$. In the same paper, the authors also present a deterministic algorithm for maintaining a $(2+\epsilon)$ approximate *fractional* matching that has update time only $O(\epsilon^{-2}\log n)$; this fractional matching does not give us the edges of an integral matching (even in bipartite graphs), although by Theorem 11 it does yield a decent approximation to the *size* of the maximum integral matching. Finally, Neiman and Solomon [Neiman and Solomon, 2013b] showed that in graphs of constant arboricity, there is a deterministic algorithm that maintains a maximal (so 2-approximate) matching in amortized update time $O(\log(n)/\log\log(n))$. (Or *worst-case* update time $O(\log(n))$ using a recent dynamic orientation algorithm of Kopelowitz *et al.* [Kopelowitz et al., 2014a].)

Another line of work gives approximation ratios better than 2, but since a maximal matching no longer suffices for this case, the update times are much slower. Neiman and Solomon [Neiman and Solomon, 2013b] gave a *deterministic* algorithm for maintaining a 3/2-approximate matching, with

a worst-case update time of $O(\sqrt{m})$. Gupta and Peng [Gupta and Peng, 2013] later improved upon the approximation ratio, presenting a deterministic algorithm that maintains a $(1+\epsilon)$-approximate matching in worst-case update time $O(\sqrt{m}\epsilon^{-2})$. Before the result presented in Chapter 7, these were the only two algorithms for a better-than-2 approximation, and neither seems to contain any techniques for breaking past the $\sqrt{m}$ bound.

(There have been a few very recent results in dynamic matching that appeared after or simultaneously with the papers covered in this Thesis. We discuss this recent work after presenting our own results in 7.1.)

Very recently there have been some conditional lower bounds for dynamic approximate matching. Kopelowitz *et al.* [Kopelowitz et al., 2014b] show that assuming 3-sum hardness any algorithm that maintains a matching in which all augmenting paths have length at least 6 requires an update time of $\Omega(m^{1/3} - \zeta)$ for any fixed $\zeta > 0$. Henzinger *et al.* show that such an algorithm requires $\Omega(m^{1/2} - \zeta)$ time if one assumes the Online Matrix-Vector conjecture. The lower bound of Henzinger *et al.* conditionally proves that $o(\sqrt{m})$ update time it not possible for maintaining an exact matching, as such a matching contains no augmenting paths. (The existing upper bounds for exact matching are still far from this $\Omega(\sqrt{m})$ lower bound.) The lower bound also suggests that if we are to achieve a $o(\sqrt{m})$ update time for a $(1+\epsilon)$-approximate or even $6/5$-approximate matching, we cannot rely on an algorithm that tries to achieve a good approximation by finding and removing short augmenting paths.

Chapter 7

Fully Dynamic Matching with Small Approximation Ratios

Publication History: This chapter provides provides a full version of results that span two papers, both published in collaboration with my advisor Cliff Stein: the first paper was published in ICALP 2015 and was limited to bipartite graphs [Bernstein and Stein, 2015], while the second was published in SODA 2016 and extended our results to general graphs [Bernstein and Stein, 2016]. The former paper [Bernstein and Stein, 2015] received the best paper award for ICALP 2015 track A.

We can see from the discussion of previous work in Section 6.2 that existing algorithms for approximate matching fall into two main categories: fast algorithms that achieve a 2-or-worse approximation, and slow algorithms with $\Omega(\sqrt{m})$ update time that achieve a better-than-2 approximation. There is also a big gap between the fastest randomized algorithm [Baswana et al., 2011a] ($O(\log(n))$ update time, 2-approximation), and the fastest deterministic one [Bhattacharya et al., 2015a] ($O(m^{1/3})$ update time, $(3+\epsilon)$-approximation). We developed algorithms that make some progress towards bridging these gaps.

7.1 Our Results

Theorem 12 *Let G be a graph subject to a series of edge insertions and deletions, and let ϵ be < 1. We can maintain a $(3/2 + \epsilon)$-approximate matching in G in deterministic amortized update time $O(m^{1/4}\epsilon^{-2.5})$.*

Even allowing randomization, this is the first result to achieve $o(\sqrt{m})$ update time and a better-than-2 approximation. The algorithm is also faster than all previous deterministic for *any* constant approximation, and the constant we achieve is quite small. Also, since $m^{1/4} = O(\sqrt{n})$, our algorithm is the first to achieve a better-than-2 approximation in time strictly sublinear in the number of nodes.

For the special case of small arboricity graphs, we present an even more efficient algorithm that breaks through the maximal matching (2-approximation) barrier. Before our results, the best algorithms for constant arboricity graphs also had a fast update time (around $\log(n)$), but only maintained a 2-approximation.

Theorem 13 *Let G be a graph subject to a series of edge insertions and deletions, and let ϵ be < 1. Say that at all times G has arboricity at most α. Then, we can maintain a $(3/2 + \epsilon)$-approximate matching in G in deterministic amortized update time $O(\alpha(\alpha + \log(n) + \epsilon^{-2}) + \epsilon^{-6})$. For constant α and ϵ the update time is $O(\log(n))$, and for α and ϵ polylogarithmic the update time is polylogarithmic.*

Theorem 14 *Let G be a graph subject to a series of edge insertions and deletions, and let $\epsilon < 1$. Say that at all times G has arboricity at most α. Then, we can maintain a $(1 + \epsilon)$-approximate* **fractional** *matching in G in deterministic amortized update time $O(\alpha(\alpha + \log n + \epsilon^{-4}) + \epsilon^{-6})$.*

Our algorithms introduce the (as far as we can tell) new notion of a γ-restricted fractional matching, and prove an accompanying theorem which we believe might be of independent interest.

Definition 21 *A γ-restricted fractional matching is a fractional matching assigning values to the edges such that for all edges e, $\text{value}(e)$ is either 1 or in the interval $[0, \gamma]$.*

It is well known that there always exists a maximum fractional matching with values 0, $1/2$ or 1 [Scheinerman and Ullman, 2011] (i.e. a $1/2$-restricted fractional matching). Moreover, the support

of this matching contains an integral matching of at least $2/3$ the value of the fractional matching. We prove a generalization of this second fact for smaller γ.

Theorem 15 *Given a graph G, let M_f be a γ-restricted fractional matching, and let M be a maximum integral matching in the graph formed by edges in the support of M_f. Then $|M| \geq$* $\text{value}(M_f)\frac{1}{\gamma+1}$.

Observe that this bound is tight when $\gamma = 1/c$ for some even c. Consider a clique on $c+1$ vertices. The fractional solution places value $1/c$ on each edge and has total weight $\binom{c+1}{2}/c = (c+1)/2$. But the best integral matching has $c/2$ edges.

New Work Published After our Results: There have been several papers on dynamic matching published simultaneously with or after the results presented in this chapter [Bernstein and Stein, 2016].

For the special case of small arboricity graphs, Peleg and Solomon [Peleg and Solomon, 2016] developed an approach that is simpler and faster than ours; whereas our paper [Bernstein and Stein, 2016] focused on general graphs and then showed that our framework yields improved results when the graph has small arboricity, their paper is specific to small arboricity graphs, and is able to take advantage of their unique properties. As a result, they show that given a dynamic graph that always has arboricity at most α, one can maintain a $(1+\epsilon)$-approximate matching with *worst-case* update time $O(\alpha)$. In particular, for constant arboricity and fixed ϵ, they achieve constant worst-case update time, compared to our $\log(n)$ amortized update time.

Very recently, Bhattacharya, Henzinger, and Nanongkai [Bhattacharya et al., 2016] settled the open question of *deterministically* maintaining an $O(1)$-approximate matching with polylog update time – their approximation ratio is $(2+\epsilon)$, which almost matches the randomized 2-approximation.

Turning to better-than-2 approximations, the same paper shows that in *bipartite* graphs, one can maintain a better-than-2 approximation to the *size* of the matching (but not the matching itself), with an arbitrary small polynomial update time. That is, for any positive constant $\zeta < 1$, there is some constant $k_\zeta < 2$ such that the algorithm maintains a $k_\zeta -$approximation with update time $O(n^\zeta)$. This algorithm is extremely important as a proof of possibility, as it gives good evidence that it might be possible to maintain a better-than-2 approximation with only polylog update time, which is perhaps the main open problem in the field. However, the result is at its current stage quite limited:

it only works for bipartite graphs, it only approximates the *size* of the matching (not the matching itself), the update time is a small polynomial (as opposed to polylog), and the approximation ratio is only barely better than 2 (no matter what the settings, the approximation ratio is always worse than 1.99).

Outline: Section 7.2 discusses the key techniques developed towards our new result. Section 7.3 then discusses notation and preliminaries, while Section 7.4 outlines the high-level framework of our algorithm. A more detailed outline of the chapter is given in Section 7.4, after the framework has been defined. Note that our result is rather technical, so for the sake of clarity, many of the formal proofs have been relegated to the end of the chapter (Section 7.8).

7.2 Techniques

As we pointed out above, there is a huge gap in update time between algorithms that settle for a 2-or-worse approximation, and those that achieve a better-than-2 approximation. The reason for this is that if one only requires a 2-approximation (or worse), it is enough to maintain a *maximal* matching (or an approximation to a maximal matching). Intuitively, maintaining a maximal matching is much easier than maintaining a maximum one because the former problem is completely local in the sense that if an edge can be legally added to the matching, then the algorithm can *always* safely do so and still end up with a maximal matching.. The moment one seeks even a 1.99 matching, however, the algorithm must be capable of rejecting an available edge because of how it interacts with other edges. Local problems tend to be much easier in the dynamic setting, because they allow us to handle an edge change without having to look at large portions of the graph.

This difference in locality can be clearly seen in our characterization of approximate matchings. The maximality condition (2-approximation) is expressed through a local constraint on each edge that depends only on the endpoints of that edge: namely, for every edge (u, v), either u or v must be matched. On the other hand, the majority of existing matching algorithms characterize better-than-2 approximations through the non-existence of short augmenting paths: a matching M is $3/2$-approximate if it contains no augmenting paths of length 3 or less; a matching M is $(1 + \epsilon)$-approximate if it contains no augmenting paths of length $2\epsilon^{-1} + 2$ or less. The problem is that the latter characterization cannot be cleanly expressed in terms of local constraints on each edge, even

if one is only dealing with augmenting paths of length 7. And while determining whether an edge (u, v) belongs in a maximal matching is very easy (if u and v are free, add the edge), determining whether an edge (u, v) belongs an augmenting path of length 7 could require use to search a large portion of the graph. Augmenting paths of length 3 are a border case, but still don't yield clean local constraints like those of a maximal matching.

Since augmenting paths seem not very suitable to the dynamic setting, we would like to find a characterization of better-than-2 approximate matchings in terms of local constraints. We were unable to do this directly, but we were able to do something almost as good: we show that we can define a small subgraph H of G such that H is defined only in terms of local constraints, and yet H is guaranteed to contain a large matching: one of our subgraphs has $\mu(H) \geq (2/3 - \epsilon)\mu(G)$, and the other has $\mu(H) \geq (1 - \epsilon)\mu(G)$. We refer to H as an *Edge Degree Constrained Subgraph* (EDCS). The idea is that it is easy to maintain a matching in the EDCS H because H is small and in particular has bounded degree; on the other hand, it is easy to maintain H in G because H is defined in terms of local constraints.

The idea of using a transition subgraph graph H was first introduced in an earlier paper of Bhattacharya *et al.* [Bhattacharya et al., 2015a], which presented the previous fastest deterministic algorithm: $O(m^{1/3})$ update time and a $(3 + \epsilon)$ approximation. Their transition subgraph, however, continued to rely on maximality constraints, and was in particular akin to a maximal B-matching. Their approach thus contained no potential for breaking through the 2-approximation barrier (in fact due to other difficulties they only achieve $(3 + \epsilon)$). This is not surprising, as they never set out to go beyond a 2-approximation: their goal was to achieve a $o(\sqrt{m})$ update time with a deterministic algorithm, and in this they succeeded.

Our main contribution is the EDCS, which is a subgraph H defined in terms of a different (non-maximality) set of local constraints that yields a better-than-2 approximation. Maintaining an EDCS is somewhat harder than maintaining the subgraph with maximality constraints used by Bhattacharya *et al.* [Bhattacharya et al., 2015a]. We overcome this with a careful analysis of the structure of an EDCS, as well as with a new algorithmic approach that relies on dynamic orientations. This approach works especially well in small arboricity graphs, but also leads to an improvement in the general case because the arboricity of a graph is always $O(\sqrt{m})$.

7.3 Preliminaries

Recall the general matching notation from Section 6.1. Our main graph $G = (V, E)$ is an undirected, unweighted graph where $|V| = n$ and $|E| = m$. Since the graph is dynamically changing over time, G will always refer to the *current* version of the graph. Our algorithm will often deal with auxiliary graphs that differ from G, so all of our notation will be explicit about the graph in question. We define $d_G(v)$ to be the degree of a vertex v in G; if the graph in question is weighted, then $d_G(v)$ is the sum of the weights of all incident edges. We define *edge degree* as $\delta(u, v) = d(u) + d(v)$. If H is a subgraph of G, we say that an edge in G is *used* if it is also in H, and *unused* if it is not in H. Throughout this paper we will only be dealing with subgraphs H that contain the full vertex set of G, so we will use the notion of a subgraph and of a subset of edges of G interchangeably. Recall the set difference operator $Z_1 \setminus Z_2 = \{x \in Z_1 | x \notin Z_2\}$.

We will often compute the value of a fractional matching by summing over the edge values incident to each vertex. To avoid double counting edges, we let each endpoint of an edge account for a fraction of that edge. More formally, given a graph G, we define an *accounting* of the edges to be a function that assigns to each edge (x, y) two values $a_x(x, y)$ and $a_y(x, y)$ such that $a_x(x, y) + a_y(x, y) \leq 1$. Given a fractional matching M_f and some accounting of the edges, we define the *profit* of x, $\rho(x)$, to be $\rho(x) = \sum_{(x,y)\in E} a_x(x, y)\text{VAL}(x, y)$.

Observation 1 *Given some fractional matching M_f in G, and some accounting of the edges of G, we always have $\text{VAL}(M_f) \geq \sum_{x\in V} \rho(x)$. This inequality holds with equality when, for each edge (x, y), we have $a_x(x, y) + a_y(x, y) = 1$.*

We now state a simple corollary of an existing result of [Gupta and Peng, 2013].

Lemma 29 ([Gupta and Peng, 2013]) *If a dynamic graph G has maximum degree B at all times, then we can maintain a $(1 + \epsilon)$-approximate matching under insertions and deletions in worst-case update time $O(B\epsilon^{-2})$ per update.*

Proof: This lemma immediately follows from a simple algorithm presented in Section 3.2 of [Gupta and Peng, 2013] which shows how to achieve update time $|E(G)|\epsilon^{-2}/\mu(G)$ (for the transition from worst-case to amortized see appendix A.3 of the same paper), as well as the fact that we always have $|E(G)|/\mu(G) \leq 2B$ because all edges must be incident to one of the $2\mu(G)$ matched vertices in the maximum matching, and each of those vertices have degree at most B. □

Orientations An orientation G' of an undirected graph G is an assignment of a direction to each edge in E. Given an orientation of edge (u, v) from u to v, we will say that u *owns* edge (u, v) and will define the *load* of a vertex u to be the number of edges owned by u. Orientations of small max load are closely linked to arboricity: every graph with arboricity α has an orientation with max load $O(\alpha)$ [Nash-Williams, 1961]. Our algorithms will maintain an orientation of the *dynamic* graph G using the algorithms referred to in the theorems below. The first result (Theorem 16) is due to Kopelowitz *et al.* [Kopelowitz et al., 2014a], while the second (Theorem 17) is a simple result that is new to this paper. We leave the proof of Theorem 17 for Section 7.8.5

Theorem 16 ([Kopelowitz et al., 2014a]) *Let G be a graph that always has arboricity at most α. One can maintain an orientation, under edge insertions and deletions, with the following properties: the maximum load at all times is $O(\alpha + \log n)$, the worst-case number of edge reorientations per insertion/deletion is also $O((\alpha + \log n))$, and the worst-case time to process an insertion/deletion in G is $O(\alpha(\alpha + \log n))$.*

Theorem 17 *In a graph G, we can maintain an orientation, under insertions and deletions, with the following properties: the max load at all times is at most $3\sqrt{m}$, the worst-case number of edge reorientations per insert/deletion in G is $O(1)$, and the worst-case time spent per insertion/deletion in G is $O(1)$.*

7.4 The Framework

Definition 22 *An unweighted edge degree constrained subgraph, denoted* $\mathrm{EDCS}(G, \beta, \beta^-)$ *is a subset of the edges $H \subseteq E$ (we will also refer to it as a subgraph) with the following properties:*

(P1) if (u, v) is used (i.e. $(u, v) \in H$) then $d_H(u) + d_H(v) \leq \beta$,

(P2) if (u, v) is unused (i.e. $(u, v) \in G \setminus H$) then $d_H(u) + d_H(v) \geq \beta^-$.

We also define a similar subgraph where edges in H have positive integer weights, effectively allowing them to be used more than once. Note that $d_H(v)$ now refers to the sum of the weights of v's incident edges.

Definition 23 *A weighted edge degree constrained subgraph(*EDCS*)* (G, β, β^-) *is a subset of the edges* $H \subseteq E$ *with positive integer weights that has the following properties:*

(P1) if (u, v) *is used then* $d_H(u) + d_H(v) \leq \beta$*,*

(P2) for all *edges* (u, v)*, we have* $d_H(u) + d_H(v) \geq \beta^-$.

Below is an outline of how our algorithm processes an edge insertion/deletion in G:

1. Update the small-max-load edge orientation using either Theorem 17 or Theorem 16 .
2. Update the subgraph H so it remains a valid edge degree constrained subgraph of the changed graph G. This step uses the orientation from step 1 for efficiency. (See Section 7.7.)
3. Update the $(1 + \epsilon)$-approx. matching in H with respect to the changes in H from step 2. (Lemma 29.)

The maintained $(1 + \epsilon)$-approximate matching of H (step 3) is also our final matching in G; much of this chapter devoted to showing that because H is an EDCS, $\mu(H)$ is not too far from $\mu(G)$.

Because much of our algorithm runs on a dynamic subgraph of G we need the following definition: Let H be a subgraph of a dynamic graph G, and let A be an algorithm that modifies the edges of H as G changes; then, we say that A has an *amortized update ratio* of r if for any large enough sequence S of edge changes (insertions or deletions) to G, the algorithm makes at most $r|S|$ edge changes to H.

We can now state the main theorems of the paper. We present general and small arboricity graphs separately, but the basic framework described above remains the same in both cases. In all the theorems below, the parameter $\epsilon > 0$ is chosen to obtain a desired approximation ratio (either $(1 + \epsilon)$ or $(3/2 + \epsilon)$).

7.4.1 General Graphs

For the sake of intuition, in the two theorems below, think of β as quite large (roughly $m^{1/4}$). Also recall that $\mu(H)$ denotes the size of the maximum matching in H.

Theorem 18 *Let G be a graph and let $\lambda = \epsilon/6$. Let H be an unweighted* EDCS *of G with $\beta^- = \beta(1-\lambda)$, where $\beta \geq 32\lambda^{-3}$ is a parameter we will choose later. Then $\mu(H) \geq (2/3 - 2\epsilon)\mu(G)$.*

Theorem 19 *Let G be a graph. Let H be an unweighted* EDCS *of G with $\beta^- = \beta(1-\lambda)$, with $\beta \geq 36\lambda^{-1}$. There is an algorithm that maintains H over updates in G (i.e. maintains H as a valid* EDCS*) with the following properties:*

- *The algorithm has amortized update time $O\left(\frac{\sqrt{m}}{\lambda^2\beta}\right)$.*
- *The amortized update ratio of the algorithm is $O(1/\lambda)$.*

Proof of Theorem 12 We use the algorithm outline presented at the beginning of Section 7.4. We set H to be an unweighted EDCS$(G, \beta, \beta(1-\lambda))$ with $\lambda = \epsilon/6$ and $\beta = m^{1/4}\epsilon^{1/2}$. By Theorem 19 we can maintain H in amortized update time $O(\frac{\sqrt{m}}{\lambda^2\beta}) = O(m^{1/4}\epsilon^{-2.5})$. The update-ratio is $O(\lambda^{-1}) = O(\epsilon^{-1})$. Since degrees in H are clearly bounded by β, by Lemma 29 we can maintain a $(1+\epsilon)$-approx. matching in H in time $O(\beta\epsilon^{-2})$; multiplying by the update ratio of maintaining H in G, we need $O(\beta\epsilon^{-3}) = O(m^{1/4}\epsilon^{-2.5})$ time to maintain the matching per change in G. By Theorem 18, $\mu(H)$ is a $(3/2 + 2\epsilon)$-approximation to $\mu(G)$, so our matching is a $(3/2 + 2\epsilon)(1+\epsilon) = (3/2 + O(\epsilon))$-approx. matching in G. □

7.4.2 Small Arboricity Graphs

For the sake of intuition, in the two theorems below think of β as quite small, roughly $O(\epsilon^{-2})$.

Theorem 20 *Let G be a graph, and let $\beta \geq 25\epsilon^{-2}$. Let H be a weighted* EDCS *with $\beta^- = \beta(1-\lambda)$, where $\lambda = \epsilon^2/25$. Consider the fractional matching M_f^H in H, with $\text{VAL}(u,v) = 1/\max\{d_H(u), d_H(v)\}$. Then M_f^H is a feasible fractional matching, and $\text{VAL}(M_f^H) \geq \mu_f(G)(1-\epsilon)$.*

Theorem 21 *Let G be a dynamic graph that at all times has arboricity $\leq \alpha$. Let H be a weighted* EDCS$(G, \beta, \beta(1-\lambda))$ *with $\beta \geq 4\lambda^{-1}$. There is an algorithm that maintains H over updates in G with the following properties:*

- *The algorithm has amortized update time $O(\alpha(\alpha + \log n + \beta\lambda^{-1}))$.*

- *The amortized update ratio of the algorithm is $O(\beta\lambda^{-1})$.*

If H is instead an unweighted EDCS$(G, \beta, \beta(1-\lambda))$*, the algorithm has amortized update time $O(\alpha(\alpha + \log n + \lambda^{-1}))$ and amortized update ratio $O(\lambda^{-1})$.*

Proof of Theorem 13 The proof is essentially identical to that Theorem 12; for the transition subgraph H, we use an (unweighted) EDCS$(G, \beta, \beta(1-\lambda))$ with $\beta = 100\epsilon^{-2}$ and $\lambda = \epsilon^2/25$ (so $\beta(1-\lambda) = \beta - 4$). □

Proof of Theorem 14 Let H be a weighted EDCS$(G, \beta, \beta(1-\lambda))$ with $\beta = 100\epsilon^{-2}$ and $\lambda = \epsilon^2/25$. By Theorem 21 we can maintain H in amortized update time $O(\alpha(\alpha + \log n + \epsilon^{-4}))$ and amortized update ratio $O(\epsilon^{-4})$.

We now need to maintain a fractional matching in H. We cannot rely on Lemma 29 as in the previous proofs, because this lemma only applies to maintaining an *integer* matching. Instead, we explicitly maintain the fractional matching M_f^H defined in Theorem 20. By definition of update ratio, every update in G only changes an average of $O(\epsilon^{-4})$ edges in H, so only $O(\epsilon^{-4})$ vertices v have their degree $d_H(v)$ changed. For each such vertex v, we spend $O(\beta) = O(\epsilon^{-2})$ time going through its incident edges in H and updating their value to the new $1/\max\{d_H(u), d_H(v)\}$. This yields a total update time of $O(\epsilon^{-6})$ for maintaining M_f^H in H. By Theorem 20, M_f^H is the desired $(1+\epsilon)$ approximation to $\mu_f(G)$. □

Overview of the paper Section 7.5 proves Theorem 15 relating to the new notion of a γ-restricted fraction matching defined in the introduction. The first half of Section 7.6 proves Theorem 20, while the second is devoted to proving Theorem 18. Theorems 19 and 21 are discussed in Section 7.7. For ease of reading, some of the more technical proofs are left for the proofs section at the end of the paper (Section 7.8).

7.5 A gamma-Restricted Fractional Matching Contains a Large Integral Matching

To prove Theorem 15, we first prove a structural theorem about matchings in non-bipartite graphs. In bipartite graphs, because of the relationship between matching and flows, a maximum matching

induces a cut in the graph. The following is an attempt to exhibit an analogous property for non-bipartite graphs.

Lemma 30 *Let M be a maximum matching in a graph G. We can partition the vertices of G into three sets, C (connected) , L (lonely) and B (both) such that the following properties hold:*

1. *All free vertices are in L.*

2. *Each vertex in C is matched to a vertex in L.*

3. *There are no edges in $L \times L$.*

4. *Each vertex in B has at most one neighboring vertex in L.*

Proof: Given a graph G with maximum matching M, we define a free-free path to be a (non-empty) simple alternating path that starts and ends with edges not in M. We define a free-matched path to be a (non-empty) simple alternating path that starts with an edge not in M and ends with an edge in M. Note that a free-free path need not start or end with a free vertex. We now define two vertex sets FER (free-edge-reachable) and MER (matched-edge-reachable). We say that a vertex v is in MER if there is a free-matched path from some free vertex w to v. Similarly, we say that a vertex v is in FER if there is a free-free path from some free vertex w to v. Note that FER and MER are not necessarily disjoint, and there may be vertices that are in neither set. We also let F denote the set of free vertices. F is disjoint from MER by definition. Observe that F is also disjoint from FER because M is a maximum matching and so contains no augmenting paths; suppose, for contradiction, that $v \in F \cap \text{FER}$: then the free vertex v is reachable by some free-free path from a free vertex w, which is precisely an augmenting path from w to v.

It is not hard to check that in bipartite graphs FER and MER are disjoint, because if a vertex v was in both sets then there would be an augmenting path in the matching. Thus, in a bipartite graph we would achieve the desired partition by setting $C = \text{FER}, L = \text{MER} \cup F, B = V - C - L$. It is not hard to check that this partition satisfies all four properties of Lemma 30, and that in fact it achieves a stronger version of property 4, where there are *no* edges from B to L. In a non-bipartite graph however, there could be vertices which are in both FER and MER, which roughly correspond to vertices in blossoms. For this reason we need a more involved definition of the set B which ends up indirectly capturing all the blossom vertices.

For nonbipartite graphs, we define the *connected* set $C = \text{FER} \setminus \text{MER}$ and *lonely* set $L = (\text{MER} \setminus \text{FER}) \cup F$. We then define $B = V \setminus (C \cup L)$. Informally, B contains non-free vertices that either are in both MER and FER or are not reachable by any alternating path from a free vertex.

We now proceed to verify the conditions of the theorem. The first condition is immediate from the definition of L. The second condition says that each vertex in C is matched to a vertex in L. To prove this, we note that each vertex v in C is also in FER, so it is reachable by a free-free path P_v from a free vertex w. We also know that v is matched since FER is disjoint from F, so let x be the vertex to which v is matched. We want to show that $x \in \text{MER} \setminus \text{FER}$. Note that the free-free path P_v cannot contain x as then it would end on the matched edge (x, v). The path P_v followed by edge (v, x) is thus a simple free-matched alternating path from w to x and so $x \in \text{MER}$. Now assume, fpoc, that also $x \in \text{FER}$. Then there is a free-free path P_x from a free vertex w' (which could be the same as w). But then P_x followed by (x, v) is a free-matched path from w' to v and therefore $v \in \text{MER}$, which contradicts the assumption that $v \in C$. (Note that P_x cannot already contain v earlier in the path, for if it did it would also contain x as an interior vertex, which contradicts P_x being simple).

To prove the third condition, assume fpoc that there is an edge $(x, y) \in L \times L$. There are three cases to consider, depending on whether x and y are free. If both x and y are free, then there is an augmenting path from (x, y), contradicting M being a maximum matching. Now consider the case where one vertex, say x, is free, and $y \in \text{MER} \setminus \text{FER}$. Then the edge (x, y) is a free-free path, so $y \in \text{FER}$, a contradiction. The last case to consider is where both x and y are in $\text{MER} \setminus \text{FER}$, yet the edge (x, y) exists. By definition, there is some free-matched path P_x from a free vertex w to x. Now, if $y \notin P_x$ then the path P_x followed by edge (x, y) is a simple free-free path from w to y, so $y \in \text{FER}$ – contradiction. If $y \in P_x$ then let P_y be the subpath of P_x from w to y. Since $y \notin \text{FER}$, P_y ends on a matched edge; thus the path P_y followed by edge (y, x) is a free-free path from w to x, so $x \in \text{FER}$, a contradiction.

The fourth condition states that each vertex in B is incident to at most one vertex in L. Consider a vertex $v \in B$. There are two cases to consider: $v \in \text{FER} \cap \text{MER}$ and $v \in V - \text{FER} - \text{MER}$. In the latter case, there clearly cannot be an edge (v, x) to some $x \in L$, since if x is free then the existence of edge (x, v) would imply that $v \in \text{FER}$, contradicting our assumption; similarly, if $x \in \text{MER}$, then there is some free-matched path P_x from a free vertex to x, and so considering the

path P_x followed by edge (x, v), we would have that $v \in \text{FER}$. We now consider the case where $v \in \text{FER} \cap \text{MER}$. Since $v \in \text{MER}$, it is the end of some free-matched path P_v starting at a free vertex w, and is possibly the end of many such paths. Fix one such path P_v, let w be the free vertex at the start of the path, and define the *apex* $\text{APEX}(P)$ to be, among all vertices in $P_v \cap L$, the one closest to v on P. We now show that the only possible edge from v to a vertex $x \in L$ is the edge $(v, \text{APEX}(P))$. We will establish this fact by ruling out all other possible edges. To this end, there are three cases to consider: $x \notin P$, $x \in P$ and is between between $\text{APEX}(P)$ and v, and $x \in P$ and between w and $\text{APEX}(P)$. In the first case, the path P followed by edge (v, x) is simple and hence free-free, and so $x \in \text{FER}$ and not in L. The second case cannot occur by the definition of APEX, for x would be the $\text{APEX}(P)$ in this case. In the third case, we claim that if there is such an edge, then $\text{APEX}(P) \in \text{FER}$, contradicting the fact that $\text{APEX}(P) \in L$. We show this by observing that there is a free-free path consisting of the portion of P from w to x, followed by the edge (x, v), and then followed by the portion of P from v to $\text{APEX}(P)$. So we have ruled out all possible edges from v to L except the edge $(v, \text{APEX}(P))$.

It might seem strange that we picked one specific free-matched path P_v and proved that the only possible edge from v to L is to the apex of that path, since in fact, there can be many free-matched paths to v. But the proof actually shows that for *any* free-matched path P from a free vertex to v, the *only* possible edge from v to L is $(v, \text{APEX}(P))$; thus, if there are two different free-matched paths to v with different apexes then we have shown that there are in fact *no* edges from v to L. $\square$

Proof of Theorem 15 Let M_f be our γ-restricted fractional matching in G, let G^* be the support of M_f and let M^* be the maximum integer matching in G^*. We want to show that $\text{VAL}(M_f) \leq |M^*|(1 + \gamma)$.

We can assume by induction on the size of M_f that all edges in M_f have value $\leq \gamma$; if some edge (x, y) had value 1, then G^* contains no other edges incident to x or y, so by the induction hypothesis we could find an integral matching of size at least $(\text{VAL}(M_f) - 1)/(1 + \gamma)$ in $G^* - \{x\} - \{y\}$. We could then add edge (x, y) to this integral matching, yielding the desired matching of size $\geq (\text{VAL}(M_f) - 1)/(1 + \gamma) + 1 > \text{VAL}(M_f)/(1 + \gamma)$.

We can partition the vertices into sets C, L, and B, that satisfy Lemma 30 with respect to the graph G^* and the matching M^*. To bound $\text{VAL}(M_f)$, we consider the following accounting of the edges in G^*. For an edge $(x, y) \in C \times C$, let $a_x(x, y) = a_y(x, y) = 1/2$. For an edge

$(x, y) \in C \times V \setminus C$, let $a_x(x, y) = 1$ and $a_y(x, y) = 0$. For an edge $(x, y) \in B \times B$, let $a_x(x, y) = a_y(x, y) = 1/2$, For an edge $(x, y) \in B \times L$, let $a_x(x, y) = 1$ and $a_y(x, y) = 0$. Note that by property 3 of Lemma 30, G^* contains no edges in $L \times L$.

Note that for all edges (x, y) we have $a_x(x, y) + a_y(x, y) = 1$. Thus, combining Observation 1 with the fact that vertices in L receive no profit under this accounting:

$$\textsc{val}(M_f) = \sum_{x \in V} \rho(x) = \sum_{x \in C} \rho(x) + \sum_{x \in B} \rho(x) .$$

To upper bound the total profit of C, we simply observe that since M_f is a fractional matching, for any vertex x, $\textsc{val}(x) \leq 1$ and $\rho(x) \leq 1$, so $\sum_{x \in C} \rho(x) \leq |C|$. To upper bound the total profit from B, consider an $x \in B$, and let $\textsc{val}_L(x)$ be the total value of all edges from x to L. Then the total value of all edges from x to $V \setminus L$ is at most $1 - \textsc{val}_L(x)$, and since our accounting only lets x account for at most half of each edge to $V \setminus L$, we have that $\rho(x) \leq \textsc{val}_L(x) + (1 - \textsc{val}_L(x))/2 = (\textsc{val}_L(x) + 1)/2$. But by property 4 of Lemma 30 there is at most one edge from x to L, and since M_f is by assumption a γ-restricted fractional matching, we have that $\textsc{val}_L(x) \leq \gamma$, so $\rho(x) \leq (1 + \gamma)/2$. We thus have

$$\begin{aligned} \textsc{val}(M_f) &= \sum_{x \in C} \rho(x) + \sum_{x \in B} \rho(x) \\ &\leq |C| + |B|(1 + \gamma)/2 \\ &\leq (1 + \gamma)(|C| + |B|/2) . \end{aligned} \tag{7.1}$$

There are $2|M^*|$ matched vertices in total; by property 1 of Lemma 30 all vertices in C and B are matched, and by property 2 there are at least $|C|$ matched vertices in L, so $2|C| + |B| \leq 2|M^*|$, so $|C| + |B|/2 \leq |M^*|$. By Equation 7.1 above this implies that $\textsc{val}(M_f) \leq (1 + \gamma)|M^*|$, as desired.

□

7.6 An Edge Degree Constrained Subgraph Contains a Large Matching

A Weighted EDCS **Contains a Large Fractional Matching** We now sketch the proof of Theorem 20. The proof is conceptually quite simple, but somewhat technical because of the λ slackness in property P2 of a weighted EDCS. Thus, we start by giving a proof that assumes a tighter version of property P2, which we call $P2'$: that for any edge (u, v) in G, $d_H(u) + d_H(v) \geq \beta$. We leave the full proof without this assumption for Section 7.8.1.

Proof Sketch of Theorem 20 We will start by defining a simple accounting on the edges of H. If $d_H(v) > \beta/2$ we set $a_v(v,w) = 1$ for all edges (v,w). If $d_H(v) < \beta/2$ we set $a_v(v,w) = 0$ for all edges (v,w). If $d_H(v) = \beta/2$ we set $a_v(v,w) = 1/2$ for all edges (v,w). Property P1 of an EDCS clearly ensures that this is a valid accounting in that $a_v(v,w) + a_w(v,w) \leq 1$ for any edge (v,w) in H.

Recall that for every edge $(u,v) \in H$, M_f^H assigns $\text{VAL}(u,v) = 1/\max\{d_H(u), d_H(v)\}$. Given the accounting above, there is a simple formula for the profit of vertex v. If $d_H(v) > \beta/2$ then, by property P1 of an EDCS, for every edge (v,w) we have $d_H(w) < d_H(v)$, so $\text{VAL}(v,w) = 1/\max\{d_H(u), d_H(v)\} = 1/d_H(v)$. Since v has $d_H(v)$ incident edges and full accounting of all of them, we have $\rho(v) = 1$. Similarly if $d_H(v) = \beta/2$ then $\rho(v) = 1/2$, since v now only accounts for half of each incident edge. Finally, if $d_H(v) < \beta/2$, then v does not account for its edges, so clearly $\rho(v) = 0$.

Now, it is well-known [Scheinerman and Ullman, 2011] that there must exist a maximum fractional matching in which all edge values are 0, $1/2$ or 1. From this fact, we can easily show G must contain some maximum fractional matching that consists of disjoint odd cycles and disjoint isolated edges; the isolated edges all have value 1, the edges on cycles have value 1/2, and all other edges have value 0. To verify this statement, observe first that all the 1 edges must be isolated in any matching. If we remove the 1 edges from the matching, the remaining edges with value $1/2$ must form a set of disjoint cycles (that are also disjoint from the set of removed 1 edges). Any even cycle can be replaced with a cycle that alternates between edges of value 0 and edges of value 1, leaving only odd cycles for the $1/2$-edges. Let M_f^G be such a maximum fractional matching in G. M_f^G gets a value of 1 from each isolated edge, and a value of $d/2$ for each odd cycle of length d; we will show that M_f^H receives exactly the same amount from each edge / odd cycle.

For each isolated edge (v,w) in M_f^G we have by property $P2'$ (the stronger version of $P2$ assumed for this proof sketch) that $d_H(v) + d_H(w) \geq \beta$, so either one of $d_H(v)$ or $d_H(w)$ is larger than $\beta/2$, or both are equal to $\beta/2$; either way, $\rho(v) + \rho(w) \geq 1$, so M_f^H gets a value of 1 from the edge, as desired. For a cycle of odd length d, let v be the vertex on the cycle with maximum $d_H(v)$. It is not hard to see that $d_H(v) \geq \beta/2$, since otherwise, by property $P2'$, the neighbors of v on the cycle would have degree higher than $d_H(v)$. Thus, we have that $\rho(v) \geq 1/2$. The remainder of the cycle clearly contains $(d-1)/2$ disjoint edges in G, and by the argument above, for each of

these edges M_f^H gains a profit of at least 1. Thus, the total profit of M_f^H from the cycle is at least $1/2 + (d-1)/2 = d/2$. □

An unweighted EDCS **Contains a Large Fractional Matching** We now turn to general graphs, and sketch the proof of the key lemma used to prove Theorem 18.

Lemma 31 *Let H be a edge degree constrained subgraph$(G, \beta, \beta(1-\lambda))$ with $\lambda = \epsilon/6$ and $\beta \geq 32\lambda^{-3}$. Then H contains an ϵ-restricted fractional matching M_f^H with total value at least $\mu(G)(2/3 - \epsilon)$.*

Proof: (sketch) As in the proof of Theorem 20 earlier in this section,we explicitly construct a fractional matching; in this case, a ϵ-restricted one. Unfortunately, this is by far the most complex proof in our paper, and it relies on the probabilistic method; if we can construct a ϵ-restricted matching that has size k in *in expectation*, then a ϵ-restricted matching of size k is guaranteed to exist. Note that our overall algorithm is still deterministic because we only use randomization in the analysis. We suspect that there exists a more natural construction of a large ϵ-restricted matching, but have so far been unable to find it. For this reason, we leave the full proof of Lemma 31 for Section 7.8.2. Here we give a sketch of the proof, which makes several simplifying assumptions that among other things allow us to avoid recourse to the probabilistic method.

Let M^G be some maximum matching in G. Let M_H^G be the edges of M^G in H, and let $M_{G \setminus H}^G$ be the edges of M^G in $G \setminus H$, so $|M_H^G| + |M_{G \setminus H}^G| = |M^G| = \mu(G)$. Let the vertex sets S^G, S_H^G, and $S_{G \setminus H}^G$ contain the endpoints of the edges of M^G, M_H^G, and $M_{G \setminus H}^G$ respectively.

Let us consider the properties of S_H^G and $S_{G \setminus H}^G$. Firstly, S_H^G contains a perfect matching using edges in H; namely, the edges of M_H^G. Secondly, $|S_H^G| + |S_{G \setminus H}^G| = 2(|M_H^G| + |M_{G \setminus H}^G|) = 2\mu(G)$. Thirdly, for every edge $(u, v) \in S_{G \setminus H}^G$ we have by property P2 of an EDCS that $d_H(u) + d_H(v) \geq \beta(1-\epsilon)$. Thus, the *average* degree $d_H(v)$ of vertices in $S_{G \setminus H}^G$ is at least $\beta(1-\epsilon)/2$. For this proof sketch, let us make a big simplifying assumption: *all* vertices in $S_{G \setminus H}^G$ have degree *exactly* $\beta/2$. Note that dropping the $(1-\epsilon)$ factor is a minor assumption we could easily do without, but setting all degrees to be the same makes the proof qualitatively simpler, and allows us to avoid recourse to the probabilistic method. Finally, as we show in the full proof, if we pick our maximum matching M^G carefully we can preserve the properties above while also ensuring the fourth property below. (This is a simplified version of Lemma 37 in the full proof of Lemma 31.)

1. S_H^G contains a perfect matching using edges in H

2. $|S_H^G| + |S_{G\setminus H}^G| = 2\mu(G)$

3. Every vertex $v \in S_{G\setminus H}^G$ has $d_H(v) = \beta/2$ (big simplifying assumption)

4. All edges incident to $S_{G\setminus H}^G$ go to S_H^G

Now consider the ϵ-restricted fractional matching M_f^H in H in which all edges in H from $S_{G\setminus H}^G$ to S_H^G are given value $2/\beta$; (this is ϵ-restricted because β is large, so $2/\beta << \epsilon$). Let us first verify that no vertex has total value more than 1. Every vertex $v \in S_{G\setminus H}^G$ has degree exactly $\beta/2$ in H (property 3 above), so it has total value $(\beta/2)(2/\beta) = 1$. Now consider a vertex $v \in S_H^G$. If there exists no edge $(v, w) \in H$ with $w \in S_{G\setminus H}^G$ then v has total value 0 and we done. Otherwise, by property 3 above we have $d_H(w) = \beta/2$, so by property P1 of an EDCS we have $d_H(v) \leq \beta/2$, so v has total value at most $(\beta/2)(2/\beta) = 1$.

Let us now consider the total value of M_f^H. Since each vertex in $S_{G\setminus H}^G$ has degree $\beta/2$ (property 3) and all edges from $S_{G\setminus H}^G$ go to S_H^G (property 4), we have a total of $\frac{\beta}{2}|S_{G\setminus H}^G|$ edges in $S_{G\setminus H}^G \times S_H^G$, and each has value $2/\beta$, so $\text{VAL}(M_f^H) = |S_{G\setminus H}^G|$.

There are now two cases to consider. The first is that $|S_{G\setminus H}^G| \geq |S_H^G|/2$. By property 2 above we then have

$$\begin{aligned} \text{VAL}(M_f^H) = |S_{G\setminus H}^G| &= \frac{1}{3}|S_{G\setminus H}^G| + \frac{2}{3}|S_{G\setminus H}^G| \\ &\geq \frac{1}{3}(S_{G\setminus H}^G + S_H^G) = \frac{2}{3}\mu(G) \end{aligned}$$

so we have successfully constructed the large ϵ-restricted matching needed by Lemma 31.

The second case is that $|S_{G\setminus H}^G| < |S_H^G|/2$. In this we choose a different ϵ-restricted matching which is simply the integral matching that assigns weight 1 to all the edges in the perfect matching of S_H^G (property 1 above). This matching has size $|S_H^G|/2$ and by property 2 above we have:

$$\frac{1}{2}|S_H^G| = \frac{1}{6}|S_H^G| + \frac{1}{3}|S_H^G| > \frac{1}{3}(S_{G\setminus H}^G + S_H^G) = \frac{2}{3}\mu(G)\ .$$

Note that in the full proof of Lemma 31 we end up mixing the two cases; that is, we end up taking some edges directly from the perfect matching in S_H^G (with weight 1), and some from $S_{G\setminus H}^G \times S_H^G$ (with small weight $< \epsilon$). □

Proof of Theorem 18 By Lemma 31, H contains an ϵ-restricted fractional matching M_f^H with total value at least $\mu(G)(2/3 - \epsilon)$. By Theorem 15, M_f^H must contain an (integral) matching M of size at least $\mu(G)(2/3 - \epsilon)\left(\frac{1}{1+\epsilon}\right) > \mu(G)(2/3 - 2\epsilon)$, as desired.

□

7.7 Maintaining an Edge Degree Constrained Subgraph

In this section we present our algorithm for maintaining the EDCS H in small arboricity graphs (Theorem 21). The algorithm for maintaining H in general graphs (Theorem 19) uses similar ideas but is very technical, so we leave it for Section 7.8.4 at the end of the paper. The first half of the two proofs (of Theorems 21 and 19) are very similar, but the parameters are somewhat different, so for the sake of clarity we completely separate the two proofs and prove each one from scratch. We turn to proving Theorem 21. Let H be a weighted EDCS$(G, \beta, \beta(1 - \lambda))$, and say that at all times the graph G has arboricity $\leq \alpha$. Recall that by the statement of Theorem 21 we have $\beta \geq 4\lambda^{-1}$.

As the graph G changes, we need an algorithm that changes the graph H in a way that preserves the EDCS properties. An adversary will make *external* insertions and deletions in G, but these differ in a crucial way. If the adversary deletes an edge $(u, v) \in G$, and (u, v) was also in H, then we must necessarily delete (u, v) from H. However, if the adversary inserts an edge $(u, v) \in G$, we do not immediately change H. In either case, after the external operation, conditions P1 or P2 may be violated for (u, v) or for other edges (in particular those incident to vertex u or v). The basic idea of our algorithm is very simple: whenever the algorithm detects an edge that violates one of the EDCS properties, it fixes the violating edge through an insertion/deletion. We call these fixing operations *internal* updates: if the edge violates property P1 it removes the edge from H, and if the edge violates property P2 it adds the edge to H. If multiple edges violate the EDCS properties they can be fixed in any order. However, fixing one violation may cause violations to other edges, and the process of performing internal operations cascades. We will analyze our algorithm with a potential function which shows that *on average*, each *external* (i.e. adversarial) update to G only leads to a small number of *internal* updates to H.

Definition 24 *We say that algorithm A for maintaining H is* locally repairing *if:*

1. *Algorithm A only internally deletes edge (u, v) from H if before the deletion $d_H(u)+d_H(v) >$*

β (i.e., the edge violated property P1).

2. *Algorithm A only inserts edge (u, v) into H if before the insertion $d_H(u)+d_H(v) < \beta(1-\lambda)$ (i.e., the edge violated property P2).*

If the algorithm is locally repairing, then *after* an internal insertion/deletion of edge (u, v), (u, v) does not violate the EDCS properties. If edge (u, v) originally violated property P1, then before the update $d_H(u) + d_H(v) > \beta$; removing (u, v) only decreases the edge degree by 2, so after the update $d_H(u) + d_H(v) > \beta - 2 > \beta(1 - \lambda)$ (because $\beta \geq 4\lambda^{-1}$), and so (u, v) satisfies property P2. Similarly, if edge (u, v) violated property P2 then after the update $d_H(u) + d_H(v) < \beta(1 - \lambda) + 2 < \beta$, so (u, v) satisfies property P1.

Lemma 32 *If an algorithm A for maintaining H is locally repairing, then A performs an amortized $O(1/\lambda)$ internal updates to H for each update to G.*

Proof: Consider the potential function

$$\Phi = \sum_{(u,v)\in H} (d_H(u) + d_H(v)) - \beta(1 - \lambda/2) \sum_{v\in V} d_H(v) .$$

We will show that an internal insertion or deletion to H *decreases* Φ by at least $\beta\lambda/2$, while an external deletion to G *increases* Φ by at most 2β, and an external insertion leaves Φ unchanged; together these facts clearly imply the lemma. Given any update to edge (x, y), we let $d^{\mathrm{O}}(x), d^{\mathrm{O}}(y), d^{\mathrm{N}}(x), d^{\mathrm{N}}(y)$ (O for old, N for new) denote the degrees of x and y before and after the edge update. Let $\Delta\Phi$ denote the change to Φ due to the update.

Consider first an internal insertion of edge (x, y). $d_H(x)$ and $d_H(y)$ each increase by 1, and thus $\sum_{v\in V} d_H(v)$ increases by 2. To bound the total change to $\sum_{(u,v)\in H}(d_H(u)+d_H(v))$, observe that the degree of each of the edges incident to x and y increases by 1 so the total increase in edge degree of all these edges is precisely $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. We also added a new edge of edge degree $d^{\mathrm{N}}(x)+d^{\mathrm{N}}(y) = d^{\mathrm{O}}(x)+d^{\mathrm{O}}(y)+2$; thus $\Delta\Phi = 2(d^{\mathrm{O}}(x)+d^{\mathrm{O}}(y))+2-2\beta(1-\lambda/2)$. But because our algorithm is locally repairing we know that for an internal insertion $d^{\mathrm{O}}(x)+d^{\mathrm{O}}(y) < \beta(1-\lambda)$, so $\Delta\Phi < 2\beta(1 - \lambda) + 2 - 2\beta(1 - \lambda/2) = -\beta\lambda + 2 \leq -\beta\lambda/2$, as desired. (The last inequality follows from $\beta \geq 4\lambda^{-1}$).

Consider next an internal deletion of edge (x, y). The total change to $\sum_{v\in V} d_H(v)$ is now -2. The deletion causes the degree of each edge incident to x and y to go down by one, so the

total change to all these edge degrees is $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. We also deleted an edge of edge degree $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. Thus, $\Delta\Phi = -2(d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)) + 2\beta(1 - \lambda/2)$. Since our algorithm is locally repairing we know that for an internal deletion $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y) > \beta$, so $\Delta\Phi \leq -\beta\lambda$, as desired.

Consider next an external deletion of edge (x, y). The same argument as for internal deletions yields $\Delta\Phi = -2(d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)) + 2\beta(1 - \lambda/2)$. Now however, we have no lower bound on $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$ except that it is clearly positive. This yields $\Delta\Phi \leq 2\beta(1 - \lambda/2) < 2\beta$, as desired.

Finally, external insertions do not change H and hence do not alter Φ. $\square$

We now present a locally-repairing algorithm for maintaining H. Note that we maintain a dynamic orientation in G using Theorem 16, so in addition to being able to process updates to G, the algorithm also has to be able to process edge reorientations in G.

Lemma 33 *There exists a locally repairing algorithm A for maintaining H such that the update time of A per update to G is $O(\alpha(1+$ number of internal updates performed by $A))$, and the update time of A per edge reorientation in G is $O(1)$.*

We say that an edge is *violating* if it violates one of constraints P1 or P2 of an EDCS. Before proving this lemma, we design and analyze a data structure for detecting a violating edge incident to a given vertex.

Lemma 34 *Let G be a graph on which we maintain a dynamic orientation in which each vertex owns $O(\alpha)$ edges. There exists a data structure* VO *(violation oracle) that supports the following operations:*

- VO.*find(v) returns, in $O(\alpha)$ time, some violating edge (v, w) incident to v, or* NIL *if none exists.*
- VO.*change-status(v,w) updates the data structure in $O(\alpha)$ time when edge (v, w) is added/removed from H, or inserted/deleted from G.*
- VO.*reorient(v,w) updates the data structure in $O(1)$ time when edge (x, y) is reoriented.*

Proof: Here we give a proof that incurs an extra $O(\log n)$ factor on each operation. We show in Section 7.8.3 how to remove this factor. For each vertex v we keep track of $d_H(v)$ and we maintain two balanced binary search trees. N_v^H contains the degrees $d_H(w)$ of all vertices w for which

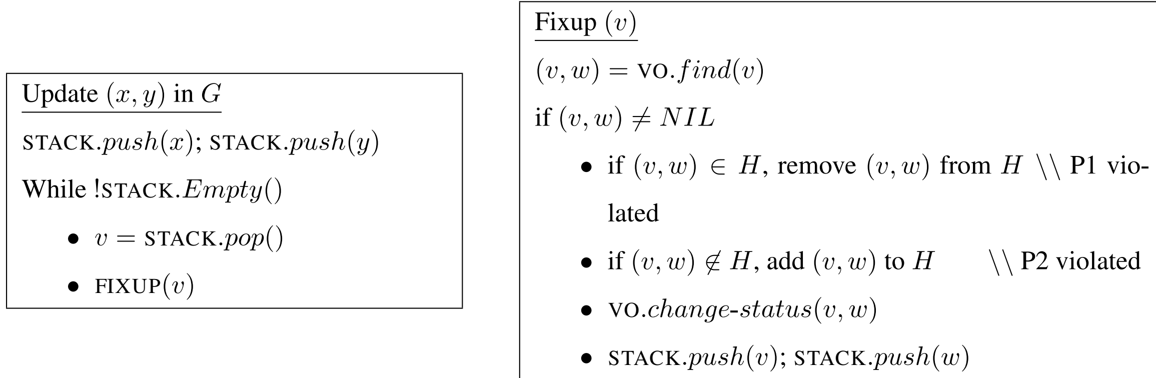

Figure 7.1: How the algorithm of Lemma 33 handles an update to G

$(v, w) \in H$ and (v, w) is *not* owned by v. $N_v^{G \setminus H}$ contains the degrees $d_H(w)$ of all vertices w for which $(v, w) \in G \setminus H$ and is not owned by v.

To implement VO.find(v), we first spend $O(\alpha)$ time scanning the owned edges of $d_H(v)$ for a violating edge. If none is found, we need to look for a violating edge (v, w) that is not owned by v. Note that if (v, w) violates constraint P1 of an EDCS then $(v, w) \in H$ and $d_H(v) + d_H(w) > \beta$; thus, we can find such an edge in $O(\log n)$ time by checking if N_v^H contains any vertices w with $d_H(w) > \beta - d_H(v)$. Similarly, we can find an edge $(v, w) \in G \setminus H$ that violates property P2 of an EDCS by searching in $N_v^{G \setminus H}$ for elements $d_H(w) < \beta(1 - \lambda) - d_H(v)$.

To implement VO.change-status(v, w) when the edge (v, w) changes we might have to insert or remove edge (v, w) from the various data structures $N_v^H, N_v^{G \setminus H}, N_w^H, N_w^{G \setminus H}$; this can clearly be done in $O(\log n)$ time. Also, $d_H(v)$ and $d_H(w)$ might have changed by 1. If $d_H(v)$ changes, for every edge (v, x) that is *owned* by v we have to update N_x^H or $N_x^{G \setminus H}$ depending on whether (v, x) is in G or $G \setminus H$. But v only owns $O(\alpha)$ edges, so this requires $O(\alpha \log n)$ time. We can similarly handle the change to $d_H(w)$ in $O(\alpha \log n)$ time.

Finally, to implement VO.reorient(v, w), if (v, w) was previously owned by v we have to move $d_H(v)$ out of N_w^H or $N_w^{G \setminus H}$ and move $d_H(w)$ into N_v^H or $N_v^{G \setminus H}$. This requires $O(\log n)$ time. We can save an $O(\log n)$ factor in this proof by replacing the binary search tree with a simple array-based structure (Section 7.8.3). □

Proof of Lemma 33 Note that all we are looking for is a procedure for quickly detecting and fixing a violating edge; Lemma 32 already guarantees that any such procedure will terminate quickly.

The algorithm at all times maintains the violation oracle VO of Lemma 34. To handle a reori-

entation of edge (x, y) the algorithm simply calls VO.reorient(x,y), which takes $O(1)$ time. The procedure for handling an update to G is defined in Figure 7.1. We maintain a stack (STACK) which will contain vertices that might possibly have an incident violating edge. The key observation is that an edge (x, y) can only become violating if one of $d_H(x)$ or $d_H(y)$ changes, so we will catch all violating edges if every time we add/remove some edge (x, y) to/from H we put x and y onto STACK. The while loops guarantees that when we terminate no violating edges are left.

We now need to show that the algorithm spends $O(\alpha)$ time per internal update. Each iteration of the while loop requires $O(\alpha)$ time to run VO.find and VO.change-status. Other than the two initial vertices v,w put on STACK by the update of edge (v, w) in G, a vertex x is only put on STACK when the algorithm performs an internal update to edge (x, y), so the time to process x and y can be charged to the internal update of (x, y). The algorithm thus requires $O(\alpha)$ time for the initial update to G, and $O(\alpha)$ time for each subsequent internal update. □

Proof of Theorem 21 First we show how to maintain an unweighted EDCS. By Theorem 16 each update to G leads to $O(\alpha + \log n)$ edge reorientations, and requires $O(\alpha(\alpha + \log n))$ time to compute the correct reorientations. By Lemma 32 each update to G leads to amortized $O(\lambda^{-1})$ internal updates, yielding the $O(\lambda^{-1})$ bound for amortized update ratio. Thus, by Lemma 33, an update to G can be processed in time $O(\alpha(\alpha + \log n + \lambda^{-1}))$.

To maintain a weighted EDCS, let G^β be the graph G where every edge has multiplicity β; maintaining a weighted EDCS on G is equivalent to maintaining an unweighted one on G^β. Each update to G can cause up to β updates to G^β, multiplying the amortized update ratio and update time by β. However, it is not hard to show that the terms corresponding to maintaining an orientation are not multiplied by β, since an orientation of the original graph G suffices. Thus the total update time is $O(\alpha(\alpha + \log n + \beta\lambda^{-1}))$. □

7.8 Appendix of the More Technical Proofs

7.8.1 Full proof of Theorem 20

We start by defining the following accounting for the edges of H. For any vertex v, if $d_H(v) \leq \frac{\beta}{2}(1 - \sqrt{\lambda})$ we set $a_v(v, w) = 0$ for all edges (v, w). If $d_H(v) \geq \frac{\beta}{2}(1 + \sqrt{\lambda})$, we set $a_v(v, w) = 1$ for all edges (v, w). Else, if $d_H(v) \in (\frac{\beta}{2}(1 - \sqrt{\lambda}), \frac{\beta}{2}(1 + \sqrt{\lambda}))$ for all edges (v, w) we set

$a_v(v,w) = 1/2 + \frac{d_H(v)-\beta/2}{\beta\sqrt{\lambda}}$. (It is easy to check that this is always between 0 and 1). To check that for any edge in H we always have $a_v(v,w) + a_w(v,w) \le 1$, let (v,w) be some edge in H, WLOG let $d_H(v) \ge d_H(w)$, and recall that by property P1 of an EDCS we have $d_H(v) + d_H(w) \le \beta$. If $d_H(w) \le (\beta/2)(1-\sqrt{\lambda})$ then $a_w(v,w) = 0$, so since $a_v(v,w) \le 1$, we are done. If $d_H(v) \ge (\beta/2)(1+\sqrt{\lambda})$ then $d_H(w) \le (\beta/2)(1-\sqrt{\lambda})$, so again we are done. Finally, if both $d_H(v)$ and $d_H(w)$ are in the interval $[(\beta/2)(1-\sqrt{\lambda}), (\beta/2)(1+\sqrt{\lambda})]$, then

$$\begin{aligned}
&a_v(v,w) + a_w(v,w) \\
&= (1/2 + \frac{d_H(v)-\beta/2}{\beta\sqrt{\lambda}}) + (1/2 + \frac{d_H(w)-\beta/2}{\beta\sqrt{\lambda}}) \\
&= 1 + \frac{d_H(v) + d_H(w) - \beta}{\beta\sqrt{\lambda}} \\
&\le 1
\end{aligned}$$

where the last inequality follows from $d_H(v) + d_H(w) \le \beta$.

Recall that for every edge $(u,v) \in H$, M_f^H assigns $\text{VAL}(u,v) = 1/\max\{d_H(u), d_H(v)\}$. Note that given a vertex v, for every edge (v,w) we have $d_H(w) \le \beta - d_H(v)$ (property P1 of an EDCS), so $\text{VAL}(v,w) \ge 1/\max\{d_H(v), \beta - d_H(v)\}$. Thus, we have

$$\begin{aligned}
\rho(v) &= \sum_{(v,w)} \text{VAL}(v,w) a_v(v,w) \\
&\ge \frac{d_H(v)}{\max\{d_H(v), \beta - d_H(v)\}} \cdot \min\{1, 1/2 + \frac{d_H(v)-\beta/2}{\beta\sqrt{\lambda}}\}
\end{aligned} \tag{7.2}$$

Lemma 35 *The function $\rho(v)$ satisfies the following properties:*

1. *If $d_H(v) \ge (\beta/2)(1-\lambda)$ then $\rho(v) \ge (1/2)(1-3\sqrt{\lambda})$.*

2. *If $d_H(v) + d_H(w) \ge \beta(1-\lambda)$ then $\rho(v) + \rho(w) \ge 1 - 5\sqrt{\lambda}$.*

Proof: To verify the first property, we simply plug in $d_H(v) = (\beta/2)(1-\lambda)$ into Equation 7.2, obtaining

$$\begin{aligned}
\rho(v) &\ge (1/2)((1-\lambda)/(1+\lambda))(1-\sqrt{\lambda}) \\
&> (1/2)(1-2\lambda)(1-\sqrt{\lambda}) \\
&> (1/2)(1-3\sqrt{\lambda}) .
\end{aligned}$$

To verify the second property, let us say that $d_H(v) \geq d_H(w)$. If we had $d_H(v) \geq (\beta/2)(1+\sqrt{\lambda})$ then it is easy to see that $\rho(v) = 1$ because $d_H(v)/\max\{d_H(v), \beta - d_H(v)\} = d_H(v)/d_H(v) = 1$, and the min term in Equation 7.2 also comes out to 1. Thus the only case left to consider is when $d_H(v) \leq (\beta/2)(1+\sqrt{\lambda})$, and so by property P2 of an EDCS, $d_H(w) \geq \beta(1-\lambda) - d_H(v) > (\beta/2)(1-2\sqrt{\lambda})$. Note that in this case

$$\begin{aligned}
\frac{d_H(w)}{\max\{d_H(w), \beta - d_H(w)\}} &\geq \frac{(\beta/2)(1-2\sqrt{\lambda})}{(\beta/2)(1+2\sqrt{\lambda})} \\
&= \frac{1-2\sqrt{\lambda}}{1+2\sqrt{\lambda}} \\
&\geq 1 - 4\sqrt{\lambda}\,.
\end{aligned}$$

Similarly, since $d_H(v) \geq d_H(w)$, we have

$$\frac{dh(v)}{\max\{d_H(v), \beta - d_H(v)\}} \geq 1 - 4\sqrt{\lambda}\,.$$

Thus, recalling that $d_H(v) + d_H(w) \geq \beta(1-\lambda)$, we have:

$$\begin{aligned}
&pr(v) + pr(w) \\
&\geq \left(1-4\sqrt{\lambda}\right)\left(\frac{1}{2} + \frac{d_H(v) - \beta/2}{\beta\sqrt{\lambda}} + \frac{1}{2} + \frac{d_H(w) - \beta/2}{\beta\sqrt{\lambda}}\right) \\
&= \left(1-4\sqrt{\lambda}\right)\left(1 + \frac{d_H(v) + d_H(w) - \beta}{\beta\sqrt{\lambda}}\right) \\
&\geq \left(1-4\sqrt{\lambda}\right)\left(1 + \frac{\beta(1-\lambda) - \beta}{\beta\sqrt{\lambda}}\right) \\
&= \left(1-4\sqrt{\lambda}\right)\left(1-\sqrt{\lambda}\right) \\
&\geq \left(1-5\sqrt{\lambda}\right)\,.
\end{aligned} \tag{7.3}$$

□

The proof of Theorem 20 is almost complete. As argued in the proof sketch in Section 7.6, it is easy to see from [Scheinerman and Ullman, 2011] that any graph G has a maximum fractional matching whose support consists of disjoint odd cycles and disjoint individual edges: the individual edges have value 1, the edges on the odd cycles all have value $1/2$, and all other edges have value 0. Let M_f^G be such a maximum matching in G. We will prove that $\text{VAL}(M_f^H) \geq (1-5\sqrt{\lambda})\text{VAL}(M_f^G)$. Note that M_f^G gets a value of 1 from each individual edge not on a cycle, and a value of $d/2$ for each odd cycle of length d. We will now show that M_f^H always gains at least a $(1-5\sqrt{\lambda})$

fraction of what M_f^G gains from these edges/cycles. Since we set $\lambda = \epsilon^2/25$, this proves that $\text{VAL}(M_f^H) \geq \text{VAL}(M_f^G)(1-\epsilon)$.

For each isolated edge (v, w) in M_f^G we have by property P2 of an EDCS that $d_H(v)+d_H(w) \geq \beta(1-\lambda)$, so by property 2 of Lemma 35 we get that $\rho(v)+\rho(w) \geq (1-5\sqrt{\lambda})$; in other words, M_f^H gains at least a $(1-5\sqrt{\lambda})$ fraction of what M_f^G gains on that edge. For a cycle of odd length d, let v be the vertex on the cycle with maximum $d_H(v)$. It is not hard to see that $d_H(v) \geq \frac{\beta}{2}(1-\lambda)$, since otherwise by property P2 of an EDCS the neighbors of v on the cycle would have degree higher than $d_H(v)$. Thus, by Property 1 of Lemma 35 we have that $\rho(v) \geq (1/2)(1-3\sqrt{\lambda}) > (1/2)(1-5\sqrt{\lambda})$. The remainder of the cycle clearly contains $(d-1)/2$ disjoint edges in G, and by the argument above, for each of these edges M_f^H gains a profit of at least $(1-5\sqrt{\lambda})$. Thus, the total profit gained by M_f^H from the cycle is at least $(1/2)(1-5\sqrt{\lambda}) + ((d-1)/2)(1-5\sqrt{\lambda}) = (d/2)(1-5\sqrt{\lambda})$; again at least a $(1-5\sqrt{\lambda})$ fraction of what M_f^G gains.

7.8.2 Proof of Lemma 31

Recall that we are dealing with an (unweighted) EDCS$(G, \beta, \beta(1-\lambda))$. We will explicitly construct a ϵ-restricted fractional matching M_f^H of large size. We start by defining a function $\phi(x)$ for $x \in [0, \beta]$; loosely speaking, $\phi(d_H(u))$ will end up corresponding to the profit gained by vertex u in the fractional matching M_f^H.

$$\phi(x) = \min\left\{1, \frac{x}{2(\beta - x)}\right\} . \tag{7.4}$$

Lemma 36 *If $a, b \in [0,1]$ and $a + b \geq \beta(1-\zeta)$ for some $\zeta \geq 0$, then $\phi(a) + \phi(a) \geq 1 - 5\zeta$.*

Proof: We first show that if $a + b \geq \beta$ then $\phi(a) + \phi(b) \geq 1$. The claim is trivially true if $\phi(a) \geq 1$ or $\phi(b) \geq 1$, so we can assume that $\phi(a) < 1$ and $\phi(b) < 1$. In this case, we have

$$\begin{aligned}\phi(a) + \phi(b) &= \frac{a}{2(\beta - a)} + \frac{b}{2(\beta - b)} \\ &\geq \frac{a}{2b} + \frac{b}{2a} = \frac{a^2+b^2}{2ab} = \frac{(a-b)^2}{2ab} + 1 \geq 1\end{aligned}$$

Now, let $\phi'(x) = \frac{\mathrm{d}}{\mathrm{dx}}\phi(x)$. To complete the lemma, it is sufficient to show that we always have $\phi'(x) \leq 5/\beta$. To prove this inequality, first note that if $x \geq 2\beta/3$ then $\phi(x) = 1$. Thus, if $x > 2\beta/3$ then $\phi'(x) = 0$. Now, if $x \leq 2\beta/3$ then $\phi'(x) = \frac{\mathrm{d}}{\mathrm{dx}}\frac{x}{2(\beta-x)} = \frac{\beta}{2(\beta-x)^2}$. This function

clearly increases with x, and is maximized precisely at $x = 2\beta/3$, in which case $\phi'(x) = \frac{9}{2\beta} < \frac{5}{\beta}$, as desired. $\square$

Lemma 37 *Given any* EDCS$(G, \beta, \beta(1-\lambda))$*, H, we can find two disjoint sets of vertices X and Y that satisfy the following properties. (Recall the function ϕ defined in Equation 7.4.)*

1. $|X| + |Y| = 2\mu(G)$.
2. *There is a perfect matching in Y using edges in H.*
3. *Letting $\sigma = |Y|/2 + \sum_{x \in X} \phi(x)$, we have $\sigma \geq \mu(G)(1-5\lambda)$.*
4. *All edges in H have at least one endpoint in Y.*

Proof: Let M^G be some maximum integral matching in G. Some of the edges in M^G are in H, while others are in $G \setminus H$. Let X_0 contain all vertices incident to edges in $M^G \cap (G \setminus H)$, and let Y_0 contain all vertices incident to edges in $M^G \cap H$. We now show that X_0 and Y_0 satisfy the first three properties of the lemma. Property 1 is satisfied because $X_0 \cup Y_0$ consists of all matched vertices in M^G. Property 2 is satisfied by definition of Y_0. To see that property 3 is satisfied, we show that the vertices in $X_0 \cup Y_0$ contribute an average of at least $(1-5\lambda)/2$ to σ. The vertices in Y_0 each contribute exactly $1/2$. Now, X_0 consists of $|X_0|/2$ disjoint edges in $G \setminus H$, and by property P2 of an EDCS, for each such edge (x, x') we have $d_H(x) + d_H(x') \geq \beta(1-\lambda)$, so by Lemma 36 we have $\phi(x) + \phi(x') \geq 1 - 5\lambda$, and between them x and x' contribute an average of at least $(1-5\lambda)/2$ to σ, as desired.

However, the sets X_0 and Y_0 might not satisfy property 4 of Lemma 37. We first show how to transform X_0, Y_0 to sets X_1, Y_1 such that the first three properties are still satisfied, and there are no edges in H between X_1 and $V \setminus (X_1 \cup Y_1)$; at this stage, however, there will possibly be edges in H between vertices in X_1. To construct X_1, Y_1, we start with $X = X_0$ and $Y = Y_0$, and present a transformation that terminates with $X = X_1$ and $Y = Y_1$. Recall that X_0 has a perfect matching using edges $G \setminus H$. The set X will maintain this property throughout the transformation, and each vertex $x \in X$ always has a unique *mate* x'. The construction does the following: as long as there exists an edge (x, z) in H where $x \in X$ and $z \in V \setminus (X \cup Y)$, let x' be the mate of x; we then remove x and x' from X and add x and z to Y. Property 1 is maintained because we removed two vertices from $|X|$ and added two to $|Y|$. Property 2 is maintained because the vertices we added to

Y were connected by an edge in H. Property 3 is maintained because X clearly still has a perfect matching in $G \setminus H$, and for every pair (x, x') the average contribution to σ among x and x' is still at least $(1-5\lambda)/2$, as above. We continue this process while an edge (x, y) in H from X to $V-X-Y$ exists; the process terminates because each time we are removing two vertices from X and adding two to Y. We thus end with two sets X_1, Y_1 such that the first three properties of the lemma are satisfied, and there are no edges between X_1 and $V - X_1 - Y_1$.

We now set $X = X_1$ and $Y = Y_1$ and show how to transform X and Y into two sets that satisfy all four properties of the lemma. Recall that X_1 still contains a perfect matching using edges in $G \setminus H$: denote this matching by M_X^G. Our final set X, however, will not guarantee such a perfect matching. Let M_X^H be a maximal matching in X using edges in H. Consider the edge set $E_X^* = M_X^G \cup M_X^H$. Now, E_X^* is a degree 2 graph so it can be decomposed into vertex-disjoint paths and cycles. Since both M_X^G and M_X^H are matchings, the paths and cycles alternate between edges in M_X^G and edges in M_X^H; in particular, they alternate between edges in $G \setminus H$ and edges in H. Each cycle contains an even number of vertices because otherwise it could not alternate. Because M_X^G is a perfect matching, every vertex in X_1 is in a path or cycle, and each path starts and ends with edges in M_X^G. In particular, each path contains an even number of vertices and is of the form $x_1, x_1', x_2, x_2', ..., x_k, x_k'$, where for every $i \leq k$ there is an edge (x_i, x_i') in M_X^G, and for every $i < k$ there is an edge (x_i', x_{i+1}) in M_X^H.

We now perform the following transformation. For each cycle C in E_X^*, we simply remove all the vertices in C from X and add them to Y. Property 1 is preserved because we are moving vertices from one set to the other. Property 2 is preserved because C contains a perfect matching in H since every second edge in C is in M_X^H. We will argue that property 3 is preserved momentarily. We now continue describing the transformation. For each path P in E_X^*, if P consists of a single edge in M_X^G, we do nothing. Else, if P is longer, then P is of the form $x_1, x_1', x_2, x_2', ..., x_k, x_k'$ indicated above. The transformation moves all vertices except x_1 and x_k' (the ends of the path) from X to Y. Property 1 is clearly preserved. Property 2 is preserved because the vertices moved had a perfect matching among them in M_X^H and so in H (the perfect matching that matches x_i' to x_{i+1} for $i < k$).

We must now check that property 3 is preserved by this transformation. As before, this involves showing that after the transformation, the average contribution of a vertex in $X \cup Y$ to σ is at least

$(1 - 5\lambda)/2$. (Because every vertex in X is incident to an edge in E_X^*, each vertex is accounted for in the transformation.) Now, all vertices that were in Y_1 remain in Y, so their average contribution remains at $1/2$. We thus need to show that the average contribution to σ among vertices in X_1 remains at least $(1 - 5\lambda)/2$ *after* the transformation. We will in particular show that given any cycle C or path P in E_X^*, the average contribution of vertices in that path/cycle is at least $(1 - 5\lambda)/2$ after the transformation. For any cycle C, all the vertices are moved to Y and thus each contribute exactly $1/2$ to σ. For any path $P = (x, x')$ that consists of a single edge in M_X^G, we still have that by property P2 of an EDCS, $d_H(x_i) + d_H(x_i') \geq \beta(1 - \lambda)$, so by Lemma 36, $\phi(x) + \phi(x') \geq 1 - 5\lambda$. Finally, consider a path $P = (x_1, x_1', ..., x_k, x_k')$. We have that for all i, $(x_i, x_i') \in G \setminus H$, so $d_H(x_i) + d_H(x_i') \geq \beta(1 - \lambda)$, so $\sum_{x \in P} d_H(x) \geq k\beta(1 - \lambda)$. On the other hand, since for all $i < k$ there is an edge (x_i', x_{i+1}) in M_X^H and so in H, we have by property P1 of an EDCS that for all $i < k$, $d_H(x_i') + d_H(x_{i+1}) \leq \beta$. Thus

$$
\begin{aligned}
d_H(x_1) + d_H(x_k') &= \sum_{x \in P} d_H(x) - \sum_{i<k} (d_H(x_i') + d_H(x_{i+1})) \\
&\geq k\beta(1 - \lambda) - (k - 1)\beta \\
&= \beta - k\beta\lambda \, .
\end{aligned}
\tag{7.5}
$$

Thus, by Lemma 36, we have $\phi(x_1) + \phi(x_k') \geq 1 - 5k\lambda$. We are now ready to bound the average contribution to σ among vertices in P *after* the transformation. Since we have the endpoints contributing a total of at least $1 - 5k\lambda$ and $2k - 2$ vertices that move to Y each contributing $1/2$, the average contribution of vertices on the path to σ is at least $((1 - 5k\lambda) + (1/2)(2k - 2))/(2k) = (1 - 5\lambda)/2$.

Our transformation thus preserves properties 1, 2, and 3. It now remains to verify that the resulting sets X and Y satisfy property 4. To see this, note that we took a maximal matching M_X^H of the set X_1 among edges in H, and moved all the matched vertices in M_X^H to Y. Thus all the vertices that remain in X are free in M_X^H, and so by definition of a maximal matching, there are no edges in H between vertices in X after the transformation. There are also no edges in H between X and $V \setminus (X \cup Y)$ because there are no such edges originally when $X = X_1$, and our transformation only moved edges from X to Y, which cannot create new edges between X and $V \setminus (X \cup Y)$. □

We now want to construct (for the proof) a ϵ-restricted fractional matching M_f^H using the edges in H such that $\textsc{val}(M_f^H) \geq (2/3-\epsilon)\mu(G)$. We start by finding two sets $|X|$ and $|Y|$ that satisfy the properties of Lemma 37. Now, by property 2 of Lemma 37, $|Y|$ contains a perfect matching M_Y^H using edges in H. Let Y^- be a subset of Y obtained by randomly sampling exactly half the edges of M_Y^H and adding their endpoints to Y^-. Let $Y^* = Y \setminus Y^-$, and observe that $|Y^-| = |Y^*| = |Y|/2$.

Let H^* be the subgraph of H (not of G) induced by $X \cup Y^*$. We define a fractional matching M_f^{H*} on the edges of H^* in which all edges have value at most ϵ. We will then let our final fractional matching M_f^H be the fractional matching M_f^{H*} joined with the perfect matching in H of Y^- (so M_f^H assigns value 1 to the edges in this perfect matching). M_f^H is, by definition, a ϵ-restricted fractional matching.

We now give the details for the construction of M_f^{H*}. Let $V^* = X \cup Y^*$ be the vertices of H^*, and let E^* be its edges. For any vertex $v \in V^*$, define $d_H^*(v)$ to be the degree of v in H^*. Recall that by property 4 of Lemma 37, if $x \in X$ then all the edges of H incident to x go to Y (but some might go to Y^-); thus, for $x \in X$, we clearly have $E[d_H^*(x)] = d_H(x)/2$.

We now define M_f^{H*} as follows. For every $x \in X$, we arbitrarily order the edges of H incident to x, and then we assign a value of $\min\left\{\epsilon, \frac{1}{\beta - d_H(x)}\right\}$ to the edges one by one, stopping when either $\textsc{val}(x)$ reaches 1 or there are no more edges in H incident to x, whichever comes first (in the case that $\textsc{val}(x)$ reaches 1 the last edge might have value less than $\min\left\{\epsilon, \frac{1}{\beta - d_H(x)}\right\}$). We now verify that M_f^{H*} is a valid fractional matching in that all vertices have value at most 1. This is clearly true of vertices $x \in X$ by construction. For a vertex $y \in Y^*$, it suffices to show that each edge incident to y receives a value of at most $1/d_H(y) \leq 1/d_H^*(y)$. To see this, first note that the only edges to which M_f^{H*} assigns non-zero values are in $X \times Y^*$. Any such edge (x, y) receives value at most $1/(\beta - d_H(x))$, but since (x, y) is in M_f^{H*} and so in H, we have by property $P1$ of an EDCS that $d_H(y) \leq \beta - d_H(x)$, and so $1/(\beta - d_H(x)) \leq 1/d_H(y)$, as desired.

By construction, for any $x \in X$, we have that in M_f^{H*}

$$\textsc{val}(x) = \min\left\{1, d_H^*(x) \cdot \min\left\{\epsilon, \frac{1}{\beta - d_H(x)}\right\}\right\}. \tag{7.6}$$

Consider the simple accounting on edges in E^* where for every edge $(x, y) \in X \times Y^*$ we set $a_x(x, y) = 1$ and $a_y(x, y) = 0$, while for other edges (y, y') we effectively ignore the edge by setting $a_y(y, y') = a_{y'}(y, y') = 0$ In this accounting, we clearly have $\rho(x) = \textsc{val}(x)$ for $x \in X$

and $\rho(y) = 0$ for $y \in Y^*$. By Observation 1 we can lower bound M_f^{H*} by summing over all $\rho(x)$ for $x \in X$. To this end, we prove the lemma below; recall that $E[d_H^*(x)] = d_H(x)/2$, that $\beta \geq 32\lambda^{-3} >> \epsilon^{-1}$, and the definition of $\phi(x)$ from Equation 7.4.

Lemma 38 *For any $x \in X$, $E[\rho(x)] \geq (1-\lambda)\phi(d_H(x))$.*

Proof: The proof is simply algebraic manipulation combined with the Chernoff bound. First consider the case in which $d_H(x) \leq \beta/2$. In this case $\frac{1}{\beta - d_H(x)} \leq 2/\beta < \epsilon$, and $d_H^*(x) \leq d_H(x) \leq \beta - d_H(x)$, so $\rho(x) = \frac{d_H^*(x)}{\beta - d_H(x)}$. Thus since $E[d_H^*(x)] = d_H(x)/2$, we have by definition of Φ that:

$$E[\rho(x)] = \frac{E[d_H^*(x)]}{\beta - d_H(x)} = \frac{d_H(x)/2}{\beta - d_H(x)} = \Phi(d_H(x))$$

Now consider the case in which $d_H(x) > \beta/2$. Then,

$$E[d_H^*(x)] = \frac{d_H(x)}{2} > \frac{\beta}{4} \geq 8\lambda^{-3} .$$

Thus by the Chernoff bound

$$\begin{aligned} Pr[d_H^*(x) &< (1-\frac{\lambda}{2})(\frac{d_H(x)}{2})] \\ &< e^{-E[d_H^*(x)](\frac{\lambda}{2})^2/2} \\ &\leq e^{-\lambda^{-1}} < \frac{\lambda}{2} , \end{aligned} \tag{7.7}$$

where the last inequality follows from simple calculus and the fact that $0 \leq \lambda \leq 1$. Now, we start by considering the fringe case where $d_H(x) \geq \beta - \epsilon^{-1}$, and so $\min\left\{\epsilon, \frac{1}{\beta - d_H(x)}\right\} = \epsilon$. By Equation 7.7 with probability at least $(1 - \frac{\lambda}{2})$, we have that $d_H^*(x) \geq \frac{(\beta - \frac{1}{\epsilon})(1-\frac{\lambda}{2})}{2} >> \epsilon^{-1}$. So with probability at least $(1-\frac{\lambda}{2})$ we have $d_H^*(x)\epsilon > 1$, so $E[\rho(x)] \geq (1-\frac{\lambda}{2}) \geq (1-\frac{\lambda}{2})\phi(x)$.

We are thus left with the case where $d_H(x) > \beta/2$, and $\rho(x) = \min\left\{1, \frac{d_H^*(x)}{\beta - d_H(x)}\right\}$. Again by Equation 7.7 we have that with probability at least $(1-\lambda/2)$,

$$\frac{d_H^*(x)}{\beta - d_H(x)} \geq \frac{d_H(x)(1-\lambda/2)}{2(\beta - d_H(x))} \geq \left(1 - \frac{\lambda}{2}\right)\phi(d_H(x)) .$$

In other words, with probability at least $(1-\lambda/2)$ we have $\rho(x) \geq (1-\frac{\lambda}{2})\phi(d_H(x))$, so $E[\rho(x)] \geq (1-\frac{\lambda}{2})^2\phi(d_H(x)) > (1-\lambda)\phi(d_H(x))$, as desired. □

We have almost completed the proof of Lemma 31. By Lemma 38 and Observation 1, we are able to lower bound $\text{VAL}(M_f^{H*})$. In particular,

$$\text{VAL}(M_f^{H*}) \geq \sum_{x \in X} \rho(x) \geq (1-\lambda) \sum_{x \in X} \phi(d_H(x)) .$$

Recall that we constructed M_f^H by taking the fractional value M_f^{H*} and adding in the half of the edges from Y that we had removed (i.e. the edges in Y^-). There are in total $|Y^-|/2 = |Y|/4$ such edges so

$$\textsc{val}(M_f^H) \geq (1-\lambda)\sum_{x\in X}\phi(d_H(x)) + \frac{|Y|}{4}\,.$$

We now lower bound this quantity using property 3 of Lemma 37.

$$\begin{aligned}
\textsc{val}(M_f^H) &\geq (1-\lambda)\sum_{x\in X}\phi(d_H(x)) + \frac{|Y|}{4} \\
&= (1-\lambda)\sum_{x\in X}\phi(d_H(x)) + \frac{|Y|}{2} - \frac{|Y|}{4} \\
&\geq (1-\lambda)\mu(G)(1-5\lambda) - \frac{|Y|}{4} \\
&\geq (1-6\lambda)\mu(G) - \frac{|Y|}{4}\,.
\end{aligned} \tag{7.8}$$

To complete the proof, recall that Y contains a perfect matching in H of $|Y|/2$ edges, so if $|Y| \geq 4\mu(G)/3$ then there already exists a matching in H of size $2\mu(G)/3$, so the main lemma we are trying to prove (Lemma 31) is trivially true. We can thus assume that $|Y| < 4\mu(G)/3$, in which case Equation 7.8 yields that

$$\begin{aligned}
\textsc{val}(M_f^H) &\geq (1-6\lambda)\mu(G) - \frac{|Y|}{4} \\
&> (1-6\lambda)\mu(G) - \frac{\mu(G)}{3} \\
&= \left(\frac{2}{3} - 6\lambda\right)\mu(G)\,.
\end{aligned}$$

This completes the proof because in Lemma 31 we set $\lambda = \epsilon/6$.

7.8.3 A Violation Oracle: Proof of Lemma 34

In our earlier proof of Lemma 34 we achieved all the desired bounds to within a $\log n$ factor. This $\log n$ factor came from the fact that we used balanced binary search trees for $N_v^{G\setminus H}$ and N_v^H. To remove the $\log n$ factor, we maintain exactly the same oracle as in the earlier proof, except that we change the data structures $N_v^{G\setminus H}$ and N_v^H to allow constant time per operation. The new data structures are extremely simple. Recall that a data structure for vertex v will include all its neighbors w. To avoid recourse to hash tables (and the resulting randomized algorithm), all the different data structures for all vertices v will use pointers to the global list of vertices V.

Lemma 39 *There exists a data structure which we call a* High Threshold Table *(*HTT*), which contains some subset of V of elements w each with an associated* KEY*(w). The* HTT *also has a threshold parameter* HIGH*, and supports the following operations in constant time.*

1. *Insert or delete some element w (a key change can be implemented as a deletion followed by an insertion).*

2. *Return some element w with* KEY$(w) >$ HIGH *(or* NIL *if none exists).*

3. *Increase or Decrease* HIGH *by* 1.

There also exists an analogous data structure Low Threshold Table *(*LTT*) that is identical except it stores a threshold* LOW*, and operation 2 requires finding an element w with* KEY$(w) <$ LOW.

Proof: We group all elements w into buckets according to KEY(w), so bucket B_k contains all w for which KEY$(w) = k$. We can store the buckets as doubly linked lists, with pointers from element w in the list to the vertex w in V, and vice versa. We then keep an *unordered* list L of all indices k such that $k >$ HIGH and B_k is non-empty. We maintain a pointer from each bucket B_k to its position in L (if $k \in L$).

Operation 1 above can be implemented as follows: to insert some vertex w into N_v^H, the algorithm simply puts w in bucket $B_{\text{KEY}(w)}$. If w is the only vertex in $B_{\text{KEY}(w)}$ and KEY$(w) >$ HIGH then it also adds index KEY(w) to L. To delete some vertex w from the HTT, the algorithm deletes w from $B_{\text{KEY}(w)}$, and if $B_{\text{KEY}(w)}$ is now empty it deletes index KEY(w) from L. For operation 2, the algorithm finds the first element k of L, and then picks an arbitrary element w from B_k; by definition of L this will guarantee that KEY$(w) >$ HIGH. Finally, for operation 3, if HIGH moves by 1, let k be the original HIGH. If HIGH decreased by 1, the algorithm simply adds k to L if bucket B_k is non-empty. If HIGH increased by 1, the algorithm removes $(k + 1)$ from L if $(k + 1)$ was in L. All three operations can thus be done in constant time. □

Going back to the proof of Lemma 34, for N_v^H we can use an HTT that includes the same elements w as in the earlier proof in Section 7.7 (where we used a balanced binary search tree for N_v^H). We set KEY$(w) = d_H(w)$ and HIGH $= \beta - d_H(v)$. The implementation of the operations VO.find(v) and VO.reorient(v) are then exactly the same as in the earlier proof. To implement VO.change-status(v,w), we might have to increase/decrease HIGH by 1 if the update to edge (v, w) increased/decreased $d_H(v)$. Similarly, for $N_v^{G \setminus H}$ we use an LTT with LOW $= \beta(1 - \lambda) - d_H(v)$.

7.8.4 Maintaining an Edge Degree Constrained Subgraph in General Graphs

In this section we prove Theorem 19. The first half of the proof is nearly identical to that in Section 7.7 modulo some parameter changes, but for the sake of clarity we repeat the whole proof with the new parameters. Recall that updates to H (the EDCS) can be external or internal. An external update is an update made by the dynamic adversary which inserts or deletes an edge in G. This external update can lead certain edges to violate the EDCS properties, so the algorithm that maintains H can also make internal updates which add or remove edges from H to maintains these properties. Recall that an external deletion of edge (u, v) also removes the edge from H, while an external insertion of (u, v) only affects G, not H (though the external insertion may be followed up by an internal insertion which adds the edge to H).

Intuitively, the algorithm only needs to perform an internal update on an edge if that edge violates one of the EDCS properties. In Section 7.7 we argued that such an algorithm only performs a small number of internal updates (Lemma 32). The problem is that in general graphs there could be many edges and finding a violating edge is difficult. If the algorithm examines a large number of edges that are not violating, we need to somehow guarantee that progress is still being made. To this end we observe that if an algorithm comes across an edge that is *close* to violating one of the properties, it makes sense to fix it on the spot. This motivates a generalization of the definition of locally repairing (Definition 24) in Section 7.7. Recall that H is an (unweighted) EDCS$(G, \beta, \beta(1 - \lambda))$, and that by the statement of Theorem 19, $\beta \geq 36\lambda^{-1}$.

Definition 25 *We say that algorithm A for maintaining* H *is* locally balancing *if:*

1. *Algorithm A only internally deletes edge* (u, v) *from* H *if before the deletion* $d_H(u)+d_H(v) > \beta(1 - 4\lambda/9)$ *(i.e. the edge was close to violating property P1).*
2. *Algorithm A only internally inserts edge* (u, v) *into* H *if before the insertion* $d_H(u)+d_H(v) < \beta(1 - 5\lambda/9)$ *(i.e., the edge was close to violating property P2).*

Let us say that an edge is unbalanced if it satisfies one of the inequalities above, i.e. if it is a candidate for being added or removed from H by a locally balancing algorithm. Note that *after* the locally balancing algorithm updates the edge, it is no longer unbalanced. If before the update edge (u, v) was in H and $d_H(u) + d_H(v) > \beta(1 - 4\lambda/9)$, then since removing the edge only decreases

its edge degree by 2, after the update $(u, v) \in G \setminus H$ and $d_H(u) + d_H(v) > \beta(1 - 4\lambda/9) - 2 \geq \beta(1 - 5\lambda/9)$, so the edge is no longer unbalanced (The last inequality follows from $\beta \geq 36\lambda^{-1}$). Similarly, if before the update (u, v) was in $G \setminus H$ and $d_H(u) + d_H(v) < \beta(1 - 5\lambda/9)$ then after the update $(u, v) \in H$ and $d_H(u) + d_H(v) \leq \beta(1 - 5\lambda/9) + 2 \leq \beta(1 - 4\lambda/9)$.

Lemma 40 *If an algorithm A for maintaining H is locally balancing, then A performs an amortized $O(1/\lambda)$ internal updates to H for each update to G.*

Proof: The proof is very similar to that of Lemma 32. We use the same potential function

$$\Phi = \sum_{(u,v)\in H} (d_H(u) + d_H(v)) - \beta(1 - \lambda/2) \sum_{v\in V} d_H(v).$$

We will show that an internal insertion or deletion to H *decreases* Φ by at least $\beta\lambda/18$, while an external deletion to G *increases* Φ by at most 2β, and an external insertion has no effect on ϕ; together these facts clearly imply the lemma. Given any update to edge (x, y), we let $d^{\mathrm{O}}(x), d^{\mathrm{O}}(y), d^{\mathrm{N}}(x), d^{\mathrm{N}}(y)$ (O for old, N for new) denote the degrees of x and y before and after the edge update. Let $\Delta\Phi$ denote the change to Φ due to the update.

Consider first an internal insertion of edge (x, y). Clearly $\sum_{v\in V} d_H(v)$ increases by 2. To bound the total change to $\sum_{(u,v)\in H}(d_H(u) + d_H(v))$, observe that all the edge degrees of edges incident to x and y go up by one, so the total increase in edge degree of all these edges is precisely $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. We also added a new edge of edge degree $d^{\mathrm{N}}(x) + d^{\mathrm{N}}(y) = d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y) + 2$; thus $\Delta\Phi = 2(d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)) + 2 - 2\beta(1 - \lambda/2)$. But because our algorithm is locally balancing we know that for an internal insertion $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y) < \beta(1 - 5\lambda/9)$, so $\Delta\Phi < 2\beta(1 - 5\lambda/9) + 2 - 2\beta(1 - \lambda/2) = -\beta\lambda/9 + 2 \leq -\beta\lambda/18$, as desired. (The last inequality follows from $\beta \geq 36\lambda^{-1}$.)

Consider an internal deletion of edge (x, y). The total change to $\sum_{v\in V} d_H(v)$ is now -2. The deletion causes all edge degrees of edges incident to x and y to go down by one, so the total change to all these edge degrees is $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. We also deleted an edge of edge degree $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$. Thus, $\Delta\Phi = -2(d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)) + 2\beta(1 - \lambda/2)$. Since our algorithm is locally balancing we know that for an internal deletion $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y) > \beta(1 - 4\lambda/9)$, so $\Delta\Phi \leq -\beta\lambda/9$, as desired.

Consider finally an external deletion of edge (x, y). The same argument as for internal deletions yields $\Delta\Phi = -2(d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)) + 2\beta(1 - \lambda/2)$. Now however, we have no lower bound on $d^{\mathrm{O}}(x) + d^{\mathrm{O}}(y)$ except that it is clearly positive. This yields $\Delta\Phi \leq 2\beta(1 - \lambda/2) < 2\beta$, as desired.

An external insertion only changes G and not H, so it has no effect on ϕ. $\square$

The main difference between maintaining an EDCS in general graphs compared to small arboricity graphs is that because we can have many edges, we cannot afford to alert all the owned neighbors of some vertex v *every* time $d_H(v)$ changes. The basic idea is that instead of directly working with $d_H(v)$, each vertex v will store a public value $\text{PUB}(v)$, and when the algorithm determines what to do with an edge (v, w), it will only look at its *public* edge degree, $\text{PUB}(v) + \text{PUB}(w)$. The algorithm will avoid excessive computation by only occasionally changing $\text{PUB}(v)$, while guaranteeing that the algorithm nonetheless functions property by ensuring that $\text{PUB}(v)$ is always not too far off from $d_H(v)$.

Definition 26 *We say that an edge (v, w) is* violating *if (v, w) violates one of the* EDCS *properties: i.e. if $(v, w) \in H$ and $d_H(v) + d_H(w) > \beta$, or $(v, w) \in G \setminus H$ and $d_H(v) + d_H(w) < \beta(1-\lambda)$. We say that an edge (v, w) is* publicly unbalanced *if $(v, w) \in H$ and $\text{PUB}(v) + \text{PUB}(w) > \beta(1-2\lambda/9)$ or if $(v, w) \in G \setminus H$ and $\text{PUB}(v) + \text{PUB}(w) < \beta(1-7\lambda/9)$. We say that an edge is publicly balanced if it is not publicly unbalanced.*

Our algorithm will always maintain the following three invariants:

Publicity Invariant: $|\text{PUB}(v) - d_H(v)| \leq \beta\lambda/9$.

Correctness Invariant: After the algorithm finishes processing any update to G, all edge are publicly balanced.

Balancing Invariant: The algorithm only performs internal insertions/deletions on edges in H that are publicly unbalanced.

It is is not hard to see that invariants 1 and 2 together guarantee that after the algorithm finishes processing an update to G, there are no violating edges in the EDCS. Invariants 1 and 3 together guarantee that the algorithm is locally balancing (Definition 25).

To find locally unbalanced edges, we present an analogue of the violation oracle of Lemma 34. One of the main reasons it is easier to work with $\text{PUB}(v)$ instead of the real $d_H(v)$ is that although updating an edge (u, v) can change $d_H(u)$ and $d_H(v)$, it cannot in and of itself create new publicly unbalanced edges; *a publicly unbalanced edge can only be created by changing* $\text{PUB}(v)$ *or* $\text{PUB}(w)$. That being said, by the publicity invariant updating an edge (u, v) can indirectly force a change to $\text{PUB}(v)$ or $\text{PUB}(w)$.

Note that in general graphs, we use Theorem 17 to maintain an orientation of the edges of G such that each vertex owns $O(\sqrt{m})$ edges. But unlike in the small arboricity case, our algorithm does not need a separate procedure for handling edge reorientations because by Theorem 17 every update to G only causes $O(1)$ reorientations, so we can bluntly model a reorientation as a deletion of the edge followed by an insertion of the same edge pointing in the other direction.

Lemma 41 *Suppose that we have a graph G for which we maintain a dynamic orientation in which each vertex owns $O(\sqrt{m})$ edges. There exists a data structure* PBO *(public balancing oracle) on the graph G that supports the following operations:*

- PBO.*find(v) returns* all *publicly unbalanced edges incident to v in time $O(\sqrt{m} +$ (number of publicly unbalanced edges returned$))$.*
- PBO.*change-status(v, w) adds/removes edge (v, w) from H, or inserts/deletes (v, w) from G in $O(1)$ time.*
- PBO.*change-public(v, p) changes the value of* $\text{PUB}(v)$ *to p in time* $O(\sqrt{m}+|p-(\text{original PUB}(v))|)$.

Proof: The proof is similar to that of Lemma 34. We use the data structures High Threshold Table (HTT) and Low Threshold Table (LTT) from Lemma 39. For every vertex v, we build a HTT N_v^H which contains all vertices w for which $(v, w) \in H$ and (v, w) is *not* owned by v; we set $\text{KEY}(w) = \text{PUB}(w)$ and $\text{HIGH} = \beta(1 - 2\lambda/9) - \text{PUB}(v)$. We also build an LTT called $N_v^{G \setminus H}$ which contains all vertices w for which $(v, w) \in G \setminus H$ and (v, w) is not owned by v; we set $\text{KEY}(w) = \text{PUB}(w)$ and $\text{LOW} = \beta(1 - 7\lambda/9) - \text{PUB}(v)$.

To implement PBO.find(v), we first find all the publicly unbalanced edges that are owned by v in $O(\sqrt{m})$ time by simply scanning the $O(\sqrt{m})$ owned edges of v. We now need to find all publicly unbalanced edges (v, w) that are not owned by v. Let us first find all edges $(v, w) \in H$ that are not owned by v and for which $\text{PUB}(v) + \text{PUB}(w) > \beta(1 - 2\lambda/9)$, which is precisely the set of elements $w \in N_v^H$ for which $\text{KEY}(w) > \text{HIGH}$. Since N_v^H is an HTT we can find one such element w in $O(1)$ time. To find all of them, we repeatedly find such an element w and then temporarily remove it from the HTT until we can no find no more such elements w: we then insert all the temporarily removed elements back into the HTT. Finding the publicly unbalanced edges not owned by v thus requires a total time of $O(1 + (\text{number of elements } w \text{ found}))$. We can similarly find all edges $(v, w) \in G \setminus H$

that are not owned by v and for which $\text{PUB}(v) + \text{PUB}(w) < \beta(1 - 7\lambda/9)$ by looking for elements $w \in N_v^{G\setminus H}$ for which $\text{KEY}(w) < \text{LOW}$.

Implementing PBO.$change - status(v, w)$ is easy, because the values $\text{PUB}(v)$ and $\text{PUB}(w)$ do not change. Thus, the most we would have to do is add or remove (v, w) from one of N_v^H, $N_v^{G\setminus H}$, N_w^H, or $N_w^{G\setminus H}$, which can be done in $O(1)$ time.

To implement PBO.$change - public(v, p)$, let $p_0(v)$ be the value of $\text{PUB}(v)$ before the change. First, for all edges (v, w) that are owned by v, the change to $\text{PUB}(v)$ will change $\text{KEY}(v)$ from p_0 to p in N_w^H or $N_w^{G\setminus H}$. A key change in an HTT or LTT can be implemented in $O(1)$ time, and our orientation guarantees that v owns $O(\sqrt{m})$ edges, so this operation takes $O(\sqrt{m})$ time. Secondly, we need to change HIGH and LOW in N_v^H and $N_v^{G\setminus H}$ since they are defined in terms of $\text{PUB}(v)$. In particular, we have to add $(p_0 - p)$ to HIGH and LOW (note that this expression might be negative). We know that in an HTT or LTT we can change HIGH or LOW by 1 in $O(1)$ time, so we can change it by $(p_0 - p)$ in time $O(|p_0 - p|)$, as desired. □

Lemma 42 *There exists a locally balancing algorithm A for maintaining H such that the amortized update time of A per update to G is $O((m\lambda^{-1}/\beta)(1+$ number of internal updates performed by $A))$.*

Proof: Just as in Lemma 33, we do not need to bound the number of internal updates: as long as the algorithm is locally balancing, Lemma 40 does the bounding for us. As discussed above, the algorithm A will satisfy the conditions of the lemma as long as it maintains the three invariants above (publicity, correctness, balancing). The algorithm maintains a stack (STACK) of all vertices that might potentially violate the publicity invariant; this invariant can only become violated when $d_H(v)$ changes, which in turn only occurs when we add/remove edge (v, w) from H, so whenever we add/remove an edge (v, w) we put v and w on STACK. If a vertex v on the stack violates the publicity invariant, we change $\text{PUB}(v)$ to $d_H(v)$; changing $\text{PUB}(v)$ can create publicly unbalanced edges, so we use PBO.find(v) to detect and fix all of these.

The full algorithm is defined in Figure 7.2. We first verify that the invariants are maintained. As discussed above, the publicity invariant is maintained because whenever $d_H(v)$ changes due to an edge change in H the algorithm algorithm adds v to STACK, and explicitly maintains the publicity invariant for all vertices on STACK.

To see that the correctness invariant is maintained, note that an edge (v, w) can only become unbalanced for two reasons: it can be a new edge added to G, or one of $\text{PUB}(v)$ or $\text{PUB}(w)$ must

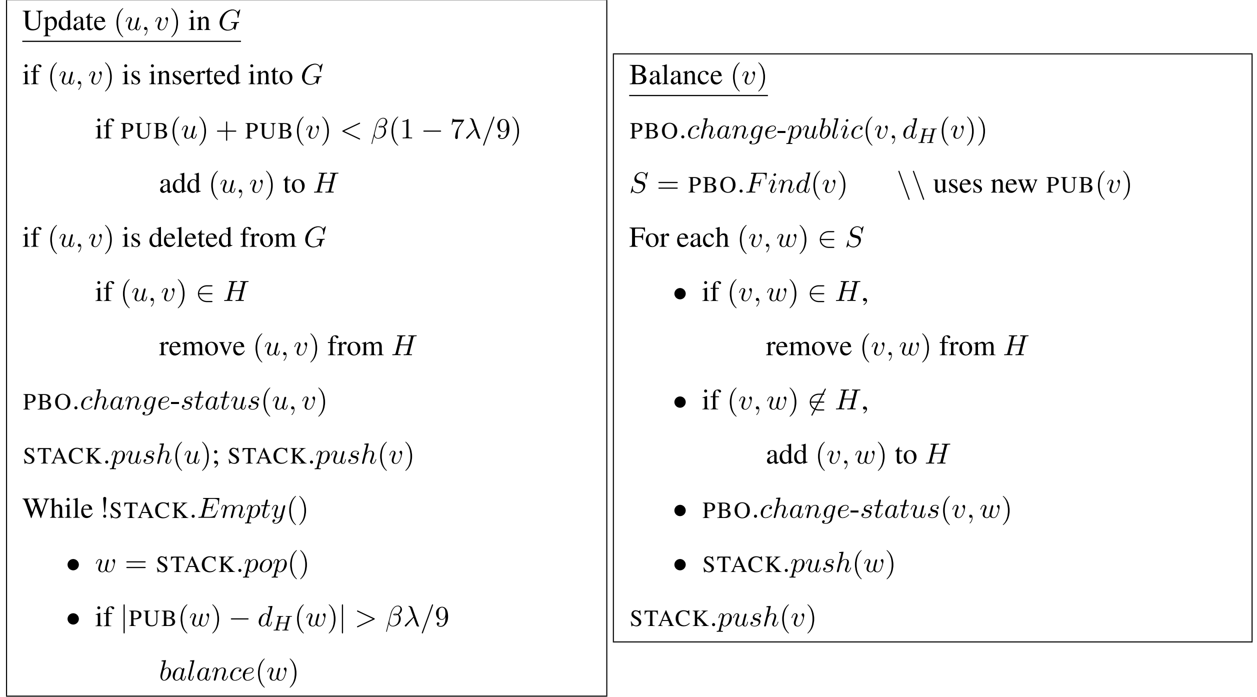

Figure 7.2: How the algorithm of Lemma 42 handles an update to G

have changed. The first lines of the update procedure in Figure 7.2 explicitly handle any unbalanced edge inserted into G by adding it to H. The only time we ever change PUB(v) is in the balance procedure, which after changing PUB(v) finds and fixes all unbalanced edges incident to v.

The balancing invariant is clearly maintained, since all internal updates occur through balance(v), which only updates the unbalanced edges found through PBO.find(v).

For running time analysis, we are allowed $O(m\lambda^{-1}/\beta)$ time per external update, plus an additional $O(m\lambda^{-1}/\beta)$ time per internal update. We thus need a charging scheme which shows that all the steps of the algorithm can be executed in time $O(m\lambda^{-1}/\beta)$ per update to H. In particular, we will think of every update to an edge (x, y) in H as giving $O(m\lambda^{-1}/\beta)$ credits to both x and y.

The only non-constant operations performed by the algorithm are PBO.find(v) and PBO.change-public$(v, d_H(v))$, both of which only occur during the execution of balance(v). The key observation is that balance(v) only occurs when the publicity invariant is violated for v, and since $d_H(v)$ only changes due to an update of an edge in H incident to v, at least $\beta\lambda/9$ such updates must have occurred since the last time balance(v) was executed. Thus, by the time balance(v) is executed again, v has accrued $O((m\lambda^{-1}/\beta) \cdot (\beta\lambda/9)) = O(\sqrt{m})$ credits, which pays for the $O(\sqrt{m})$ term

in PBO.find(v) and PBO.change-public($v, d_H(v)$).

The other terms in PBO.find(v) and PBO.change-public($v, d_H(v)$) only require an additional $O(1)$ time per update to H. In PBO.find(v) the additional term is O(number of unbalanced edges found), and each unbalanced edge found leads to an internal update to H. In PBO.change-public($v, d_H(v)$), the additional term is $O(|d_H(v) - p_0|)$, where p_0 was the original $\text{PUB}(v)$. For p_0 and $d_H(v)$ to differ, at least $|p_0 - d_H(v)|$ updates to edges in H incident to v must have occurred since the last execution of balance(v) (when $\text{PUB}(v)$ was set to p_0), so if each of these updates gives $O(1)$ credits to v, it will be enough to implement PBO.change-public($v, d_H(v)$). □

Proof of Theorem 19 We use the algorithm of Lemma 42. Since this algorithm is locally balancing, by Lemma 40 each update to G leads to amortized $O(\lambda^{-1})$ internal updates, yielding the $O(\lambda^{-1})$ bound for amortized update ratio. Thus, by Lemma 42, an update to G can be processed in amortized time $O(\sqrt{m}\lambda^{-2}/\beta)$. By Theorem 17, the time to maintain an orientation in G only causes a constant overhead on top of this. □

7.8.5 Dynamic Orientation: Proving Theorem 17

For ease of reading, we repeat the statement of Theorem 17 below:

Theorem 22 *In a graph G, we can maintain an orientation, under insertions and deletions, with the following properties: the max load at all times is at most $3\sqrt{m}$, the worst-case number of edge reorientations per insert/deletion in G is $O(1)$, and the worst-case time spent per insertion/deletion in G is $O(1)$.*

The dynamic orientation algorithm we use is a bit technical, but at its core boils down to a simple lazy update algorithm. For simplicity of analysis, we assume that we begin with a graph with no edges, and update from there. Let us start with a few definitions and observations.

Definition 27 *Define a vertex to be* small *if it has degree (not load) less than $2\sqrt{m}$ and* large *if it has degree greater than or equal to $2\sqrt{m}$. We say that a vertex is* very small *if it has degree less than $\sqrt{m}$ (note: a very small vertex is also small). Given some orientation, define a vertex in the current orientation to be* heavy *if it has load greater than $2\sqrt{m}$, and light otherwise. Define a vertex to be* very heavy *if it has load greater than $3\sqrt{m}$ (note: a very heavy vertex is also heavy). Note*

that heavy vertices are always large, and small vertices are always light. Finally, m always refers to the number of edges in the current *version of the graph.*

Observation 2 *A graph can contain at most $\sqrt{m}$ large vertices, and at most $2\sqrt{m}$ vertices that are not very small (i.e. vertices of degree $\geq \sqrt{m}$). Otherwise, the total degree of these vertices would be greater than $2\sqrt{m}\sqrt{m} = 2m$, so the number of edges in the graph would be greater than m.*

Observation 2 yields a simple method for computing a $2\sqrt{m}$-orientation in linear time. Simply let small vertices own all of their edges; if an edge is between two small vertices or two large vertices, it can be oriented arbitrarily. Now no vertex can have load greater than $2\sqrt{m}$, as then all its owned edges would go to large vertices, in which case there would be more than $2\sqrt{m}$ large vertices in the graph, which contradicts Observation 2.

The basic idea behind our algorithm is as follows: as G changes, we will reorient edges to maintain the following invariant:

Invariant 1 *There exists no edge (u, v) such that u is very small, v is very heavy, and (u, v) is owned by v.*

Lemma 43 *Any orientation that satisfies Invariant 1 has maximum load at most $3\sqrt{m}$.*

Proof: The proof follows directly from Observation 2. The orientation cannot have a very heavy vertex v, because by Invariant 1, for all $3\sqrt{m}$ edges (u, v) owned by v, vertex u would have to be not very small, but by Observation 2 there are at most $2\sqrt{m}$ not very small vertices in the graph. □

Note that there are two ways a vertex u can go from being small to very small. The obvious way is if many edges incident to u are deleted. Another possibility, however, is if many edges are added to other parts of the graph: for example, if originally u has degree 10 and the graph has 70 edges then u is small but not very small, but if 31 more edges are added to the graph then u will become very small. Similarly, a vertex v can become very heavy either due to edge insertions incident to v, or due to a large number of edge deletions in the graph as a whole. We will have to handle these two types of transitions separately.

Our dynamic orientation algorithm will proceed by occasionally scanning some set of edges and *fixing* the edges that are oriented in the wrong direction.

Definition 28 *Given an edge (u, v), we define the* FIX(u, v) *operation as follows:*

1. *if u is small and v is large then reorient edge (u, v) so that u owns the edge (if u already owned (u, v) then do nothing).*

2. *if v is small and u is large then reorient (u, v) so that v owns the edge.*

3. *if u and v are both small or both large, then do nothing (i.e. keep the current orientation of edge (u, v)).*

We now show a simple algorithm for maintaining a $3\sqrt{m}$-orientation in *amortized* update time $O(1)$. We will not do a formal analysis, since this algorithm is only meant to serve as intuition for the worst-case algorithm below. For each vertex v the algorithm will maintain the set of edges owned by v. The algorithm is as follows: whenever due to some change in G a vertex v reaches load above $3\sqrt{m}$, we run operation FIX(u, v) for all edges (u, v) owned by v. Note that after all the edges owned by v are fixed, the load of v will be at most $2\sqrt{m}$. This is because for all edges (u, v) where u is small, ownership of edge (u, v) will be given to u, and by Observation 2 there are are most $2\sqrt{m}$ large vertices u.

We now sketch the update time analysis. Whenever a vertex v reaches load above $3\sqrt{m}$ the algorithm fixes all the edges owned by v – that is, $3\sqrt{m} + 1 = O(\sqrt{m})$ edges. Rebuilding a vertex v thus requires $O(\sqrt{m})$ time and $O(\sqrt{m})$ edge reorientations. How often can such a rebuilding occur? We know that after v is rebuilt, it has load at most $2\sqrt{m}$. There are now two ways in which v could once again transition to being very heavy, and so have to be rebuilt once again. One possibility is that there are $\sqrt{m}$ edge insertions incident to v: the $O(\sqrt{m})$ rebuild time can then be amortized over these $\sqrt{m}$ edge insertions, leading to amortized time $O(1)$. Alternatively, if the total number of edges in the graph decreases by a factor $(3/2)^2 \geq 2$, then *all* vertices could go from having load $2\sqrt{m}$ to $3\sqrt{m}$, and so all vertices will have to be rebuilt. But the $O(m)$ time required to rebuild the entire graph can be amortized over the $\Omega(m)$ deletions required for m to decrease by a factor of 2.

Worst Case Update time As with the amortized algorithm, our worst-case algorithm will repeatedly run the FIXoperation. But instead of doing this work all at once when a vertex violates Invariant 1, the algorithm will stagger this work over many updates. To maintain Invariant 1, we need to concern ourselves with edges (u, v) where u is very small and v is very heavy. In particular, we need to ensure that v does not end up with ownership of the edge. In order to arrive at this extreme case,

however, there must be a long transition period where u is small and v is heavy, but they are not yet *very* small or *very* heavy. If we can ensure that at some point during this transition period we run FIX(u, v), then edge u will gain ownership of edge (u, v), and Invariant 1 will be maintained.

Recall that there are two ways u can transition from small to very small (or, respectively, v from heavy to very heavy): either $\Omega(\sqrt{m})$ deletions (resp. insertions) of edges incident to u (local changes), or $\Omega(m)$ insertions (resp. deletions) of edges in the graph as a whole (global changes). Our algorithm needs to handle both types of transitions, so every time the adversary inserts/deletes edge (u, v) in the graph, the algorithm will fix a constant number of edges incident to u and v (local fixes), and a constant number of edges in the graph as a whole (global fixes). We will show that these fixes are enough to maintain Invariant 1.

The Algorithm More formally, for every edge v, the algorithm will maintain two sets of edges: $N(v)$ contains all edges incident to v, while $N^O(v)$ contains all edges incident to v that are owned by v. Each set will be stored as a doubly linked list, and the algorithm will repeatedly fix these lists in a round-robin fashion. Thus for each list there will be a current pointer from which fixes will proceed; whenever a new edge is inserted into the list, it will be inserted at the very end, i.e. right behind the current pointer. The algorithm will also maintain a global list E of all the edges in the graph, as well as a current pointer for this list.

We now describe the algorithm. Whenever the adversary inserts or deletes edge (u, v), the algorithm performs the following operations:

1. if the adversary *inserts* edge (u, v), then run FIX(u, v) (if u and v are both small or both large orient edge (u, v) arbitrarily).

2. Run operation FIX on the next 10 edges in the global edge list E. Move the current pointer accordingly. We refer to these fixes as *global fixes*.

3. If u is small, run operation FIX on the next 10 edges in $N(u)$. Move the current pointer accordingly. Do the same for vertex v (if v is small). We refer to these fixes as *local fixes incident to* u *(resp.* v*).*

4. if u is heavy, run operation FIX on the next 10 edges in $N^O(u)$. Move the current pointer accordingly. Do the same for vertex v (if v is heavy). We refer to these fixes as *local fixes*

incident to u *(resp.* v*).*

Analysis It is clear that our algorithm fixes at most 31 edges for every change to the graph: 10 global fixes, at most 20 local fixes, and a final fix for the newly inserted edge. Thus the algorithm has worst-case constant update time, and performs a worst-case constant number of reorientations per change to G.

We must now show that the algorithm maintains an orientation with maximum load $3\sqrt{m}$. By Lemma 43, it is sufficient to prove the following:

Lemma 44 *The dynamic orientation algorithm maintains Invariant 1.*

Proof: Let us say, for contradiction, that at some point during the execution of the algorithm there is an edge (u, v) in the graph such that u is very small, v is very heavy, and (u, v) is owned by v. Let T be the first time such an event occurs. Let us consider the last time $T^{'}$ during which v was given ownership of edge (u, v) (this might have been during the original insertion of (u, v)). Since the algorithm only assigns ownership within the FIX(u, v) operation, we must have that at time $T^{'}$ either u was large or v was small (and hence light), since otherwise ownership could not have gone to v. Thus, at some point between $T^{'}$ and T a transition must have occurred, after which u was small and v has heavy. Let T^{*} be the time of the *last* such transition. That is, right before T^{*} either u was large or v was light, but during the whole time interval $[T^{*}, T]$ we always have that u is small and v is heavy. Also, since T^{*} comes after $T^{'}$, and $T^{'}$ was the last time ownership of (u, v) was given to v, we have that between T^{*} and T, edge (u, v) is always owned by v. This leads to the cornerstone of our proof by contradiction: we know that *the algorithm must never have run* FIX(u, v) *during the time interval* $[T^{*}, T]$, because throughout that interval u is small and v is heavy, so FIX(u, v) would have given ownership to u.

We know that right before T^{*} either u was large or v was light. The analysis of the two cases is essentially identical, but for clarity we handle them separately. For both cases we define the following: let m^{*} be the number of edges in the graph at time T^{*}, and let m be the number of edges at time T.

Case 1: right before time T^{*} vertex v was light. At time T we have that v is *very* heavy. Note that a lot of dynamic updates to G must occur for v to go from light to very heavy. For example,

there could have been $\sqrt{m}$ edge insertions incident to v, or there could have been a lot of edge deletions in other parts of the graph. On the other hand, there could not have been too many updates between time T^* and time T, because eventually one of these updates would cause the algorithm to run $\textsc{fix}(u,v)$, which we argued above cannot occur in the time interval $[T^*, T]$. We will thus exhibit both an upper and a lower bound on the necessary number of updates in time interval $[T^*, T]$, and show that they are contradictory, thus showing that the originally assumed violation of Invariant 1 cannot occur.

Let us first upper bound the number of updates in time interval $[T^*, T]$. We know that during this interval, the algorithm performed less than m^* global fixes, because (u, v) already existed at time T^* and edges are fixed in a round-robin fashion, so if there were m^* global fixes the algorithm would have fixed all edges that existed at time T^* and in particular would have run $\textsc{fix}(u,v)$, which we know is impossible. Since the algorithm performs 10 global fixes per update to G, there could have been at most $m^*/10$ updates in time interval $[T^*, T]$. Similarly, we can argue that there can must be less than $3\sqrt{m^*}$ local fixes incident to v in the time interval $[T^*, T]$, as otherwise the algorithm would run $\textsc{fix}(u,v)$. This is because during the entire time interval we always have $(u, v) \in N^O(v)$, and yet at time T^* vertex v is not yet very heavy so it has load less than $3\sqrt{m^*}$; since v is always heavy in the time interval $[T^*, T]$, the local fixes all go through $N^O(v)$ in a round-robin fashion, so there can be at most $3\sqrt{m^*}$ such local fixes before the algorithm reaches (u, v). Since the algorithm performs 10 local fixes per edge change, we know that the adversary performed at most $\frac{3\sqrt{m^*}}{10}$ insertions/deletions of edges incident to v during the interval $[T^*, T]$. *To summarize, we know that during the interval $[T^*, T]$, the adversary performed a total of at most $\frac{m^*}{10}$ updates, and at most $\frac{3\sqrt{m^*}}{10}$ updates of edges incident to v*

We know yield a contradiction by showing that since v is by definition light right before T^*, this small number of updates between time T^* and time T is not enough to cause v to be very heavy at time T. We know that

$$\textsc{load}(v) \text{ at time } T \geq 3\sqrt{m}. \tag{7.9}$$

While

$$\textsc{load}(v) \text{ right before time } T^* \leq 2\sqrt{m^*}. \tag{7.10}$$

However, since there were at most $m^*/10$ edge changes to G during time interval $[T^*, T]$, we know

that

$$m \geq \frac{9}{10} m^*. \tag{7.11}$$

Also, note that during the whole interval from T^* to T vertex v was heavy, so $\text{FIX}(u, v)$ would never reorient an edge towards v; thus, the only way the load of v could increase in time interval $[T^*, T]$ is due to newly inserted edges incident to v. However, we know that there were at most $\frac{3\sqrt{m^*}}{10}$ edge changes incident to v during that time interval, so

$$\text{LOAD}(v) \text{ at time } T \leq \frac{3\sqrt{m^*}}{10} + \text{LOAD}(v) \text{ right before time } T^*. \tag{7.12}$$

Combining Equations 7.10, 7.11, and 7.12 above yields

$$\begin{aligned}
\text{LOAD}(v) \text{ at time } T &\leq \frac{3\sqrt{m^*}}{10} + \text{LOAD}(v) \text{ right before time } T^* \\
&\leq \frac{3\sqrt{m^*}}{10} + 2\sqrt{m^*} \\
&\leq \frac{3\sqrt{10m/9}}{10} + 2\sqrt{10m/9} \\
&\leq 2.3(\frac{\sqrt{10}}{\sqrt{9}})\sqrt{m} \\
&< 3\sqrt{m}
\end{aligned} \tag{7.13}$$

Note that Equation 7.13 directly contradicts Equation 7.9, so we are done.

Case 2: vertex u was large right before time T^*. The analysis is exactly analogous to Case 1 above: we know that in the time interval $[T^*, T]$ there must have less than m^* global edge fixes and less than $2\sqrt{m^*}$ local edge fixes incident to u, because otherwise we would have fixed edge (u, v). But this implies that there were at most $m^*/10$ edge updates to G as a whole, and at most $\sqrt{m^*}/5$ edge updates incident to u. A bit of simple algebra analogous to that in the Equations above then shows that this small number of updates is not sufficient for u to transition from large to very small, which yields the desired contradiction. □

7.9 Conclusions

Our main result is a fully dynamic algorithm for maximum matching with update time $O(m^{1/4})$ and approximation ratio $O(3/2 + \epsilon)$. This is the first fully dynamic matching algorithm to achieve a

$o(m^{1/2})$ update time while maintaining a better-than-2-approximate matching. It is also the fastest known deterministic algorithm for achieving *any* constant approximation, and certainly any better-than-2 approximation. (Since the publication of our paper, Bhattacharya *et al.* [Bhattacharya et al., 2016] have made some progress in both of these dimensions; see end of Section 7.1.) The two main open questions are whether we can achieve a $(1+\epsilon)$ approximation in update time $O(m^{1/2-\zeta})$ for some fixed $\zeta > 0$, and whether we can achieve some better-than-2 approximation (even say 1.99) in polylog update time. These results would be interesting even if randomized, and/or limited to bipartite graphs.

Another possible direction of future research would be to extend our techniques. Our main technique is a new characterization of better-than-2 approximate matchings in terms of local constraints. Given that local problems tend to be much easier in the dynamic setting, as well as in several other models of computations such as streaming or distributed computing, it would be interesting to develop other local characterizations of approximate matchings, especially ones that don't rely on an intermediate subgraph. Also, can approximate *weighted* matchings be characterized in such a fashion?

Part III

Bibliography

Bibliography

[Abraham et al., 2014] Abraham, I., Chechik, S., and Talwar, K. (2014). Fully dynamic all-pairs shortest paths: Breaking the o(n) barrier. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 1–16.

[Abraham et al., 2010] Abraham, I., Fiat, A., Goldberg, A. V., and Werneck, R. F. (2010). Highway dimension, shortest paths, and provably efficient algorithms. In *Proc. the 21st SODA*, pages 782–793, Austin, Texas, USA.

[Aingworth et al., 1999] Aingworth, D., Chekuri, C., Indyk, P., and Motwani, R. (1999). Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181.

[Awerbuch et al., 1998] Awerbuch, B., Berger, B., Cowen, L., and Peleg, D. (1998). Near-linear time construction of sparse neighborhood covers. *SIAM J. Comput.*, 28(1):263–277.

[Baswana et al., 2011a] Baswana, S., Gupta, M., and Sen, S. (2011a). Fully dynamic maximal matching in o (log n) update time. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:383–392.

[Baswana et al., 2011b] Baswana, S., Gupta, M., and Sen, S. (2011b). Fully dynamic maximal matching in O (log n) update time. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 383–392.

[Baswana et al., 2003] Baswana, S., Hariharan, R., and Sen, S. (2003). Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proceedings of the Fourteenth Annual ACM-*

SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA., pages 394–403.

[Baswana et al., 2007] Baswana, S., Hariharan, R., and Sen, S. (2007). Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *J. Algorithms*, 62(2):74–92.

[Baswana and Kavitha, 2006] Baswana, S. and Kavitha, T. (2006). Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proceedings of the 47th Annual Symposium on Foundations of Computer Science*, FOCS, pages 591–602.

[Baswana et al., 2012] Baswana, S., Khurana, S., and Sarkar, S. (2012). Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms*, 8(4):35.

[Battista and Tamassia, 1996] Battista, G. D. and Tamassia, R. (1996). On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997.

[Bender et al., 2009] Bender, M. A., Fineman, J. T., and Gilbert, S. (2009). A new approach to incremental topological ordering. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 1108–1115.

[Bernstein, 2009] Bernstein, A. (2009). Fully dynamic approximate all-pairs shortest paths with constant query and close to linear update time. In *Proc. of the 50th FOCS*, pages 50–60, Atlanta, GA, USA.

[Bernstein, 2012] Bernstein, A. (2012). Near linear time $(1+\epsilon)$-approximation for restricted shortest paths in undirected graphs. In *Proc. of the 23rd SODA*, pages 189–201, Kyoto, Japan.

[Bernstein, 2013] Bernstein, A. (2013). Maintaining shortest paths under deletions in weighted directed graphs. In *STOC*, pages 725–734.

[Bernstein, 2016] Bernstein, A. (2016). Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574.

[Bernstein and Roditty, 2011] Bernstein, A. and Roditty, L. (2011). Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proc. of the 22nd SODA*, pages 1355–1365, San Francisco, California, USA.

[Bernstein and Stein, 2015] Bernstein, A. and Stein, C. (2015). Fully dynamic matching in bipartite graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 167–179.

[Bernstein and Stein, 2016] Bernstein, A. and Stein, C. (2016). Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 692–711.

[Bhattacharya et al., 2015a] Bhattacharya, S., Henzinger, M., and Italiano, G. F. (2015a). Deterministic fully dynamic data structures for vertex cover and matching. In *SODA 2015, 4-6 January, San Diego, CA, USA*, pages 785–804.

[Bhattacharya et al., 2015b] Bhattacharya, S., Henzinger, M., and Italiano, G. F. (2015b). Deterministic fully dynamic data structures for vertex cover and matching. In *SODA 2015, 4-6 January, San Diego, CA, USA*, pages 785–804.

[Bhattacharya et al., 2016] Bhattacharya, S., Henzinger, M., and Nanongkai, D. (2016). New deterministic approximation algorithms for fully dynamic matching. *CoRR*, abs/1604.05765.

[Bosek et al., 2014] Bosek, B., Leniowski, D., Sankowski, P., and Zych, A. (2014). Online bipartite matching in offline time. In *55th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2014, 16-21 October, 2014, Philadelphia, PA, USA*, pages 384–393.

[Chaudhuri et al., 2009] Chaudhuri, K., Daskalakis, C., Kleinberg, R. D., and Lin, H. (2009). Online bipartite perfect matching with augmentations. In *INFOCOM*, pages 1044–1052.

[Chechik, 2014] Chechik, S. (2014). Approximate distance oracles with constant query time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663.

[Cohen, 1998] Cohen, E. (1998). Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. Comput.*, 28(1):210–236.

[Cohen and Zwick, 2001] Cohen, E. and Zwick, U. (2001). All-pairs small-stretch paths. *J. Algorithms*, 38(2):335–353.

[Demetrescu and Italiano, 2001] Demetrescu, C. and Italiano, G. F. (2001). Fully dynamic all pairs shortest paths with real edge weights. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 260–267.

[Demetrescu and Italiano, 2004] Demetrescu, C. and Italiano, G. F. (2004). A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992.

[Demetrescu and Italiano, 2005] Demetrescu, C. and Italiano, G. F. (2005). Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n^2 barrier. *J. ACM*, 52(2):147–156.

[Dinitz, 2006] Dinitz, Y. (2006). Dinitz' algorithm: The original version and Even's version. In *Essays in Memory of Shimon Even*, pages 218–240.

[Dor et al., 2000] Dor, D., Halperin, S., and Zwick, U. (2000). All-pairs almost shortest paths. *SIAM J. Comput.*, 29(5):1740–1759.

[Duan and Pettie, 2014] Duan, R. and Pettie, S. (2014). Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1.

[Elkin and Peleg, 2004] Elkin, M. and Peleg, D. (2004). (1+epsilon, beta)-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631.

[Eppstein et al., 1997] Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. (1997). Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696.

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

[Feldman et al., 2010] Feldman, J., Henzinger, M., Korula, N., Mirrokni, V., and Stein, C. (2010). Online stochastic packing applied to display ad allocation. *Algorithms–ESA 2010*, pages 182–194.

[Frederickson, 1985] Frederickson, G. N. (1985). Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798.

[Galil et al., 1999] Galil, Z., Italiano, G. F., and Sarnak, N. (1999). Fully dynamic planarity testing with applications. *J. ACM*, 46(1):28–91.

[Gupta et al., 2014] Gupta, A., Kumar, A., and Stein, C. (2014). Maintaining assignments online: Matching, scheduling, and flows. In *SODA*, pages 468–479.

[Gupta and Peng, 2013] Gupta, M. and Peng, R. (2013). Fully dynamic (1+e)-approximate matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 548–557.

[Henzinger and King, 1995] Henzinger, M. and King, V. (1995). Fully dynamic biconnectivity and transitive closure. In *Proc. of the 36th FOCS*, pages 664–672, Milwaukee Wisconsin.

[Henzinger et al., 2013] Henzinger, M., Krinninger, S., and Nanongkai, D. (2013). Dynamic approximate all-pairs shortest paths: Breaking the $o(mn)$ barrier and derandomization. In *FOCS 2013*, pages 538–547.

[Henzinger et al., 2014a] Henzinger, M., Krinninger, S., and Nanongkai, D. (2014a). Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *FOCS 2014*, pages 146–155.

[Henzinger et al., 2014b] Henzinger, M., Krinninger, S., and Nanongkai, D. (2014b). Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *STOC*, pages 674–683.

[Henzinger et al., 2014c] Henzinger, M., Krinninger, S., and Nanongkai, D. (2014c). A subquadratic-time algorithm for decremental single-source shortest paths. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1053–1072.

[Henzinger et al., 2015a] Henzinger, M., Krinninger, S., and Nanongkai, D. (2015a). Improved algorithms for decremental single-source reachability on directed graphs. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 725–736.

[Henzinger et al., 2015b] Henzinger, M., Krinninger, S., Nanongkai, D., and Saranurak, T. (2015b). Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th ACM Symposium on Theory of Computing (STOC 2015)*.

[Henzinger et al., 2015c] Henzinger, M., Krinninger, S., Nanongkai, D., and Saranurak, T. (2015c). Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 21–30.

[Henzinger and King, 1999] Henzinger, M. R. and King, V. (1999). Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516.

[Henzinger and Thorup, 1997] Henzinger, M. R. and Thorup, M. (1997). Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379.

[Holm et al., 1998] Holm, J., de Lichtenberg, K., and Thorup, M. (1998). Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 79–89.

[Holm et al., 2001] Holm, J., de Lichtenberg, K., and Thorup, M. (2001). Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760.

[Italiano et al., 1993] Italiano, G. F., Poutré, J. A. L., and Rauch, M. (1993). Fully dynamic planarity testing in planar embedded graphs (extended abstract). In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings*, pages 212–223.

[Ivkovic and Lloyd, 1994] Ivkovic, Z. and Lloyd, E. L. (1994). Fully dynamic maintenance of vertex cover. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, WG '93, pages 99–111, London, UK, UK. Springer-Verlag.

[Kapron et al., 2013] Kapron, B. M., King, V., and Mountjoy, B. (2013). Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1131–1142.

[Katriel and Bodlaender, 2006] Katriel, I. and Bodlaender, H. L. (2006). Online topological ordering. *ACM Transactions on Algorithms*, 2(3):364–379.

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

[Kopelowitz et al., 2014a] Kopelowitz, T., Krauthgamer, R., Porat, E., and Solomon, S. (2014a). Orienting fully dynamic graphs with worst-case time bounds. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, pages 532–543.

[Kopelowitz et al., 2014b] Kopelowitz, T., Pettie, S., and Porat, E. (2014b). 3sum hardness in (dynamic) data structures. *CoRR*, abs/1407.6756.

[Lacki, 2011] Lacki, J. (2011). Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*, pages 1438–1445.

[Mehta et al., 2005] Mehta, A., Saberi, A., Vazirani, U., and Vazirani, V. (2005). Adwords and generalized on-line matching. In *focs05*, pages 264–273.

[Micali and Vazirani, 1980a] Micali, S. and Vazirani, V. V. (1980a). An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings 21st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 17–27.

[Micali and Vazirani, 1980b] Micali, S. and Vazirani, V. V. (1980b). An o(sqrt(|v|) |e|) algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 17–27.

[Nash-Williams, 1961] Nash-Williams, C. S. J. A. (1961). Edge disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36:445–450.

[Neiman and Solomon, 2013a] Neiman, O. and Solomon, S. (2013a). Simple deterministic algorithms for fully dynamic maximal matching. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 745–754.

[Neiman and Solomon, 2013b] Neiman, O. and Solomon, S. (2013b). Simple deterministic algorithms for fully dynamic maximal matching. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 745–754.

[Onak and Rubinfeld, 2010a] Onak, K. and Rubinfeld, R. (2010a). Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 457–464.

[Onak and Rubinfeld, 2010b] Onak, K. and Rubinfeld, R. (2010b). Maintaining a large matching and a small vertex cover. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 457–464.

[Pearce and Kelly, 2006] Pearce, D. J. and Kelly, P. H. J. (2006). A dynamic topological sort algorithm for directed acyclic graphs. *ACM Journal of Experimental Algorithmics*, 11.

[Peleg and Schäffer, 1989] Peleg, D. and Schäffer, A. A. (1989). Graph spanners. *Journal of Graph Theory*, 13(1):99–116.

[Peleg and Solomon, 2016] Peleg, D. and Solomon, S. (2016). Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 712–729.

[Peleg and Ullman, 1989] Peleg, D. and Ullman, J. D. (1989). An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747.

[Pettie, 2009] Pettie, S. (2009). Low distortion spanners. *ACM Transactions on Algorithms*, 6(1).

[Roditty, 2013] Roditty, L. (2013). Decremental maintenance of strongly connected components. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1143–1150.

[Roditty and Zwick, 2004a] Roditty, L. and Zwick, U. (2004a). Dynamic approximate all-pairs shortest paths in undirected graphs. In *Proc. of the 45th FOCS*, pages 499–508, Rome, Italy.

[Roditty and Zwick, 2004b] Roditty, L. and Zwick, U. (2004b). On dynamic shortest paths problems. In *Proc. of the 12th ESA*, pages 580–591.

[Roditty and Zwick, 2008a] Roditty, L. and Zwick, U. (2008a). Improved dynamic reachability algorithms for directed graphs. *SIAM J. Comput.*, 37(5):1455–1471.

[Roditty and Zwick, 2008b] Roditty, L. and Zwick, U. (2008b). Improved dynamic reachability algorithms for directed graphs. *Siam J. Comp.*, 37(5):1455–1471.

[Roditty and Zwick, 2012] Roditty, L. and Zwick, U. (2012). Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683.

[Sankowski, 2005] Sankowski, P. (2005). Subquadratic algorithm for dynamic shortest distances. In *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings*, pages 461–470.

[Sankowski, 2007] Sankowski, P. (2007). Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 118–126.

[Scheinerman and Ullman, 2011] Scheinerman, E. R. and Ullman, D. H. (2011). Fractional graph theory: a rational approach to the theory of graphs.

[Thorup, 2000] Thorup, M. (2000). Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 343–350.

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

[Thorup, 2005] Thorup, M. (2005). Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proc. of the 37th STOC*, pages 112–119.

[Thorup, 2007] Thorup, M. (2007). Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127.

[Thorup and Zwick, 2001] Thorup, M. and Zwick, U. (2001). Compact routing schemes. In *SPAA*, pages 1–10.

[Thorup and Zwick, 2005] Thorup, M. and Zwick, U. (2005). Approximate distance oracles. *Journal of the ACM*, 52(1):1–24.

[Thorup and Zwick, 2006] Thorup, M. and Zwick, U. (2006). Spanners and emulators with sublinear distance errors. In *Proc. of the 17th SODA*, pages 802–809, Miami, Florida.

[Vazirani, 1994] Vazirani, V. V. (1994). A theory of alternating paths and blossoms for proving correctness of the *O*(sqrt{*v e*}) general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109.

[Vazirani, 2012] Vazirani, V. V. (2012). An improved definition of blossoms and a simpler proof of the MV matching algorithm. *CoRR*, abs/1210.4594.