universität
wien

Dissertation

Titel der Dissertation

Faster Approximation Algorithms for Partially Dynamic Shortest Paths Problems

verfasst von

Dipl. Ing. Sebastian Krinninger

angestrebter akademischer Grad

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2015

Studienkennzahl lt. Studienblatt:	A 786 880
Dissertationsgebiet lt. Studienblatt:	Informatik, IK: Computer-unterstützte Optimierung
Betreuerin:	Univ.-Prof. Dr. Monika Henzinger
Zweitbetreuer:	Univ.-Prof. Dr. Georg Ch. Pflug

Abstract

We call an algorithm dynamic if it regularly updates the result of its computations as the input undergoes changes over time. The primary goal is to be faster than the naive algorithm that recomputes the result from scratch after each change. In graph algorithms, the changes to the input graph are usually edge insertions or deletions. In this thesis we focus on partially dynamic graph algorithms where only one type of updates is allowed; either insertions (incremental model) or deletions (decremental model).

We develop faster approximation algorithms for certain variants of the partially dynamic shortest paths problem with respect to the total update time, i.e., the sum of the running times needed to update the result after each change. In this thesis we obtain the following approximation algorithms:

- A randomized algorithm for the decremental approximate all-pairs shortest paths (APSP) problem in unweighted undirected graphs with both a multiplicative error of $1 + \epsilon$ and an additive error of 2 that has a total update time of $\tilde{O}(n^{5/2})$.
- A deterministic algorithm for the decremental approximate APSP problem in unweighted undirected graphs with a multiplicative error of $1 + \epsilon$ and a total update time of $O(mn \log n)$.
- A randomized algorithm for the decremental approximate single-source shortest paths (SSSP) problem in weighted undirected graphs with a multiplicative error of $1 + \epsilon$ and a total update time of $O(m^{1+o(1)})$.
- A randomized algorithm for the decremental single-source reachability problem in directed graphs with a total update time of $o(mn)$ and an extension of the technique to the decremental approximate SSSP problem in weighted directed graphs, achieving $o(mn)$ as well.
- A deterministic algorithm for the incremental approximate SSSP problem in unweighted undirected graphs with a multiplicative error of $1+\epsilon$ and a total update time of $O(m^{3/2}n^{1/2})$ and an extension of the technique to both the incremental and the decremental approximate SSSP problem in the CONGEST model of distributed computing.

Zusammenfassung

Ein Algorithmus heißt dynamisch wenn seine Eingabe mit der Zeit immer wieder Änderungen unterworfen ist und er deshalb das Ergebnis seiner Berechnungen regelmäßig anpasst. Das Hauptziel ist es, schneller zu sein als ein naiver Algorithmus, der das Ergebnis nach jeder Änderung von Grund auf neu berechnet. Für Graphalgorithmen bestehen die Änderungen in der Regel aus Kanteneinfügungen und -löschungen. In dieser Arbeit konzentrieren wir uns auf partiell-dynamische Algorithmen, die nur eine Art von Änderungen erlauben; entweder Einfügungen (inkrementelles Modell) oder Löschungen (dekrementelles Modell).

Wir entwickeln schnellere Approximationsalgorithmen für gewisse Varianten des partiell-dynamischen Kürzeste Wege Problems in Bezug auf die Gesamtlaufzeit, also die Summe der Laufzeiten, die jeweils benötigt wird, um das Ergebnis nach einer Änderung zu aktualisieren. Als Ergebnis dieser Arbeit erhalten wir folgende Approximationsalgorithmen:

- Einen randomisierten Algorithmus für das dekrementelle paarweise Kürzeste Wege Problem in ungewichteten ungerichteten Graphen, der sowohl einen multiplikativen Fehler von $1 + \epsilon$ als auch einen additiven Fehler von 2 hat und dessen Gesamtlaufzeit $\tilde{O}(n^{5/2})$ beträgt.
- Einen deterministischen Algorithmus für das dekrementelle paarweise Kürzeste Wege Problem in ungewichteten ungerichteten Graphen mit multiplikativem Fehler $1 + \epsilon$ und Gesamtlaufzeit $O(mn \log n)$.
- Einen randomisierten Algorithmus für das dekrementelle Kürzeste Wege Problem mit Startknoten in gewichteten ungerichteten Graphen mit multiplikativem Fehler $1 + \epsilon$ und Gesamtlaufzeit $O(m^{1+o(1)})$.
- Einen randomisierten Algorithmus für das dekrementelle Erreichbarkeitsproblem mit Startknoten in gerichteten Graphen mit Gesamtlaufzeit $o(mn)$ sowie eine Erweiterung der Methode für das dekrementelle Kürzeste Wege Problem mit Startknoten in gewichteten gerichteten Graphen, ebenfalls mit Gesamtlaufzeit $o(mn)$.
- Einen deterministischen Algorithmus für das inkrementelle Kürzeste Wege Problem mit Startknoten in ungewichteten ungerichteten Graphen mit multiplikativem Fehler $1 + \epsilon$ und Gesamtlaufzeit $O(m^{3/2}n^{1/2})$ sowie eine Erweiterung der Methode für sowohl einen inkrementellen als auch einen dekrementellen Algorithmus für das Kürzeste Wege Problem mit Startknoten im CONGEST-Modell für verteiltes Rechnen.

Acknowledgments

I want to thank Monika for her never-ending guidance and support over the years, Danupon for the long-lasting and fruitful collaboration, Ittai and Shiri for inviting me to Microsoft Research, Krishnendu and Veronika for the interesting collaborations, Aleksander and Pino for serving on my committee, Prof. Pflug and the IK-members for their input, Martin for sharing the common path, Birgit and Ulli for keeping administrative work low, and all other colleagues at TAA for the pleasant atmosphere. I also want to thank the anonymous reviewers of our papers for their valuable comments and all developers of the free and open-source software I use on a daily basis.

Special thanks go to Harald and Lukas for all that jazz and to Elisabeth and Peter for TEX, Ti*k*Z, and pizza. I am grateful for my family's constant support during the past years (even though I'm not becoming a *real* doctor); my parents Brigitte and Bruno and my sisters Julia and Laura were always there for me, and Oma has even sent me letters overseas. A wholehearted thank you goes to Judith for all her love and consideration – *muxu*!

The research leading to these results has received funding from the University of Vienna / IK I049-N, the Austrian Science Fund (FWF) / P23499-N23, and the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506 and grant agreement no. 317532.

Bibliographic Note

Several results of this thesis were already published in conference and journal papers and thus the chapters of this thesis are based on the following papers:

- **Chapter 2:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *SIAM Journal on Computing* (forthcoming). Announced at FOCS'13.
- **Chapter 3:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155.
- **Chapter 4:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 674–683.
- **Chapter 4:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Improved Algorithms for Decremental Single-Source Reachability on Directed Graphs". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2015, forthcoming.
- **Chapter 5:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2013, pp. 607–619.
- **Chapters 3 and 5:** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Dynamic Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Related Work . . . 3
- 1.3 Preliminaries . . . 7

2 Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization **9**
- 2.1 Introduction . . . 10
  - 2.1.1 The Problem . . . 10
  - 2.1.2 Our Results . . . 11
  - 2.1.3 Techniques . . . 14
  - 2.1.4 Related Work . . . 21
- 2.2 Background . . . 24
  - 2.2.1 Basic Definitions . . . 24
  - 2.2.2 Decremental Shortest-Path Tree Data Structure . . . 26
  - 2.2.3 The Framework of Roditty and Zwick . . . 31
- 2.3 $\tilde{O}(n^{5/2})$-Total Time $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-Approximation Algorithms 34
  - 2.3.1 $(1, 2, \lceil 2/\epsilon \rceil)$-Locally Persevering Emulator of Size $\tilde{O}(n^{3/2})$ . . . 35
  - 2.3.2 Maintaining Distances Using Monotone Even-Shiloach Tree 39
  - 2.3.3 From Approximate SSSP to Approximate APSP . . . 49
  - 2.3.4 Putting Everything Together . . . 55
- 2.4 Deterministic Decremental $(1+\epsilon)$-Approximate APSP with $O(mn \log n)$ Total Update Time . . . 57
  - 2.4.1 Deterministic Moving Centers Data Structure . . . 59
  - 2.4.2 Deterministic Center Cover Data Structure . . . 63
  - 2.4.3 Deterministic Fully Dynamic Algorithm . . . 76
- 2.5 Conclusion . . . 77

3 Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time **79**
- 3.1 Introduction . . . 80
- 3.2 Preliminaries . . . 82
- 3.3 Technical Overview . . . 83

3.4 From Approximate SSSP to Approximate Balls 92
3.4.1 Relation to Exact Balls 94
3.4.2 Properties of Approximate Balls 96
3.5 From Approximate Balls to Approximate SSSP 98
3.5.1 Algorithm Description 100
3.5.2 Running Time Analysis 101
3.5.3 Definitions of Values for Approximation Guarantee 101
3.5.4 Analysis of Approximation Guarantee 105
3.6 Putting Everything Together 113
3.6.1 Approximate SSSP 113
3.6.2 Approximate APSP 119
3.7 Conclusion 121

4 Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs 123
4.1 Introduction 124
4.2 Preliminaries 127
4.2.1 Problem Description 127
4.2.2 Definitions and Basic Properties 128
4.2.3 Algorithm Overview for s-t Reachability 130
4.2.4 Single-Source Shortest Paths 135
4.2.5 Strongly Connected Components 136
4.3 Single-Source Single-Sink Reachability 140
4.3.1 Algorithm Description 140
4.3.2 Correctness 143
4.3.3 Running Time Analysis 143
4.3.4 Extension to Single-Source Reachability 150
4.4 Approximate Shortest Path 152
4.4.1 Preliminaries 152
4.4.2 Algorithm Description 154
4.4.3 Correctness 155
4.4.4 Running Time 161
4.5 Faster Single-Source Reachability in Dense Graphs 164
4.5.1 Approximate Path Union Data Structure 164
4.5.2 Reachability via Center Graph 168
4.6 Conclusion 173

5 Sublinear-Time Maintenance of Breadth-First Spanning Trees in Partially Dynamic Networks 175
5.1 Introduction 175
5.2 Main Technical Idea 180
5.3 Incremental Algorithm 182
5.3.1 General Framework 182
5.3.2 Sequential model 187

5.3.3 Distributed Model 188
5.3.4 Removing the Connectedness Assumption 191
5.4 Decremental Algorithm 191
5.4.1 Analysis of Procedure for Repairing the Tree 193
5.4.2 Analysis of Decremental Distributed Algorithm 196
5.5 Conclusion and Open Problems 199

Bibliography **201**

CHAPTER 1

Introduction

1.1 Problem Statement

Computing shortest paths in graphs is a fundamental problem in computer science. If the graph for example models a transport network, then the shortest path from a node x to a node y in the graph gives the fastest way of travelling from location x to location y. To model more realistic scenarios one might allow the graph to change over time, where changing usually means inserting and deleting edges. In a transport network for example edges might be deleted when connections become temporarily unavailable because of congestion. A naive solution to handle such a dynamic scenario is to recompute shortest paths from scratch after each change in the graph by using a static algorithm. However it is usually more efficient to use special algorithms that are tailored to the dynamic setting. Other typical dynamic graph problems are minimum spanning tree [48, 52, 62, 71], connectivity [58, 60, 71, 77, 122], matching [15, 25, 56, 101, 102, 118], transitive closure [39, 61, 79, 81, 82, 108, 112, 114, 117], and strongly connected components [20, 21, 90, 109, 114]. Developing faster algorithms for dynamic shortest paths, in terms of provable upper bounds on the asymptotic running time, is the goal of this thesis.

A *fully dynamic all-pairs shortest paths (APSP) algorithm* is a data structure allowing the following operations for all pairs of nodes u and v of a graph G:

- Insert(u, v): Add the edge (u, v) to G
- Delete(u, v): Remove the edge (u, v) from G
- Query(u, v): Return the distance $d_G(u, v)$ from u to v in G

Each insertion or deletion of an edge is called an *update* of the graph. After every update the algorithm is allowed to spend some time to adapt to the change so that it can answer subsequent distance queries. The running time spent by the algorithm

after each update is called *update time* and the time needed to answer a query is called *query time*. Unless noted otherwise, we assume that query times are small, i.e., $O(1)$ or $O(\text{polylog } n)$. Very often the update time is *amortized* over a sequence of updates.

Within this framework, the general problem can be restricted in one or more of the following ways:

- We say that an algorithm is *partially dynamic* if it only allows one type of updates. *Incremental algorithms* only allow insertions of edges and *decremental algorithms* only allow deletions of edges. As we usually want to amortize over $\Theta(m)$ insertions or deletions (where m is the final or initial number of edges in the graph, respectively), partially dynamic algorithms are usually compared by their *total update time*, which is the sum of the running times spent after each update.

- In the *single-source shortest paths (SSSP)* problem we maintain the distances from a distinguished source node s to all other nodes.

- We say that an algorithm provides an (α, β)-*approximation* (where $\alpha \geq 1$ and $\beta \geq 0$) if upon a query for the distance from u to v it returns a distance estimate $\delta(u, v)$ such that $d_G(u, v) \leq \delta(u, v) \leq \alpha d_G(u, v) + \beta$. When $\beta = 0$, we usually simply say α-approximation instead of $(\alpha, 0)$-approximation. We will often obtain $(1 + \epsilon)$-approximations where $0 < \epsilon \leq 1$ is a parameter of the algorithm.

- Further variants of the dynamic shortest paths problem arise by distinguishing *directed and undirected* as well as *weighted and unweighted* graphs.

There are two main motivations for studying restricted versions of the dynamic shortest paths problem, as done in this thesis. The first motivation is to use specialized algorithms as building blocks for solving the more general problems, e.g., by using a decremental algorithm to obtain a fully dynamic algorithm. The second motivation is to gain efficiency over the more generalized algorithms, e.g., by avoiding bottlenecks inherent in maintaining distances exactly. In the design of our algorithms we also consider the goal of obtaining *deterministic* algorithms, which is orthogonal to running time improvements. Deterministic algorithms are very desirable for dynamic algorithms as they do not require to limit the knowledge of the adversary generating the sequence of updates and queries in any way.

This thesis is organized as follows. We start with the decremental approximate APSP problem in unweighted undirected graphs in Chapter 2. We obtain both a randomized $(1 + \epsilon, 2)$-approximation with total update time $\tilde{O}(n^{5/2})$[1] and a deterministic $(1 + \epsilon)$-approximation with total update time $O(mn \log n)$. Prior to this work there was a randomized $(1 + \epsilon)$-approximation with total update time $\tilde{O}(mn)$. Thus, our deterministic algorithm matches this running time and our randomized algorithm

[1] We use $\tilde{O}(\cdot)$ notation to hide logarithmic factors.

improves it at the cost of a small additive error. We then shift the focus to approximate SSSP. In Chapter 3 we develop an algorithm for weighted undirected graphs with almost linear total update time, which is optimal up to subpolynomial factors. We also extend our technique to obtain a new trade-off between the approximation guarantee and the total update time for decremental APSP. In Chapter 4 we consider decremental approximate SSSP in the more general setting of weighted directed graphs. We obtain a $(1+\epsilon)$-approximation with a total update time of $O(mn^{9/10})$. This is the first algorithm that breaks the $O(mn)$ bound, even for the simpler single-source reachability problem. In Chapter 5 we go back to unweighted undirected graphs, but this time in the *incremental* setting. We obtain a simple *deterministic* algorithm with a total update time of $O(m^{3/2}n^{1/2})$. We furthermore extend the technique to both the incremental and the decremental approximate SSSP problem in the CONGEST model of distributed computing.

1.2 Related Work

We discuss related work in detail in the individual chapters of this thesis. Here we only give a big overview over running times of algorithms for fully dynamic APSP (Table 1.1), decremental APSP (Table 1.2), and decremental SSSP (Table 1.3). It can be seen that most algorithms have an amortized update time. Furthermore, there has been a shift from exact solutions to approximate solutions. To gain some intuition we highlight the following three results:

- *Fully dynamic APSP:* The algorithm of Demetrescu and Italiano [36] (with a modification of Thorup [121]) solves the fully dynamic APSP problem on directed graphs with arbitrary real edge weights. The algorithm is deterministic and has an amortized running time of $\tilde{O}(n^2)$ *per update* and constant query time.

- *Decremental APSP:* Bernstein [23] gave a decremental algorithm for maintaining $(1+\epsilon)$-approximate APSP in weighted directed graphs. The algorithm is randomized and has a total update time of $\tilde{O}(mn \log W)$ and constant query time, if the edge weights are integers from 1 to W.

- *Decremental SSSP:* Bernstein and Roditty [24] gave a decremental algorithm for the $(1+\epsilon)$-approximate SSSP problem in unweighted undirected graphs. The algorithm is randomized and has a total update time of $O(n^{2+o(1)})$ and constant query time.

In the tables below we have not explicitly listed results for the incremental model as most decremental algorithms also work in the incremental model with slight modifications. To the best of our knowledge the first incremental algorithms were given by Ausiello et al. [9, 10].

Update time	Amortization	Query time	Approximation	Randomization	Graph type	Weights	Reference
$O(n^{9/7} \log S)$ [(a)]	amortized	$O(1)$	exact	deterministic	planar directed	$\in \mathbb{Z}$	[59]
$\tilde{O}(n^{2.5}\sqrt{W})$	amortized	$O(1)$	exact	deterministic	directed	$\in \{1, 2, \ldots, W\}$	[79]
$O(n^2 \log(nW)/\epsilon^2)$	amortized	$O(1)$	$1+\epsilon$	deterministic	directed	$\in \{1, 2, \ldots, W\}$	[79]
$\tilde{O}(n^2)$	amortized	$O(1)$	$2+\epsilon$	deterministic	directed	$\in \mathbb{Z}^{\succ 0}$	[79]
$\tilde{O}(n^{2.5}\sqrt{S})$ [(b)]	amortized	$O(1)$	exact	deterministic	directed	$\in \mathbb{R}$	[37]
$\tilde{O}(n^2)$	amortized	$O(1)$	exact	deterministic	directed	$\in \mathbb{R}^{\succeq 0}$	[36]
$\tilde{O}(n^2)$	amortized	$O(1)$	exact	deterministic	directed	$\in \mathbb{R}$	[121]
$\tilde{O}(n^{2.75})$	worst-case	$O(1)$	exact	deterministic	directed	$\in \mathbb{R}^{\succeq 0}$	[124]
$\tilde{O}(m\sqrt{n})$	amortized	$O(n^{3/4})$	exact	randomized	directed	unweighted	[115]
$\tilde{O}(mn/t)$ [(c)]	amortized	$O(t)$ [(c)]	$1+\epsilon$	randomized	undirected	unweighted	[113]
$O(n^{1.932})$	worst-case	$O(n^{1.288})$	exact	randomized	directed	unweighted	[119]
$\tilde{O}(m \log W)$ [(d)]	amortized	$O(\log \log \log n)$	$2+\epsilon$	randomized	undirected	$\in \{1, 2, \ldots, W\}$	[22]
$\tilde{O}(mn/t)$ [(e)]	amortized	$O(t)$ [(e)]	$1+\epsilon$	deterministic	undirected	unweighted	**Chapter 2**
$\tilde{O}(m^{1/2}n^{1/k})$ [(e)]	amortized	$O(k^2\rho^2)$ [(e)]	$2^{O(\rho k)}$ [(e)]	randomized	undirected	unweighted	[3]

[(a)] S equals the sum of the absolute values of all negative edge weights
[(b)] Every edge gets at most S different weights over all updates
[(c)] $t \leq \sqrt{m}$
[(d)] R equals the ratio between the heaviest and the lightest edge weight
[(e)] $t \leq \sqrt{n}$
[(f)] $k \geq 1$ and $\rho = 1 + \lceil \log n^{1-1/k} / \log (m/n^{1-1/k}) \rceil$

Table 1.1: Running times of fully dynamic all-pairs shortest paths algorithms (n is the number of nodes, m is the number of edges). For simplicity we assume that ϵ is a constant.

Total update time	Query time	Approximation	Randomization	Graph type	Weights	Reference
$\tilde{O}(mn^2/t + mn)$	$O(t)$	exact	randomized	directed	unweighted	[61]
$\tilde{O}(n^3 S)$ [(a)]	$O(1)$	exact	randomized	directed	$\in \mathbb{R}$	[37]
$O(n^3)$	$O(1)$	exact	randomized	directed	unweighted	[16]
$O(n^2\sqrt{m}/\epsilon)$	$O(1)$	$1+\epsilon$	randomized	directed	unweighted	[16]
$\tilde{O}(n^{16/9}m^{1/3} + mn)$	$O(1)$	3	randomized	undirected	unweighted	[17]
$\tilde{O}(n^{24/13}m^{3/13} + mn)$	$O(1)$	5	randomized	undirected	unweighted	[17]
$\tilde{O}(n^{16/9}m^{7/27} + mn)$	$O(1)$	7	randomized	undirected	unweighted	[17]
$\tilde{O}(mn)$	$O(1)$	$1+\epsilon$	randomized	undirected	unweighted	[115]
$\tilde{O}(n^{2+1/k})$ [(b)]	$O(k)$ [(e)]	$2k-1+\epsilon$	randomized	undirected	unweighted	[24]
$\tilde{O}(mn \log W)$	$O(1)$	$1+\epsilon$	randomized	directed	weighted	[23]
$\tilde{O}(n^{2.5})$	$O(1)$	$(1+\epsilon, 2)$	randomized	undirected	unweighted	**Chapter 2**
$\tilde{O}(mn)$	$O(1)$	$1+\epsilon$	deterministic	undirected	unweighted	**Chapter 2**
$O(n^{2.5+o(1)})$	$O(1)$	$(1+\epsilon, 2)$	randomized	undirected	unweighted	[2]
$\tilde{O}(mn^{1/k})$ [(c)]	$O(k\rho)$ [(c)]	$2^{O(\rho k)}$ [(c)]	randomized	undirected	unweighted	[3]
$O(m^{1+1/k+o(1)} \log^2 W)$ [(d)]	$O(k^k)$ [(d)]	$(2+\epsilon)^k - 1$ [(d)]	randomized	undirected	weighted	**Chapter 3**

[(a)] Every edge gets at most S different weights over all updates
[(b)] $2 \le k \le \log n$
[(c)] $k \ge 1$ and $\rho = 1 + \lceil \log n^{1-1/k} / \log(m/n^{1-1/k}) \rceil$
[(d)] $2 \le k \le \log n$

Table 1.2: Running times of decremental all-pairs shortest paths algorithms (n is the number of nodes, m is the number of edges). For simplicity we assume that ϵ is a constant.

Total update time	Query time	Approximation	Randomization	Graph type	Weights	Reference
$O(mn)$	$O(1)$	exact	deterministic	undirected	unweighted	[49]
$O(mn)$	$O(1)$	exact	deterministic	directed	unweighted	[61]
$O(mnW)$	$O(1)$	exact	deterministic	directed	$\in \{1, 2, \ldots, W\}$	[79]
$O(n^{2+o(1)})$	$O(1)$	$1+\epsilon$	randomized	undirected	unweighted	[24]
$O(m^{1+o(1)})$	$O(1)$	$1+\epsilon$	randomized	undirected	weighted	**Chapter 3**
$O(mn^{0.9+o(1)} \log W)$	$O(1)$	$1+\epsilon$	randomized	directed	weighted	**Chapter 4**

Table 1.3: Running times of decremental single-source shortest paths algorithms (n is the number of nodes, m is the number of edges). For simplicity we assume that ϵ is a constant.

1.3 Preliminaries

We will introduce necessary notation and concepts in the succeeding chapters. Nevertheless two concepts are used so frequently that we discuss them here separately. The first such concept is the following algorithmic primitive for maintaining shortest paths trees up to bounded depth.

Theorem 1.3.1 (Even-Shiloach tree [49, 61, 79])**.** *There is a decremental algorithm, called* Even-Shiloach tree *(short: ES-tree), that, given a weighted directed graph G undergoing edge deletions with positive integer edge weights, a source node s, and a parameter $D \geq 1$, maintains a shortest paths tree from s and the corresponding distances up to depth D with total update time $O(mD)$, i.e., the algorithm maintains $d_G(s, v)$ and the parent of v in the shortest paths tree for every node v such that $d_G(s, v) \leq D$. By reversing the edges of G it can also maintain the distance from v to s for every node v in the same time.*

Note that if we want to use the ES-tree to maintain a full shortest paths tree (i.e., containing all shortest paths from the source), then we have to set D equal to the maximum (finite) distance from the source. If the graph is unweighted, then it is sufficient to set $D = n - 1$ and we can thus maintain a full shortest paths tree with total update time $O(mn)$. If the graph is weighted and W is the maximum edge weight, then the maximum distance might be as large as $(n-1)W$ and the algorithm then takes time $O(mnW)$, which is less efficient than recomputation from scratch even if $W = n$. Similarly, if we want to maintain a shortest paths tree up to $h \leq n-1$ hops (containing all shortest paths with at most h edges), the algorithm above takes time $O(mh)$ in unweighted graphs and $O(mhW)$ in weighted graphs. Using a scaling technique [22, 23, 95], the running time in weighted graphs can be reduced if we allow approximation: we can maintain a $(1+\epsilon)$-approximate shortest paths tree containing all shortest paths up to h hops in total time $\tilde{O}(mn \log W/\epsilon)$.

The second concept used repeatedly in our algorithms is sampling nodes or edges at random. We say that an event happens *with high probability (whp)* if it happens with probability at least $1 - 1/n^c$, for some constant c. All our randomized algorithms will be correct whp against an oblivious adversary who fixes its sequence of updates and queries before the algorithm is initialized, revealing its choices to the algorithm one after the other. It is well-known, and exploited by many other algorithms for dynamic shortest paths and reachability, that by sampling a set of nodes with a sufficiently large probability we can guarantee that certain sets of nodes contain at least one of the sampled nodes. To the best of our knowledge, the first use of this technique in graph algorithms goes back to Ullman and Yannakakis [127].

Lemma 1.3.2. *Let T be a set of size t and let $S_1, S_2, \ldots, S_k$ be subsets of T of size at least q. Let U be a subset of T that was obtained by choosing each element of T independently with probability $p = (a \ln(kt))/q$, for some parameter a. Then, for every $1 \leq i \leq k$, the set S_i contains a node of U with high probability (whp), i.e., probability at least $1 - 1/t^a$, and the size of U is $O((t \log(kt))/q)$ in expectation.*

Proof. The bound on the size of U simply follows from the linearity of expectation. For every $1 \leq i \leq k$ let E_i be the event that $S_i \cap U = \emptyset$, i.e., that S_i contains no node of U. Furthermore, let E be the event that there is a set S_i that contains no node of U. Note that $E = \bigcup_{1 \leq i \leq k} E_i$.

We first bound the probability of the event E_i for $1 \leq i \leq k$. The size of $S_i \cap U$ is determined by a Bernoulli trial with success probability p. The probability that $|S_i \cap U| = 0$ is therefore given by

$$\Pr(E_i) = \left(1-p\right)^{|S_i|} \leq \left(1-p\right)^{q} = \left(1 - \frac{a \ln(kt)}{q}\right)^{q} \leq \frac{1}{e^{a \ln(kt)}} = \frac{1}{k^a t^a}.$$

Here we use the well-known inequality $(1 - 1/y)^y \leq 1/e$ that holds for every $y > 0$. Now we simply apply the union bound twice and get

$$\Pr(E) = \Pr\left(\bigcup_{1 \leq i \leq k} E_i\right) \leq \sum_{1 \leq i \leq k} \Pr(E_i) \leq \sum_{1 \leq i \leq k} \left(\frac{1}{kt}\right)^a = k \frac{1}{k^a t^a} \leq \frac{1}{t^a}. \qquad \square$$

We now sketch one example of how we intend to use Lemma 1.3.2 for dynamic graphs. Consider an unweighted graph G undergoing edge deletions. Suppose that we want the following condition to hold for every pair of nodes x and y with probability at least $1 - 1/n$ in *all* versions of G (i.e., initially and after each deletion): if $d_G(x, y) \geq q$, then there is a shortest path from x to y that contains a node in U. We apply Lemma 1.3.2 as follows. The set T is the set of nodes of G and has size n. The sets $S_1, S_2, \ldots, S_k$ are obtained as follows: for every version of G (i.e., after each deletion) and every pair of nodes x and y such that $d_G(x, y) \geq q$, we define a set S_i that contains the nodes on the first shortest path from x to y (for an arbitrary, but fixed order on the paths). As the graph undergoes edge deletions, there are at most $m \leq n^2$ versions of the graph. Furthermore, there are n^2 pairs of nodes. Therefore we have $k \leq n^4$ such sets. Thus, for the property above to hold with probability $1 - 1/n$, we simply have to sample each node with probability $(\ln kt)/q = (\ln n^5)/q = (5 \ln n)/q$ by Lemma 1.3.2.

CHAPTER 2

Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization

We study dynamic $(1+\epsilon)$-approximation algorithms for the all-pairs shortest paths problem in unweighted undirected n-node m-edge graphs under edge deletions. The fastest algorithm for this problem is a randomized algorithm with a total update time of $\tilde{O}(mn/\epsilon)$ and constant query time by Roditty and Zwick [113]. The fastest deterministic algorithm is from a 1981 paper by Even and Shiloach [49]; it has a total update time of $O(mn^2)$ and constant query time. We improve these results as follows:

(1) We present an algorithm with a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time that has an additive error of 2 in addition to the $1+\epsilon$ multiplicative error. This beats the previous $\tilde{O}(mn/\epsilon)$ time when $m = \Omega(n^{3/2})$. Note that the additive error is *unavoidable* since, even in the *static* case, an $O(n^{3-\delta})$-time (a so-called *truly subcubic*) combinatorial algorithm with $1+\epsilon$ multiplicative error cannot have an additive error less than $2-\epsilon$, unless we make a major breakthrough for Boolean matrix multiplication [41] and many other long-standing problems [128].

The algorithm can also be turned into a $(2+\epsilon)$-approximation algorithm (without additive error) with the same time guarantees, improving the recent $(3+\epsilon)$-approximation algorithm with $\tilde{O}(n^{5/2+O(\sqrt{\log(1/\epsilon)/\log n})})$ running time of Bernstein and Roditty [24] in terms of both approximation and time guarantees.

(2) We present a deterministic algorithm with a total update time of $\tilde{O}(mn/\epsilon)$ and a query time of $O(\log\log n)$. The algorithm has a multiplicative error of $1+\epsilon$ and gives the first improved deterministic algorithm since 1981. It also answers an open question raised by Bernstein [23]. The deterministic algorithm can be turned into a deterministic fully dynamic $(1+\epsilon)$-approximation with an amortized update time of $\tilde{O}(mn/(\epsilon t))$ and a query time of $\tilde{O}(t)$ for every $t \leq \sqrt{n}$.

In order to achieve our results, we introduce two new techniques: (1) A *monotone Even-Shiloach tree* algorithm which maintains a bounded-distance shortest-paths tree on a certain type of emulator called *locally persevering emulator*. (2) A derandomization technique based on *moving Even-Shiloach trees* as a way to derandomize the standard random set argument. These techniques might be of independent interest.

2.1 Introduction

Dynamic graph algorithms is one of the classic areas in theoretical computer science with a countless number of applications. It concerns maintaining properties of dynamically changing graphs. The objective of a dynamic graph algorithm is to efficiently process an online sequence of update operations, such as edge insertions and deletions, and query operations on a certain graph property. It has to quickly maintain the graph property despite an *adversarial* order of edge deletions and insertions. Dynamic graph problems are usually classified according to the types of updates allowed: *decremental* problems allow only deletions, *incremental* problems allow only insertions, and *fully dynamic* problems allow both.

2.1.1 The Problem

We consider the *decremental all-pairs shortest paths* (APSP) problem where we wish to maintain the distances in an undirected unweighted graph under a sequence of the following delete and distance query operations:

- Delete(u, v): delete edge (u, v) from the graph, and
- Distance(x, y): return the distance between node x and node y in the current graph G, denoted by $d_G(x, y)$.

We use the term *single-source shortest paths* (SSSP) to refer to the special case where the distance query can be done only when $x = s$, for a pre-specified *source node* s. The efficiency is judged by two parameters: *query time* denoting the time needed to answer *each* distance query, and *total update time* denoting the time needed to process *all* edge deletions. The running time will be in terms of n, the number of nodes in the graph, and m, the number of edges *before* any deletion. We use $\tilde{O}$-notation to hide an $O(\text{polylog}\, n)$ term. When it is clear from the context, we use "time" instead of "total update time", and, unless stated otherwise, the query time

is $O(1)$. One of the main focuses of this problem in the literature, which is also the goal in this chapter, is to *optimize the total update time* while keeping the query time and *approximation guarantees* small. We say that an algorithm provides an (α, β)-*approximation* if the distance query on nodes x and y on the current graph G returns an estimate $\delta(x, y)$ such that $d_G(x, y) \leq \delta(x, y) \leq \alpha d_G(x, y) + \beta$. We call α and β *multiplicative* and *additive errors*, respectively. We are particularly interested in the case where $\alpha = 1 + \epsilon$, for an arbitrarily small constant $\epsilon > 0$, β is a small constant, and the query time is constant or near-constant.

Previous Results Prior to our work, the best total update time for *deterministic* decremental APSP algorithms was $\tilde{O}(mn^2)$ by one of the earliest papers in the area from 1981 by Even and Shiloach [49]. The fastest *exact randomized* algorithms are the $\tilde{O}(n^3)$-time algorithms by Demetrescu and Italiano [37] and Baswana, Hariharan, and Sen [16]. The fastest *approximation* algorithm is the $\tilde{O}(mn)$-time $(1 + \epsilon, 0)$-approximation algorithm by Roditty and Zwick [113]. If we insist on an $O(n^{3-\delta})$ running time, for some constant $\delta > 0$, Bernstein and Roditty [24] obtain an $\tilde{O}(n^{2+1/k+O(1/\sqrt{\log n})})$-time $(2k - 1 + \epsilon, 0)$-approximation algorithm, for any integer $k \geq 2$, which gives, e.g., a $(3 + \epsilon, 0)$-approximation guarantee in $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ time. All these algorithms have an $O(1)$ worst case query time. See Section 2.1.4 for more detail and other related results.

2.1.2 Our Results

We present improved randomized and deterministic algorithms. Our deterministic algorithm provides a $(1 + \epsilon, 0)$-approximation and runs in $\tilde{O}(mn/\epsilon)$ total update time. Our randomized algorithm runs in $\tilde{O}(n^{5/2}/\epsilon)$ time and can guarantee both $(1 + \epsilon, 2)$- and $(2 + \epsilon, 0)$-approximations. Table 2.1 compares our results with previous results. In short, we make the following improvements over previous algorithms (further discussions follow).

- The total running time of deterministic algorithms is improved from Even and Shiloach's $\tilde{O}(mn^2)$ to $\tilde{O}(mn)$ (at the cost of $(1 + \epsilon, 0)$-approximation and $O(\log \log n)$ query time). This is the first improvement since 1981.

- For $m = \omega(n^{3/2})$, the total running time is improved from Roditty and Zwick's $\tilde{O}(mn/\epsilon)$ to $\tilde{O}(n^{5/2}/\epsilon)$, at the cost of an additive error of two, which appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be treated as a multiplicative error of $O(\epsilon)$) and is *unavoidable* (as discussed below).

- Our $(2 + \epsilon, 0)$-approximation algorithm improves the algorithm of Bernstein and Roditty in terms of both total update time and approximation guarantee. The multiplicative error of $2 + \epsilon$ is essentially the best we can hope for, if we do not want any additive error.

Reference	Total Running Time	Approximation	Deterministic?
[49]	$\tilde{O}(mn^2)$	Exact	Yes
Here	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	Yes
[16, 37]	$\tilde{O}(n^3)$	Exact	No
[113]	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	No
Here	$\tilde{O}(n^{5/2}/\epsilon)$	$(1+\epsilon, 2)$	No
[24]	$\tilde{O}(n^{5/2+\sqrt{\log(6/\epsilon)}/\sqrt{\log n}})$	$(3+\epsilon, 0)$	No
Here	$\tilde{O}(n^{5/2}/\epsilon)$	$(2+\epsilon, 0)$	No

Table 2.1: Comparisons between our and previous algorithms that are closely related. For details of these and other results see Section 2.1.4. All algorithms, except our deterministic algorithm, have $O(1)$ query time. Our deterministic algorithm has $O(\log\log n)$ query time.

To obtain these algorithms, we present two novel techniques, called *Moving Even-Shiloach Tree* and *Monotone Even-Shiloach Tree*, based on a classic technique of Even and Shiloach [49]. These techniques are reviewed in Section 2.1.3.

Improved Deterministic Algorithm In 1981, Even and Shiloach [49] presented a deterministic decremental SSSP algorithm for undirected, unweighted graphs with a total update time of $O(mn)$ over all deletions. By running this algorithm from n different nodes, we get an $O(mn^2)$-time decremental algorithm for APSP. No progress on deterministic decremental APSP has been made since then. Our algorithm achieves the first improvement over this algorithm, at the cost of a $(1+\epsilon, 0)$-approximation guarantee and $O(\log\log n)$ query time. (Note that our algorithm is also faster than the current fastest randomized algorithm [113] by a $\log n$ factor.) Our deterministic algorithm also answers a question recently raised by Bernstein [23] which asks for a deterministic algorithm with a total update time of $\tilde{O}(mn/\epsilon)$. As pointed out in [23] and several other places, this question is important due to the fact that deterministic algorithms can deal with an *adaptive offline adversary* (the strongest adversary model in online computation [19, 26]) while the randomized algorithms developed so far assume an *oblivious adversary* (the weakest adversary model) where the order of edge deletions must be fixed before an algorithm makes random choices. Our deterministic algorithm answers exactly this question. Using known reductions, we also obtain a deterministic fully dynamic $(1+\epsilon)$-approximation with an amortized running time of $\tilde{O}(mn/(\epsilon t))$ per update and a query time of $\tilde{O}(t)$ for every $t \leq n$.

Improved Randomized Algorithm Our aim is to improve the $\tilde{O}(mn)$ running time of Roditty and Zwick [113] to so-called *truly subcubic time*, i.e., $O(n^{3-\delta})$ time for some constant $\delta > 0$, a running time that is highly sought of in many problems (e.g., [111, 128, 129]). Note, however, that this improvement has to come at the cost of worse approximation:

Fact 2.1.1 ([41, 128]). *For any $\alpha \geq 1$ and $\beta \geq 0$ such that $2\alpha + \beta < 4$, there is* no *combinatorial (α, β)-approximation algorithm, not even a static one, for APSP on unweighted undirected graphs that is truly subcubic, unless we make a major breakthrough on many long-standing open problems, such as a combinatorial Boolean matrix multiplication and triangle detection.*

This fact can be explained as follows. Due to a reduction by Dor, Halperin, and Zwick [41], a combinatorial[1] algorithm for APSP, even a $(2 - \epsilon, 0)$-approximation or $(1 + \epsilon, 1)$-approximation one[2], with running time $O(n^{3-\delta})$, for any $\delta > 0$, will imply a combinatorial algorithm for *Boolean matrix multiplication* with the same running time, another breakthrough result. Further, due to Vassilevska Williams and Williams [128, Theorem 1.3], the $O(n^{3-\delta})$-time combinatorial algorithm will imply breakthrough results for a few other problems. Since combinatorial dynamic algorithms can be used to solve static APSP, the same argument applies. In particular, the additive error of two in our $(1 + \epsilon, 2)$-approximation algorithm is unavoidable if we wish to get a $O(n^{1-\delta})$ running time (a so-called *truly subcubic* time) and keep a small multiplicative error of $1 + \epsilon$. For the same reason, a multiplicative error of two in our $(2 + \epsilon, 0)$-approximation algorithm is also unavoidable. Similarly, the running time of our deterministic algorithm cannot be improved further unless we allow larger additive or multiplicative errors.

Roditty and Zwick [115] also showed a similar relation for decremental exact SSSP. In weighted graphs, lower bounds can be obtained even for non-combinatorial algorithms by assuming the hardness of all-pairs shortest-paths computation [1, 115]. Very recently (after the preliminary version [65] of this work appeared), Henzinger et al. [70] showed that Fact 2.1.1 holds even for *non-combinatorial* algorithms assuming that there is no truly subcubic-time algorithm for a problem called *online Boolean matrix-vector multiplication*. Henzinger et al. [70] argue that refuting this assumption will imply the same breakthrough as mentioned in Fact 2.1.1 if the term "combinatorial algorithm" (which is not a well-defined term) is interpreted in a certain way (in particular if it is interpreted as "Strassen-like algorithm", as defined in [14], which captures all known fast matrix multiplication algorithms). Thus, the best approximation guarantee we can expect from truly subcubic algorithms is, e.g., a multiplicative or additive error of at least two. Our algorithms achieve essentially these *best approximation guarantees*: in $\tilde{O}(n^{5/2}/\epsilon)$ time, we get a $(1 + \epsilon, 2)$-approximation, and, if we do not want any additive error, we can get a $(2 + \epsilon, 0)$-approximation (see Theorem 2.3.22 and corollary 2.3.23 for the precise statements of these results).[3] We note that, prior to our work, Bernstein and Roditty's algorithm [24] can achieve, e.g., a $(3 + \epsilon, 0)$-approximation guarantee in $\tilde{O}(n^{5/2+O(\sqrt{1/\log n})})$ time. This result is improved by our $(2 + \epsilon, 0)$-approximation algorithm in terms of both time and

[1]The vague term "combinatorial algorithm" is usually used to refer to algorithms that do not use algebraic operations such as matrix multiplication.

[2]In general, the reduction of Dor et al. holds for any (α, β) approximation as long as $2\alpha + \beta < 4$.

[3]We note that there is still some room to eliminate the ϵ-term, i.e., to get a $(1, 2)$-approximation algorithm. But anything beyond this is unlikely to be possible.

approximation guarantees, and is far worse than our $(1+\epsilon, 2)$-approximation guarantee, especially when the distance is large. Also note that the running time of our $(1+\epsilon, 2)$-approximation algorithm improves the $\tilde{O}(mn)$ one of Roditty and Zwick [113] when $m = \omega(n^{3/2})$, except that our algorithm gives an additive error of two which is unavoidable and appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be counted as a multiplicative error of $O(\epsilon)$).

2.1.3 Techniques

Our results build on two previous algorithms. The first algorithm is the classic SSSP algorithm of Even and Shiloach [49] (with the more general analysis of King [79]), which we will refer to as *Even-Shiloach tree*. The second algorithm is the $(1+\epsilon, 0)$-approximation APSP algorithm of Roditty and Zwick [113]. We actually view the algorithm of Roditty and Zwick as a *framework* which runs several Even-Shiloach trees and maintains some properties while edges are deleted. We wish to alter the Roditty-Zwick framework but doing so usually makes it hard to bound the cost of maintaining Even-Shiloach trees (as we will discuss later). Our main technical contribution is the development of new variations of the Even-Shiloach tree, called *moving Even-Shiloach tree* and *monotone Even-Shiloach tree*, which are suitable for our modified Roditty-Zwick frameworks. Since there are many other algorithms that run Even-Shiloach trees as subroutines, it might be possible that other algorithms will benefit from our new Even-Shiloach trees as well.

Review of Even-Shiloach Tree The Even-Shiloach tree has two parameters: a root (or source) node s and the range (or depth) R. It maintains a shortest paths tree rooted at s and the distances between s and all other nodes in the dynamic graph, up to distance R (if the distance is more than R, it will be set to ∞). It has a query time of $O(1)$ and a total update time of $O(mR)$ over all deletions. The total update time crucially relies on the fact that the distance between s and any node v changes *monotonically*: it will increase at most R times before it exceeds R (i.e., from 1 to R). This "monotonicity" property heavily relies on the "decrementality" of the model, i.e., the distance between two nodes never decreases when we delete edges, and is easily destroyed when we try to use the Even-Shiloach tree in a more general setting (e.g., when we want to allow edge insertions or alter the Roditty-Zwick framework). Most of our effort in constructing both randomized and deterministic algorithms will be spent on recovering from the destroyed decrementality.

Monotone Even-Shiloach Tree for Improved Randomized Algorithms

The high-level idea of our randomized algorithm is to run an existing decremental algorithm of Roditty and Zwick [113] on a sparse *weighted* graph that approximates the distances in the original graph, usually referred to as an *emulator* (see Section 2.3.1 for more detail). This approach is commonly used in the static setting (e.g., [6, 12, 31, 33, 41, 44, 47, 125, 130]), and it was recently used for the first time in the decremental

setting by Bernstein and Roditty [24]. As pointed out by Bernstein and Roditty, while it is a simple task to run an existing APSP algorithm on an emulator in the static setting, doing so in the decremental setting is not easy since it will destroy the "decrementality" of the setting: when an edge in the original graph is deleted, we might have to *insert* an edge into the emulator. Thus, we cannot run decremental algorithms on an arbitrary emulator, because from the perspective of this emulator, we are not in a decremental setting.

Bernstein and Roditty manage to get around this problem by constructing an emulator with a special property[4]. Roughly speaking, they show that their emulator guarantees that *the distance between any two nodes changes* $\tilde{O}(n)$ *times*. Based on this simple property, they show that the $(2k-1,0)$-approximation algorithm of Roditty and Zwick [113] can be run on their emulator with a small running time. However, they *cannot* run the $(1+\epsilon,0)$-approximation algorithm of Roditty and Zwick on their emulator. The main reason is that this algorithm relies on a more general property of a graph under deletions: for any R between 1 and n, the distance between any two nodes changes at most R times *before it exceeds* R (i.e., it changes from 1 to R). They suggested to find an emulator with this more general property as a future research direction.

In our algorithm, we manage to run the $(1+\epsilon,0)$-approximation algorithm of Roditty and Zwick on our emulator, but in a *conceptually different* way from Bernstein and Roditty. In particular, we do not construct the emulator asked for by Bernstein and Roditty; rather, we show that there is a type of emulators such that, while edge insertions can occur often, their effect can be *ignored*. We then modify the algorithm of Roditty and Zwick to incorporate this ignorance. More precisely, the algorithm of Roditty and Zwick relies on the classic Even-Shiloach tree. We develop a simple variant of this classic algorithm called *monotone Even-Shiloach tree* that can handle restricted kinds of insertions and use it to replace the classic Even-Shiloach tree in the algorithm of Roditty and Zwick.

Our modification to the Even-Shiloach tree is as follows. Recall that the Even-Shiloach tree can maintain the distances between a specific node s and all other nodes, up to R, in $O(mR)$ total update time under edge deletions. This is because, for any node v, it has to do work $O(\deg(v))$ (the degree of v) only when the distance between s and v changes, which will happen at most R times (from 1 to R) in the decremental model. Thus, the total work on each node v will be $O(R\deg(v))$ which sums to $O(mR)$ in total. This algorithm does not perform well when there are edge insertions: one edge insertion could cause a *decrease* in the distance between s and v by as much as $\Omega(R)$, causing an additional $\Omega(R)$ distance changes. The idea of our monotone Even-Shiloach tree is extremely simple: *ignore distance decreases*! It is easy to show that the total update time of our algorithm remains the same $O(mR)$ as the classic one. The hard part is proving that it gives a good approximation when

[4]In fact, their emulator is basically identical to one used earlier by Bernstein [22], which is in turn a modification of a spanner developed by Thorup and Zwick [125, 126]. However, the properties they proved are entirely new.

run on an emulator. This is because it does not maintain the *exact* distances on an emulator anymore. So, even when the emulator gives a good approximate distance on the original graph, our monotone Even-Shiloach tree might not. Our monotone Even-Shiloach tree does not give any guarantee for the distances in the emulator, but we can show that it still approximates the distances in the original graph. Of course, this will *not* work on any emulator; but we can show that it works on a specific type of emulators that we call *locally persevering emulators*.[5] Roughly speaking, a locally persevering emulator is an emulator where, for any "nearby"[6] nodes u and v in the original graph, either

(1) there is a shortest path from u to v in the original graph that also appears in the emulator, or

(2) there is a path in the emulator that approximates the distance in the original graph and *behaves in a persevering way*, in the sense that all edges of this path are in the emulator since before the first deletion and their weights never decrease. We call the latter path a *persevering path*.

Once we have the right definition of a locally persevering emulator, proving that our monotone Even-Shiloach tree gives a good distance estimate is conceptually simple (we sketch the proof idea below). Our last step is to show that such an emulator exists and can be efficiently maintained under edge deletions. We show (roughly) that we can maintain an emulator, which $(1+\epsilon, 2)$-approximates the distances and has $\tilde{O}(n^{3/2})$ edges, in $\tilde{O}(n^{5/2}/\epsilon)$ total update time under edge deletions. By running the $\tilde{O}(mn)$-time algorithm of Roditty and Zwick on this emulator, replacing the classic Even-Shiloach tree by our monotone version, we have the desired $\tilde{O}(n^{5/2}/\epsilon)$-time $(1+\epsilon, 2)$-approximation algorithm. To turn this algorithm into a $(2+\epsilon, 0)$-approximation, we observe that we can check if two nodes are of distance one easily; thus, we only have to use our $(1+\epsilon, 2)$-approximation algorithm to answer a distance query when the distance between two nodes is at least two. In this case, the additive error of two can be treated as a multiplicative factor.

Proving the Approximation Guarantee of the Monotone Even-Shiloach Tree To illustrate why our monotone Even-Shiloach tree gives a good approximation when run on a locally persevering emulator, we sketch a result that is weaker and simpler than our main results; we show how to $(3, 0)$-approximate distances from a particular node s to other nodes. This fact easily leads to a $(3+\epsilon, 0)$-approximation $\tilde{O}(n^{5/2}/\epsilon)$-time algorithm, which gives the same approximation guarantee as the algorithm of Bernstein and Roditty [24], and is slightly faster and reasonably simpler.

[5] We remark that there are other emulators that can be maintained in the decremental setting, e.g., [8, 18, 22, 24, 45, 113, 125, 126]. We are the first to introduce the notion of locally persevering emulators and show that there is an emulator that has this property.

[6] Note that the word "nearby" will be parameterized by a parameter τ in the formal definition. So, formally, we must use the term (α, β, τ)-locally persevering emulator where α and β are multiplicative and additive approximation factors, respectively. See Section 2.3.1 for detail.

To achieve this, we use the following emulator which is a simple modification of the emulator of Dor et al. [41]: Randomly select $\tilde{\Theta}(\sqrt{n})$ nodes. At any time, the emulator consists of all edges incident to nodes of degree at most $\sqrt{n}$ and edges from each random node c to every node v of distance at most 2 from c with weight equal to the distance between v and c. When the distance exceeds 2, the edge is deleted from the emulator. It can be shown that this emulator can be maintained in $\tilde{O}(mn^{1/2}) = \tilde{O}(n^{5/2})$ time under edge deletions. Moreover, it is a $(3, 0)$-emulator with high probability, since for every edge (u, v), either

(i) (u, v) is in the emulator, or

(ii) there is a path $\langle u, c, v\rangle$ of length at most three, where c is a random node.

Observe further that if (ii) happens, then the path $\langle u, c, v\rangle$ is *persevering* (as in Item (2) above):

(ii') $\langle u, c, v\rangle$ must be in this emulator since before the first deletion, and the weights of the edges (u, c) and (c, v) have never decreased.

It follows that this emulator is locally persevering.[7] Now we show that when we run the monotone Even-Shiloach tree on the above emulator, it gives $(3, 0)$-approximate distances between s and all other nodes. Recall that the monotone Even-Shiloach tree maintains a distance estimate, say $\ell(v)$, between s and every node v in the emulator[8]. For every node v, the value of $\ell(v)$ is regularly updated, except that when the degree of a node drops to $\sqrt{n}$ and the resulting insertion of an edge, say (u, v), decreases the distance between v and s in the emulator; in particular, $\ell(v) > \ell(u) + w(u, v)$ where $w(u, v)$ is the weight of edge (u, v). A usual way to modify the Even-Shiloach tree for dealing with such an insertion [24] is to decrease the value of $\ell(v)$ to $\ell(u) + w(u, v)$. Our monotone Even-Shiloach tree will *not* do this and keeps $\ell(v)$ unchanged. In this case, we say that the node v and the edge (u, v) *become stretched*. In general, an edge (u, v) is *stretched* if $\ell(v) > \ell(u) + w(u, v)$ or $\ell(u) > \ell(v) + w(u, v)$, and a node is stretched if it is incident to a stretched edge. Two observations that we will use are

(O1) as long as a node v is stretched, it will not change $\ell(v)$, and

(O2) a stretched edge must be an inserted edge.

We will argue that $\ell(v)$ of every node v is at most three times its true distance to s in the original graph. To prove this for a stretched node v, we simply use the fact that this is true before v becomes stretched (by induction), and $\ell(v)$ has not changed since then (by (O1)). If v is not stretched, we consider a shortest path $\langle v, u_1, u_2, \dots, s\rangle$ from v to s in the original graph. We will prove that

$$\ell(v) \le \ell(u_1) + 3;$$

[7] We note that we are being vague here. To be formal, we later define the notion of (α, β, τ)-locally persevering emulator in Definition 2.3.2, and the emulator we just defined will be $(3, 0, 1)$-locally persevering.

[8] Here ℓ stands for "level" as $\ell(v)$ is the level of v in the breadth-first search tree rooted at s.

thus, assuming that $\ell(u_1)$ satisfies the claim (by induction), $\ell(v)$ will satisfy the claim as well. To prove this, observe that if the edge (v, u_1) is contained in the emulator then we know that $\ell(v) \leq \ell(u_1) + 1$ (since v is not stretched), and we are done. Otherwise, by the fact that this emulator is locally persevering, we know that there is a path $\pi = \langle v, c, u_1 \rangle$ of length at most three in the emulator, and it is persevering (see Item (ii')). By (O2), *edges in π are not stretched.* It follows that

$$\ell(v) \leq \ell(c) + w(v, c) \leq \ell(u_1) + w(v, c) + w(c, u_1) \leq \ell(u_1) + 3,$$

where $w(v, c)$ and $w(c, u_1)$ are the current weights of edges (v, c) and (c, u_1), respectively, in the emulator. The claim follows.

In Section 2.3, we show how to refine the above argument to obtain a better guarantee, namely a $(1 + \epsilon, 2)$-approximation. The first refinement, which is simple, is extending the emulator above to a $(1 + \epsilon, 2)$-emulator. This is done by adding edges from every random node c to all nodes at distance at most $1/\epsilon$ from c. The next refinement, which is the main one, is the formal definition of (α, β, τ)-locally persevering emulators for some parameters α, β, and τ, and extending the proof outlined above to show that the monotone Even-Shiloach tree on such an emulator will give an $(\alpha + \beta/\tau, \beta)$-approximate distance estimate. We finally show that our simple $(1 + \epsilon, 2)$-emulator is a $(1, 2, 1/\epsilon)$-locally persevering emulator.

Moving Even-Shiloach Tree for Improved Deterministic Algorithms

Many distance-related algorithms in both dynamic and static settings use the following *randomized argument* as an important technique: if we select $\tilde{O}(h)$ nodes, called *centers*, uniformly at random, then every node will be at distance at most n/h from one of the centers with high probability [113, 127] (see Lemma 1.3.2). This even holds in the decremental setting (assuming an oblivious adversary). Like other algorithms, the Roditty-Zwick algorithm also heavily relies on this argument, which is the only reason why it is randomized. Our goal is to derandomize this argument. Specifically, for several different values of h, the Roditty-Zwick framework selects $\tilde{O}(h)$ random centers and uses the randomized argument above to argue that every node in a connected component of size at least n/h is *covered* by a center in the sense that it will always be within distance at most n/h from at least one center; we call this set of centers a *center cover*. It also maintains an Even-Shiloach tree of depth $R = O(n/h)$ from these h centers, which takes a total update time of $\tilde{O}(mR)$ for each tree and thus $\tilde{O}(hmR) = \tilde{O}(mn)$ over all trees. To derandomize the above process, we have two constraints:

(1) the center cover must be maintained (i.e., every node in a component of size at least n/h has a center nearby), and

(2) the number of centers (and thus Even-Shiloach trees maintained) must be $\tilde{O}(h)$ in total.

Maintaining these constraints in the *static* setting is fairly simple, as in the following algorithm.

Algorithm 2.1.2. *As long as there is a node v in a "big" connected component (i.e., of size at least n/h) that is not covered by any center, make v a new center.*

Algorithm 2.1.2 clearly guarantees the first constraint. The second constraint follows from the fact that the distance between any two centers is more than n/h. Since understanding the proof for guaranteeing the second constraint is important for understanding our *charging argument* later, we sketch it here. Let us label the centers by numbers $j = 1, 2, \ldots, h$. For a center with number j, we let B^j be a "ball" of radius $n/(2h)$; i.e., B^j is a set of nodes at distance at most $n/(2h)$ from center number j. Observe that B^j and $B^{j'}$ are disjoint for distinct centers j and j' since the distance between these centers is more than n/h Moreover, $|B^j| \geq n/(2h)$ since every center is in a big connected component. So, the number of balls (thus the number of centers) is at most $n/(n/(2h)) = 2h$. This guarantees the second constraint. Thus, we can guarantee both constraints in the static setting.

This, however, is not enough in the dynamic setting since *after* edge deletions, some nodes in big components might not be covered anymore and, if we keep repeating Algorithm 2.1.2, we might have to keep creating new centers to the point that the second constraint is violated. The key idea that we introduce to avoid this problem is to allow a center and the Even-Shiloach tree rooted at it to *move*. We call this a *moving Even-Shiloach tree* or *moving centers* data structure. Specifically, in the moving Even-Shiloach tree, we view a root (center) s *not* as a node, but as a *token* that can be placed on any node, and the task of the moving Even-Shiloach tree is to maintain the distance between the node that the root is placed on and all other nodes, up to distance R. We allow a *move operation* where we can move the root to a new node and the corresponding Even-Shiloach tree must be adjusted accordingly. To illustrate the power of the move operation, consider the following simple modification of Algorithm 2.1.2. (Later, we also have to modify this algorithm due to other problems that we will discuss next.)

Algorithm 2.1.3. *As long as there is a node v in a big connected component that is not covered by any center, we make it a center as follows. If there is a center in a small connected component, we move this center to v; otherwise, we open a new center at v.*

Algorithm 2.1.3 reuses centers and Even-Shiloach trees in small connected components[9] without violating the first constraint since nodes in small connected components do not need to be covered. The second constraint can also be guaranteed by showing that $|B^j| \geq n/(2h)$ for all j when we open a new center. Thus, by using moving Even-Shiloach trees, we can guarantee the two constraints above. We are, however, *not done yet.* This is because our new move operation also incurs a cost! *The most nontrivial idea in our algorithm is a charging argument to bound this cost.*

[9]We note the detail that we need a deterministic dynamic connectivity data structure [62, 71] to implement Algorithm 2.1.3. The additional cost incurred is negligible.

There are two types of cost. First, the *relocation cost* which is the cost of constructing a new breadth-first search tree rooted at the new location of the center. This cost can be bounded by $O(m)$ since we can construct a breadth-first search tree by running the static $O(m)$-time algorithm. Thus, it will be enough to guarantee that we do not move Even-Shiloach trees more than $O(n)$ times. In fact, this is already guaranteed in Algorithm 2.1.3 since we will *never* move an Even-Shiloach tree back to a previous node. The second cost, which is *much harder* to bound, is the *additional maintenance cost.* Recall that we can bound the total update time of an Even-Shiloach tree by $O(mR)$ because of the fact that the distance between its root (center) and each other node changes at most R times before exceeding R, by increasing from 1 to R. However, when we move the root from, say, a node u to its neighbor v, the distance between the new root v and some node, say x, might be smaller than the previous distance from u to x. In other words, *the decrementality property is destroyed.* Fortunately, observe that the distance change will be *at most one* per node when we move a tree to a neighboring node. Using a standard argument, we can then conclude that *moving a tree between neighboring nodes costs an additional distance maintenance cost of* $O(m)$. This motivates us to define the notion of *moving distance* to measure how far we move the Even-Shiloach trees in total. We will be able to bound the maintenance cost by $O(mn)$ if we can show that the total moving distance (summing over all moving Even-Shiloach trees) is $O(n)$. Bounding the total moving distance by $O(n)$ while having only $O(h)$ Even-Shiloach trees is the most challenging part in obtaining our deterministic algorithm. We do it by using a careful charging argument. We sketch this argument here. For more intuition and detail, see Section 2.4.

Charging Argument for Bounding the Total Moving Distance Recall that we denote the centers by numbers $j = 1, 2, \ldots, h$. We make a few modifications to Algorithm 2.1.3. The most important change is the introduction of the set C^j for each center j (which is the root of a moving Even-Shiloach tree). This will lead to a few other changes. The importance of C^j is that we will "charge" the moving cost of center j to nodes in C^j; in particular, we bound the total moving distance to be $O(n)$ by showing that the moving distance of center j can be bounded by $|C^j|$, and C^j and $C^{j'}$ are disjoint for distinct centers j and j'. The other important changes are the definitions of "ball" and "small connected component" which will now depend on C^j.

- We change the definition of B^j from a ball of radius $n/(2h)$ to a ball of radius $(n/(2h)) - |C^j|$.
- We redefine the notion of "small connected component" as follows: we say that a center j is in a small connected component if the connected component containing it has less than $(n/(2h)) - |C^j|$ nodes (instead of n/h nodes).

These new definitions might not be intuitive, but they are crucial for the charging argument. We also have to modify Algorithm 2.1.3 in a counter-intuitive way: the

most important modification is that we have to give up the nice property that the distance between any two centers is more than $n/(2h)$ as in Algorithms 2.1.2 and 2.1.3. In fact, we will *always* move a center out of a small connected component, and we will move it *as little as possible*, even though the new location could be near other centers. In particular, consider the deletion of an edge (u, v). It can be shown that there is *at most one* center j that is in a small connected component (according to the new definition), and this center j must be in the same connected component as u or v. Suppose that such a center j exists, and it is in the same connected component as u, say X. Then, we will move center j to v, which is just enough to move j out of component X (it is easy to see that v is the node outside of X that is nearest to j before the deletion). We will also update C^j by adding all nodes of X to C^j. This finishes the moving step, and it can be shown that there is no center in a small connected component now. Next, we make sure that every node is covered by opening a new center at nodes that are not covered, as in Algorithm 2.1.2. To conclude, our algorithm is as follows.

Algorithm 2.1.4. *Consider the deletion of an edge (u, v). Check if there is a center j that is in a "small" connected component X (of size less than $(n/(2h)) - |C^j|$). If there is such a j (there will be at most one such j), move it out of X to a new node which is the unique node in $\{u, v\} \setminus X$. After moving, execute the static algorithm as in Algorithm 2.1.2.*

To see that the total moving distance is $O(n)$, observe that when we move a center j out of component X in Algorithm 2.1.4, we incur a moving distance of at most $|X|$ (since we can move j along a path in X). Thus, we can always bound the total moving distance of center j by $|C^j|$. We additionally show that C^j and $C^{j'}$ are disjoint for different centers j and j'. So, the total moving distance over all centers is at most $\sum_j |C^j| \leq n$. We also have to bound the number of centers. Since we give up the nice property that centers are far apart, we cannot use the same argument to show that the sets B^j are disjoint and big (i.e., $|B^j| \geq n/(2h)$), as in Algorithm 2.1.3 and Algorithm 2.1.4. However, using C^j, we can still show something very similar: $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint for distinct j and j', and $|B^j \cup C^j| \geq n/(2h)$. Thus, we can still bound the number of centers by $O(h)$ as before.

2.1.4 Related Work

Dynamic APSP has a long history, with the first papers dating back to 1967 [93, 97][10]. It also has a tight connection with its *static* counterpart (where the graph does not change), which is one of the most fundamental problems in computer science: On the one hand, we wish to devise a dynamic algorithm that beats the naive algorithm where we recompute shortest paths *from scratch* using static algorithms after every deletion. On the other hand, the best we can hope for is to match

[10]The early papers [93, 97], however, were not able to beat the naive algorithm where we compute APSP from scratch after every change.

the total update time of decremental algorithms to the best running time of static algorithms. To understand the whole picture, let us first recall the current situation in the static setting. We will focus on combinatorial algorithms[11] since our and most previous decremental algorithms are combinatorial. Static APSP on unweighted undirected graphs can be solved in $O(mn)$ time by simply constructing a breadth-first search tree from every node. Interestingly, this algorithm is the fastest combinatorial algorithm for APSP (despite other fast non-combinatorial algorithms based on matrix multiplication). In fact, a faster combinatorial algorithm will be a *major breakthrough*, not just because computing shortest paths is a long-standing problem by itself, but also because it will imply faster algorithms for other long-standing problems, as stated in Fact 2.1.1.

The fact that the best static algorithm takes $O(mn)$ time means two things: First, the naive algorithm will take $O(m^2n)$ total update time. Second, the best total update time we can hope for is $O(mn)$. A result that is perhaps the first to beat the naive $O(m^2n)$-time algorithm is from 1981 by Even and Shiloach [49], for the special case of SSSP. Even and Shiloach actually studied decremental connectivity, but their main data structure gives an $O(mn)$ total update time with $O(1)$ query time for decremental SSSP; this implies a total update time of $O(mn^2)$ for decremental APSP. Roditty and Zwick [115] later provided evidence that the $O(mn)$-time decremental unweighted SSSP algorithm of Even and Shiloach is the fastest possible by showing that this problem is at least as hard as several natural static problems such as Boolean matrix multiplication and the problem of finding all edges of a graph that are contained in triangles. For the incremental setting, Ausiello, Italiano, Marchetti-Spaccamela, and Nanni [9] presented an $\tilde{O}(n^3)$-time APSP algorithm on unweighted directed graphs. (An extension of this algorithm for graphs with small integer edge weights is given in [10].) After that, many efficient fully-dynamic algorithms have been proposed (e.g., [37, 38, 50, 59, 79]). Subsequently, Demetrescu and Italiano [36] achieved a major breakthrough for the fully dynamic case: they obtained a fully dynamic deterministic algorithm for the weighted directed APSP problem with an amortized time of $\tilde{O}(n^2)$ *per update*, implying a total update time of $\tilde{O}(mn^2)$ over all deletions in the decremental setting, the same running time as the algorithm of Even and Shiloach. (Thorup [121] presented a small improvement of this result.) An amortized update time of $\tilde{O}(n^2)$ is essentially optimal if the distance matrix is to be explicitly maintained, as done by the algorithm of Demetrescu and Italiano [36], since each update operation may change $\Omega(n^2)$ entries in the matrix. Even for unweighted, undirected graphs, no faster algorithm is known. Thus, the $O(mn^2)$ total update time of Even and Shiloach *remains the best* for deterministic decremental algorithms, even on undirected unweighted graphs and if approximation is allowed.

For the case of randomized algorithms, Demetrescu and Italiano [37] obtained an exact decremental algorithm on weighted directed graphs with $\tilde{O}(n^3)$ total update time[12] (if weight increments are not considered). Baswana, Hariharan, and Sen [16]

[11] The vague term "combinatorial algorithm" is usually used to refer to algorithms that do not use algebraic operations such as matrix multiplication.

[12] This algorithm actually works in a much more general setting where each edge weight can assume

obtained an exact decremental algorithm on unweighted directed graphs with $\tilde{O}(n^3)$ total update time. They also obtained a $(1+\epsilon, 0)$-approximation algorithm with $\tilde{O}(m^{1/2}n^2)$ total update time. In [17], they improved the running time further on undirected unweighted graphs, at the cost of a worse approximation guarantee: they obtained approximation guarantees of $(3,0)$, $(5,0)$, $(7,0)$ in $\tilde{O}(mn^{10/9})$, $\tilde{O}(mn^{14/13})$, and $\tilde{O}(mn^{28/27})$ time, respectively. Roditty and Zwick [113] presented two improved algorithms for unweighted, undirected graphs. The first was a $(1+\epsilon, 0)$-approximate decremental APSP algorithm with constant query time and a total update time of $\tilde{O}(mn)$. This algorithm remains the current fastest. The second algorithm achieves a worse approximation bound of $(2k-1, 0)$, for any $2 \leq k \leq \log n$, but has the advantage of requiring less space ($O(m + n^{1+1/k})$). By modifying the second algorithm to work on an emulator, Bernstein and Roditty [24] presented the first truly subcubic algorithm which gives a $(2k-1+\epsilon, 0)$-approximation and has a total update time of $\tilde{O}(n^{1+1/k+O(1/\sqrt{\log n})})$. They also presented a $(1+\epsilon, 0)$-approximation $\tilde{O}(n^{2+O(1/\sqrt{\log n})})$-time algorithm for SSSP, which is the first improvement since the algorithm of Even and Shiloach. Very recently, Bernstein [23] presented a $(1+\epsilon, 0)$-approximation $\tilde{O}(mn \log W)$-time algorithm for the directed weighted case, where W is the ratio of the largest edge weight ever seen in the graph to the smallest such weight.

We note that the $(1+\epsilon, 0)$-approximation $\tilde{O}(mn)$-time algorithm of Roditty and Zwick matches the state of the art in the static setting; thus, it is essentially tight. However, by allowing additive error, this running time was improved in the static setting. For example, Dor, Halperin, and Zwick [41], extending the approach of Aingworth et al. [6], presented a $(1, 2)$-approximation for APSP in unweighted undirected graphs with a running time of $O(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$. Elkin [44] presented an algorithm for unweighted undirected graphs with a running time of $O(mn^{\rho} + n^2\zeta)$ that approximates the distances with a multiplicative error of $1+\epsilon$ and an additive error that is a function of ζ, ρ and ϵ. There is no decremental algorithm with additive error prior to our algorithm.

Subsequent Work Independent of our work, Abraham and Chechik [2] developed a randomized $(1+\epsilon, 2)$-approximate decremental APSP algorithm with a total update time of $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ and constant query time. This result is very similar to one of ours, except that the running time in [2] is slightly more than $\tilde{O}(n^{5/2})$. After the preliminary version of our work ([65]) appeared, we extended the randomized algorithm in this chapter and obtained the following two algorithms for APSP [63]: (i) a $(1+\epsilon, 2(1+2/\epsilon)^{k-2})$-approximation with total time $\tilde{O}(n^{2+1/k}(37/\epsilon)^{k-1})$ for any $2 \leq k \leq \log n$ (improving the time in this chapter with a larger additive error when $k \geq 3$), and (ii) a $(3+\epsilon)$-approximation with total time $\tilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$ (it is faster than the algorithm in this chapter for sparse graphs but causes more multi-

S different values. Note that the amortized time per update of this algorithm is $\tilde{O}(Sn)$, but this holds only when there are $\Omega(n^2)$ updates (see [37, Theorem 10]). Also note that the algorithm is randomized with one-sided error.

plicative error). These two algorithms heavily rely on the monotone Even-Shiloach tree introduced in this chapter. In the same paper, the monotone Even-Shiloach tree was also used in combination with techniques of Chapter 5 to obtain the first subquadratic-time algorithm for approximate SSSP. Very recently, we obtained an almost linear total update time for $(1+\epsilon)$-approximate SSSP in weighted undirected graphs (see Chapter 3), where the monotone Even-Shiloach tree again played a central role. We also obtained the first improvement over Even-Shiloach's algorithm for single-source reachability and approximate single-source shortest paths on *directed* graphs (see Chapter 4).

2.2 Background

2.2.1 Basic Definitions

In the following we give some basic notation and definitions.

Definition 2.2.1 (Dynamic graph)**.** *A* dynamic graph $\mathcal{G}$ *is a sequence of graphs* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *that share a common set of nodes* V*. The set of edges of the graph* G_i *(for* $0 \le i \le k$*) is denoted by* $E(G_i)$*. The number of nodes of* $\mathcal{G}$ *is* $n = |V|$ *and the initial number of edges of* $\mathcal{G}$ *is* $m = |E(G_0)|$*. The set of edges ever contained in* $\mathcal{G}$ *up to time* t *(where* $0 \le t \le k$*) is* $E_t(\mathcal{G}) = \cup_{0 \le i \le t} E(G_i)$*. A dynamic weighted graph* $\mathcal{H}$ *is a sequence of weighted graphs* $\mathcal{H} = (H_i, w_i)_{0 \le i \le k}$ *that share a common set of nodes* V*. For* $0 \le i \le k$ *and every edge* $(u, v) \in E(H_i)$*, the weight of* (u, v) *is given by* $w_i(u, v)$*.*

Let us clarify how a dynamic graph $\mathcal{G} = (G_i)_{0 \le i \le k}$ is processed by a dynamic algorithm. The dynamic graph $\mathcal{G}$ is a sequence of graphs picked by an adversary before the algorithm starts. In its initialization phase, the algorithm may process the initial graph G_0 and in the i-th update phase, the algorithm may process the graph G_i. At the beginning of the i-th update phase, the graph G_i is presented to the algorithm implicitly as the set of updates from G_{i-1} to G_i. The algorithm will, for example, be informed which edges were deleted from the graph. After the initialization phase and after each update phase, the algorithm has to be able to answer queries. In our case, these queries will usually be distance queries and the algorithm will answer them in constant or near-constant time. The *total update time* of the algorithm is the total time spent for processing the initialization and *all* k updates.

Definition 2.2.2 (Updates)**.** *For a dynamic graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *we say for an edge* (u, v) *that*

- (u, v) *is* deleted *at time* t *if* (u, v) *is contained in* G_{t-1} *but not in* G_t*.*
- (u, v) *is* inserted *at time* t *if* (u, v) *contained in* G_t *but not in* G_{t-1}*.*

For a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0 \le i \le k}$*, we additionally say for an edge* (u, v) *that*

- *the weight of* (u, v) *is* increased *at time* t *if* $w_t(u, v) > w_{t-1}(u, v)$ *(and* (u, v) *is contained in both* G_{t-1} *and* G_t*).*

- *the weight of* (u, v) *is* decreased *at time* t *if* $w_t(u, v) < w_{t-1}(u, v)$ *(and* (u, v) *is contained in both* G_{t-1} *and* G_t*).*

Every deletion, insertion, weight increase or weight decrease is called an update*. The* total number of updates up to time t *of a dynamic (weighted) graph* $\mathcal{G}$ *is denoted by* $\phi_t(\mathcal{G})$*.*

Definition 2.2.3 (Decremental graph)**.** *A* decremental graph $\mathcal{G}$ *is a dynamic graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *such that for every* $1 \le i \le k$ *there is exactly one edge deletion at time* i*. Note that* G_i *is the graph after the* i*-th edge deletion.*

By our definition decremental graphs are always unweighted. For a weighted version of this concept it would make sense to additionally allow edge weight increases. In a decremental graph $\mathcal{G} = (G_i)_{0 \le i \le k}$ we necessarily have $k \le m$ because every edge can be deleted only once. For decremental shortest paths algorithms the total update time usually does *not* depend on the number of deletions k. This is the case because of the amortization argument typically used for these algorithms. For this reason, it will often suffice for our purposes to bound $\phi_k(\mathcal{G})$ or $|E_k(\mathcal{G})|$ by numbers that do not depend on k.

We now formulate the approximate all-pairs shortest paths (APSP) problem we are trying to solve.

Definition 2.2.4 (Distance)**.** *The distance of a node* x *to a node* y *in a graph* G *is denoted by* $d_G(x, y)$*. If* x *and* y *are not connected in* G*, we set* $d_G(x, y) = \infty$*. In a weighted graph* (H, w) *the distance of* x *to* y *is denoted by* $d_{H,w}(x, y)$*.*

Definition 2.2.5. *An* (α, β)-approximate decremental all-pairs shortest paths (APSP) data structure *for a decremental graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *maintains, for all nodes* x *and* y *and all* $0 \le i \le k$*, an estimate* $\delta_i(x, y)$ *of the distance between* x *and* y *in* G_i*. After the* i*-th edge deletion (where* $0 \le i \le k$*), it provides the following operations:*

- Delete(u, v)*: Delete the edge* (u, v) *from* G_i*.*

- Distance(x, y)*: Return an estimate* $\delta_i(x, y)$ *of the distance between* x *and* r *in* G_i *such that* $d_{G_i}(x, y) \le \delta_i(x, y) \le \alpha d_{G_i}(x, y) + \beta$*.*

The total update time *is the total time needed for performing all* k *delete operations and the initialization and the* query time *is the worst-case time needed to answer a single distance query. The data structure is* exact *if* $\alpha = 1$ *and* $\beta = 0$*.*

Similarly, we define a data structure for decremental single-source shortest paths (SSSP). We incorporate two special requirements in this definition. First, we are interested in SSSP data structures that only need to work up to a certain distance

range[13] R^{d} from the source node which is specified by a parameter R^{d}. Second, we demand that the data structure tells us whenever a node leaves this distance range. The latter is a technical requirement that simplifies some of our proofs.

Definition 2.2.6. *An* (α, β)-approximate decremental single-source shortest paths (SSSP) data structure *with* source (or: root) node r *and* distance range parameter R^{d} *for a decremental graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *maintains, for every node* x *and all* $0 \le i \le k$*, an estimate* $\delta_i(x, r) \in \{0, 1, \dots, \lfloor \alpha R^d + \beta \rfloor, \infty\}$ *of the distance between* x *and* r *in* G_i*. After the* i*-th edge deletion (where* $0 \le i \le k$*), it provides the following operations:*

- Delete(u, v)*: Delete the edge* (u, v) *from* G_i *and return the set of all nodes* x *such that* $\delta_i(x, r) \le \alpha R^d + \beta$ *and* $\delta_{i+1}(x, r) > \alpha R^d + \beta$
- Distance(x)*: Return an estimate* $\delta_i(x, r)$ *of the distance between* x *and* r *in* G_i *such that* $\delta_i(x, r) \ge d_{G_i}(x, r)$ *and if* $d_{G_i}(x, r) \le R^d$*, then* $\delta_i(x, r) \le \alpha d_{G_i}(x, r) + \beta$*.*

The total update time *is the total time needed for performing all* k *delete operations and the initialization and the* query time *is the worst-case time needed to answer a single distance query. The data structure is* exact *if* $\alpha = 1$ *and* $\beta = 0$*.*

Finally, we define the remaining notions on graphs we will use.

Definition 2.2.7 (Degree)**.** *We say that* v is a neighbor of u *if there is an edge* (u, v) *in* G*. The* degree *of a node* u *in the graph* G*, denoted by* $\deg_G(u)$*, is the number of neighbors of* u *in* G*. The* dynamic degree *of a node* u *in a dynamic graph* $\mathcal{G}$ *is* $\deg_{\mathcal{G}}(u) = |\{(u, v) \mid (u, v) \in E_k(\mathcal{G})\}|$.

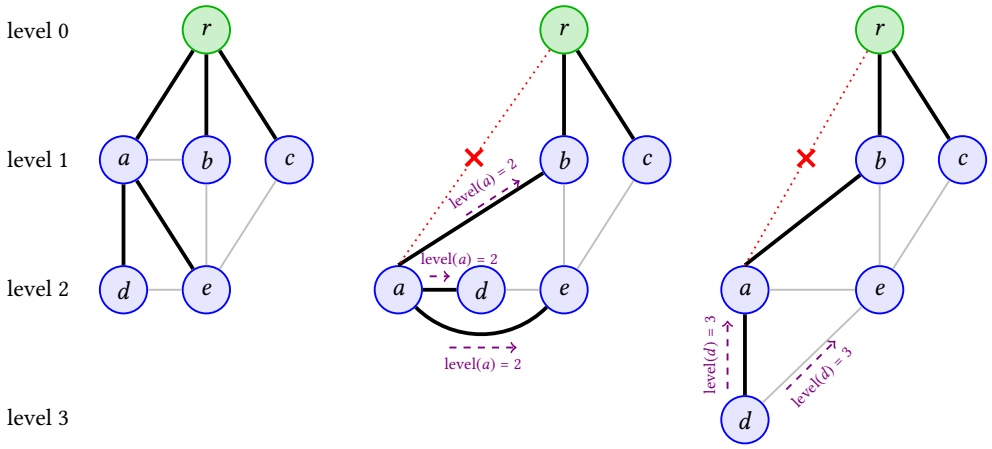
Definition 2.2.8 (Paths)**.** *Let* (H, w) *be a weighted graph and let* π *be a path in* (H, w) *. The number of nodes on the path* π *is denoted by* $|\pi|$ *and the total weight of the path (i.e., the sum of the weights of its edges) is denoted by* $w(\pi)$*.*

Definition 2.2.9 (Connected component)**.** *For every graph* G *and every node* x *we denote by* $\mathrm{Comp}_G(x)$ *the connected component of* x *in* G*, i.e., the set of nodes that are connected to* x *in* G*.*

2.2.2 Decremental Shortest-Path Tree Data Structure

The central data structure in dynamic shortest paths algorithms is the dynamic single-source shortest-paths tree introduced by Even and Shiloach, in short *ES-tree*. Even and Shiloach [49] developed this data structure for undirected, unweighted graphs. Later on, Henzinger and King [61] observed that it can be adapted to work on directed graphs and King [79] gave a modification for directed, weighted graphs with positive integer edge weights. In the following we review some important properties of this data structure.

[13] In this chapter, there are two related parameters R^{d} (introduced here) representing the "distance range" of an SSSP data structure (e.g., the Even-Shiloach tree described in Section 2.2.2) and R^{c} (which will be introduced in Section 2.2.3) representing the "cover range" of the center cover data structure.



(a) (b) (c)

Figure 2.1: Example of the view of the ES-tree as nodes talking to each other. (a) The ES-tree before the edge deletion. (b) After deleting the edge (r, a), the level of the node a changes to 2. The node a sends a message to all neighbors to inform them about this change. (c) This causes the node d to change its level and thus d sends a message to inform its neighbors. There are no other changes, so the new ES-tree is as in (c). Thus, there are 5 messages involved in constructing the new ES-tree, and Algorithm 2.1 shows that this process can be implemented in 5 time units.

We describe an ES-tree on dynamic weighted undirected graphs for a given root node r and a given distance range parameter R^{d}. The data structure can handle arbitrary edge deletions and weight increases. The data structure maintains, for every node v, a label $\ell(v)$, called the *level* of v. The level of v corresponds to the distance between v and the root r. Any node v whose distance to r is more than R^{d} has $\ell(v) = \infty$. Initially, the values of $\ell(v)$ can be computed in $\tilde{O}(m)$ time using, e.g., Dijkstra's algorithm. The level $\ell(v)$ implicitly implies the shortest-path tree since the parent of every node v is a node z such that $\ell(v) = \ell(z) + w(v, z)$. (Every node v such that $\ell(v) = \infty$ will not be in the shortest-paths tree.) Every deletion of an edge (u, v) possibly affects the levels of several nodes. The algorithm tries to adjust the levels of these nodes as follows.

Informal Description How the ES-tree handles deletions can be intuitively viewed as nodes in the input graph talking to each other as follows. Imagine that every node v in the input graph is a computing unit that tries to maintain its level $\ell(v)$ corresponding to its current distance to the root. It knows the levels of its neighbors and has to make sure that

$$\ell(v) = \min_{u} (\ell(u) + w(u, v)) \tag{2.1}$$

where the minimum is over all current neighbors u of v. When we delete an edge incident to v, the value of $\ell(v)$ might change. If this happens, *v sends a message* to each of its neighbors to inform about this change, since the levels of these nodes might have to change as well. Every neighbor of v then updates its level accordingly and if its level changes, it sends messages to its neighbors (including v), too. (See Figure 2.1 for an example.) An important point, which we will show soon, is that *we can implement the ES-tree in time proportional to the number of messages.* This means that when a node v's level is changed, we can bound the time we need to maintain the ES-tree by its current degree. Thus, the contribution of a node v to the running time to update the ES-tree after the i-th deletion or weight increase is $\deg_{G_i}(v)$ times its level change, i.e., $\min(R^{\mathrm{d}}, \ell_i(v)) - \min(R^{\mathrm{d}}, \ell_{i-1}(v))$ (the minimum is to avoid the case where $\ell_i(v) = \infty$). This intuitively leads to the following lemma.

Lemma 2.2.10 (King [79])**.** *The ES-tree is an exact decremental SSSP data structure for shortest paths up to a given length R^d. It has constant query time and in a decremental graph $\mathcal{G} = (G_i, w_i)_{0 \le i \le k}$, its total update time can be bounded by*

$$O\left(\phi_k(\mathcal{G}) + t_{\mathrm{SP}} + \sum_{1 \le i \le k} \sum_{v \in V} \deg_{G_i}(v) \left(\min(R^d, \ell_i(v)) - \min(R^d, \ell_{i-1}(v))\right)\right)$$

where t_{SP} is the time needed for computing a single-source shortest paths tree up to depth R^d and, for $0 \le i \le k$, $\ell_i(v)$ is the level of v after the ES-tree has processed the i-th deletion or weight increase.

Recall that $\phi_k(\mathcal{G})$ is the total number of updates (deletions and weight increases). Note that when the graph is unweighted, only deletions are allowed. In this case, the $\phi_k(\mathcal{G})$ term can be ignored. Lemma 2.2.10 can be simplified by using two specific bounds. These bounds are in fact what we need later in this chapter.

Corollary 2.2.11. *There is an exact decremental SSSP data structure for paths up to a given length R^d that has constant query time and in a decremental graph $\mathcal{G} = (G_i, w_i)_{0 \le i \le k}$ with source node r, its total update time can be bounded by*

$$O\left(\phi_k(\mathcal{G}) + t_{\mathrm{SP}} + \sum_{v \in V} \deg_{G_0}(v) \cdot \left(\min(R^d, d_{G_k}(v, r)) - \min(R^d, d_{G_0}(v, r))\right)\right)$$

and $O(mR^d)$ where t_{SP} is the time needed for computing a single-source shortest paths tree up to depth R^d and $d_{G_0}(v, r)$ is the initial distance of r to v and $d_{G_k}(v, r)$ is the distance of v to r after all k edge deletions.

The first bound in Corollary 2.2.11 is because, for every node v, we can use $\deg_{G_i}(v) \le \deg_{G_0}(v)$, and we can express the running time caused by v's level change in terms of its initial level ($\min(R^{\mathrm{d}}, d_{G_0}(v, r))$ and its final level ($\min(R^{\mathrm{d}}, d_{G_k}(v, r))$). We will need this bound in Section 2.4. The second bound follows easily from the first one and we will need it in Section 2.3.

Algorithm 2.1: ES-tree (formulated for weighted undirected graphs)

```
// Internal data structures:
// N(u): for every node u a heap N(u) whose intended use is to
   store for every neighbor v of u in the current graph the
   value of ℓ(v) + w(u, v) where w(u, v) is the weight of the edge
   (u, v) in the current graph
// Q: global heap whose intended use is to store nodes whose
   levels might need to be updated
```

Procedure INITIALIZE()
  Compute shortest paths tree from r in (G_0, w_0) up to depth R^{d}
  foreach *node* u **do**
    Set $\ell(u) = d_{G_0}(u, r)$
    for *every edge* (u, v) **do** insert v into heap $N(u)$ of u with key $\ell(v) + w(u, v)$

Procedure DELETE(u, v)
  INCREASE(u, v, ∞)

Procedure INCREASE(u, v, $w(u, v)$)
  // Increase weight of edge (u, v) to $w(u, v)$
  Insert u and v into heap Q with keys $\ell(u)$ and $\ell(v)$ respectively
  Update key of v in heap $N(u)$ to $\ell(v) + w(u, v)$ and key of u in heap $N(v)$ to $\ell(u) + w(u, v)$
  UPDATELEVELS()

Procedure UPDATELEVELS()
  while *heap* Q *is not empty* **do**
    Take node y with minimum key $\ell(y)$ from heap Q and remove it from Q
    $\ell'(y) \leftarrow \min_z(\ell(z) + w(y, z))$
    // $\ell'(y)$ can be retrieved from the heap $N(y)$.
       $\arg\min_z(\ell(z) + w(y, z))$ is y's parent in the ES-tree
    if $\ell'(y) > \ell(y)$ **then**
      $\ell(y) \leftarrow \ell'(y)$
      if $\ell'(y) > R^d$ **then** $\ell(y) \leftarrow \infty$
      foreach *neighbor* x *of* y **do**
        update key of y in heap $N(x)$ to $\ell(y) + w(x, y)$
        insert x into heap Q with key $\ell(x)$ if Q does not already contain x

Implementation The pseudocode for achieving the above result can be found in Algorithm 2.1. (For simplicity we show an implementation using heaps, which causes an extra $\log n$ factor in the running time. King [79] explains how to avoid heaps in order to improve the running time by a factor of $\log n$.) For every node x the ES-tree maintains a heap $N(x)$ that stores for every neighbor y of x in the current graph the value of $\ell(y) + w(x, y)$ where $w(x, y)$ is the weight of the edge (x, y) in the current graph. (Intuitively, $N(x)$ corresponds to the "knowledge" of x about its neighbors.) These data structures can be initialized in $\tilde{O}(m)$ time by running, for example, Dijkstra's algorithm (see procedure Initialize).[14]

Edge deletions and weight increases are handled in procedure Delete and Increase respectively; in fact, deletion is a special case of weight increase where we set the edge weight to ∞. Every weight increase of an edge (u, v) might cause the levels of some nodes to increase. The algorithm uses a heap Q to keep track of such nodes. Initially (at the time $w(u, v)$ is increased) the algorithm inserts u and v to Q as the levels of u and v might increase (see Line 9). It also updates $N(u)$ and $N(v)$ as in Line 10. Then it updates the levels on nodes in Q using procedure UpdateLevels.

Procedure UpdateLevels processes the nodes in Q in the order of their current level (see the while-loop starting on Line 13). In every iteration it will process the node y in Q with smallest $\ell(y)$ (as in Line 14). The lowest level that is possible for a node y is $\ell'(y) = \min_z(\ell(z) + w(y, z))$, the minimum of $\ell(z) + w(y, z)$ over all neighbors z of y in the current graph (following Equation (2.1)). Therefore every node y will repeatedly update its level to $\ell'(y)$ (unless its level already has this value); see Line 15. (An exception is when the level of a node x exceeds the desired depth R^{d}. In this case the level of x is set to ∞ and x will never be connected to the tree again. See Line 18.) If this updating rule leads to a level increase, the algorithm has to update the heap $N(x)$ of every neighbor x and put x to the heap Q (since the level of x might increase) as in the for-loop starting on Line 19 (this is equivalent to having y send a message to x in the informal description).

The running time analysis takes into account the level increases occurring in the ES-tree. It is based on the following observation: For every node x processed in the while-loop of the procedure UpdateLevels in Algorithm 2.1, if the level of x increases, the algorithm has to spend time $O(\deg(x) \log n)$ for updating the heaps $N(y)$ of all neighbors y of x and adding these neighbors to heap Q. If the level of x does not increase, the algorithm only has to spend time $O(\log n)$. In the second case the running time can be charged to one of the following events that causes x to be in Q: (1) a weight increase of some edge (x, y) and (2) a level increase of some neighbor of x. This leads to the result in Lemma 2.2.10.

[14] Alternatively we could compute the initial shortest paths tree using the Even-Shiloach algorithm itself: Let G_0' be the modification of G_0 where we add an edge (r, v) of weight 1 for every node v. We obtain G_0 from G_0' by deleting each such edge. Starting from a trivial shortest paths tree in G_0' in which the parent of every node $v \neq r$ is r, we obtain the shortest paths tree of G_0 in time $O(\sum_{v \in V} \deg_{G_0}(v) \cdot \min(R^{\mathrm{d}}, d_{G_0}(v, r))) = O(mR^{\mathrm{d}})$.

2.2.3 The Framework of Roditty and Zwick

In the following we review the algorithm of Roditty and Zwick [113] because its main ideas are the basis of our own algorithms. We will put their arguments in a certain structure that clarifies for which part of the algorithm we obtain improvements. Their algorithm is based on the following observation. Consider approximating the distance $d_G(x, y)$ for some pair of nodes x and y. For some $0 < \epsilon \leq 1$, we want a $(1 + O(\epsilon), 0)$-approximate value of $d_G(x, y)$. Assume that we know an integer p such that 2^p is a "distance guess" of $d_G(x, y)$, i.e.,

$$2^p \leq d_G(x, y) \leq 2^{p+1}. \tag{2.2}$$

Now, suppose that there is a node z that is close to x, i.e.,

$$d_G(x, z) \leq \epsilon 2^p. \tag{2.3}$$

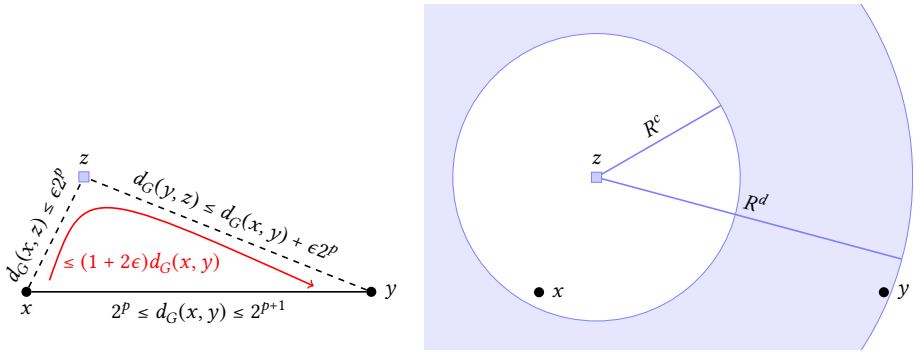
Then, it follows that we can use $d_G(x, z) + d_G(z, y)$ as a $(1 + 2\epsilon)$-approximation of the true distance $d_G(x, y)$; this follows from applying the triangle inequality twice (also see Figure 2.2a):

$$\begin{aligned} d_G(x, y) \leq d_G(x, z) + d_G(z, y) &\leq d_G(x, z) + (d_G(z, x) + d_G(x, y)) \\ &\leq (1 + 2\epsilon) d_G(x, y)\,. \end{aligned} \tag{2.4}$$

Thus, under the assumption that for any x we only want to determine the distances from x to nodes y with $d_G(x, y)$ in the range from 2^p to 2^{p+1}, we only have to make sure that there is a node z that satisfies Equation (2.3); we call such node z a *center*. We will maintain a set U of nodes such that for every node x there is a node $z \in U$ that satisfies Equation (2.3). In fact, we only need this to be true for nodes x that are in a "big" connected component since if the connected component containing x is too small then there is no node y that satisfies Equation (2.2). We call such U a *center cover*. More precisely:

Definition 2.2.12 (Center cover)**.** *Let U be a set of nodes of a graph G, and let R^c be a positive integer denoting the* cover range*. We say that a node x is* covered *by a node $c \in U$ in G if $d_G(x, c) \leq R^c$. We say that U is a* center cover of G with parameter R^c *if every node x that is in a connected component of size at least R^c is covered by some node $c \in U$ in G.*

One main component of Roditty-Zwick's framework as we describe it is the *center cover data structure*. This data structure maintains a center cover U as above. Furthermore, for every center $z \in U$, we will maintain the distance to every node y such that $d_G(z, y) \leq 2^{p+2}$. This will allow us to compute $d_G(x, z) + d_G(z, y)$ as an approximate value of $d_G(x, y)$ (as in Equation (2.4)). In general, we treat the number 2^{p+2} as another parameter of the data structure denoted by R^d (called distance range parameter). The values of R^c and R^d are typically closely related; in particular, $R^c \leq R^d = O(R^c)$. The center cover data structure is formally as follows (also see Figure 2.2b).



(a) (b)

Figure 2.2: (a) depicts Equations (2.2) to (2.4). (b) shows the cover range (small circle) and distance range (big circle) used by the center cover data structure (Definition 2.2.13).

Definition 2.2.13 (Center cover data structure)**.** *A* center cover data structure *with* cover range parameter R^c *and* distance range parameter R^d *for a decremental graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *maintains, for every* $0 \le i \le k$*, a set of* centers $C_i = \{1, 2, \dots, l\}$ *and a set of nodes* $U_i = \{c_i^1, c_i^2, \dots, c_i^l\}$ *such that,* U_i *is a center cover of* G_i *with parameter* R^c*. For every center* $j \in C_i$ *and every* $0 \le i \le k$*, we call* $c_i^j \in U_i$ *the* location *of center* j *in* G_i *and for every node* x *we say that* x *is covered by* j *if* x *is covered by* c_i^j *in* G_i*. After the* i*-th edge deletion (where* $0 \le i \le k$*), the data structure provides the following operations:*

- Delete(u, v)*: Delete the edge* (u, v) *from* G_i*.*
- Distance(j, x)*: Return the distance* $d_{G_i}(c_i^j, x)$ *between the location* c_i^j *of center* j *and the node* x*, provided that* $d_{G_i}(c_i^j, x) \le R^d$*. If* $d_{G_i}(c_i^j, x) > R^d$*, then return* ∞*.*
- FindCenter(x)*: If* x *is in a connected component of size at least* R^c *in* G_i*, return a center* j *(with location* c_i^j*) such that* $d_{G_i}(x, c_i^j) \le R^c$*. If* x *is in a connected component of size less than* R^c *in* G_i*, then either return* $\perp$ *or return a center* j *(with location* c_i^j*) such that* $d_{G_i}(x, c_i^j) \le R^c$*.*

The total update time *is the total time needed for performing all* k *delete operations and the initialization. The* query time *is the worst-case time needed to answer a single* Distance *or* FindCenter *query.*

As the update time of the data structure will depend on the number l of centers, the goal is to keep l as small as possible, preferably $l = \tilde{O}(n/R^c)$. As an example, consider the following *randomized* implementation of Roditty and Zwick [113]: randomly pick a set U of $((n/R^c) \operatorname{polylog} n)$ nodes as the set of centers. With high probability, this set will remain a center cover during all deletions (see Lemma 1.3.2). The Distance and FindCenter queries can be answered in $O(1)$ time by maintaining an ES-tree of depth mR^d for every center. The total time to maintain this data structure

is thus $\tilde{O}(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. We typically set $R^{\mathrm{c}} = \Omega(R^{\mathrm{d}})$. In this case, the total time becomes $\tilde{O}(mn)$.

Note that while the implementation of Roditty and Zwick always uses the same set of centers U, the center cover data structure that we define is flexible enough to allow this set to *change over time*, i.e., it is possible that $U_i \neq U_{i+1}$ for some i. In fact, our definition separates between the notion of *centers* (set C_i) and *locations* (set U_i) as it will allow one center to change its location over time. This is necessary when we want to maintain $o(n)$ centers deterministically since if we fix the centers and their locations, then an adversary can delete all edges adjacent to the centers, making all non-center nodes uncovered. (The randomized algorithm of Roditty and Zwick can avoid this by using randomness and assuming that the adversary is oblivious.)

Using Center Cover Data Structure to Solve APSP Given a center cover data structure, an approximate decremental APSP data structure is obtained as follows. We maintain $\lceil \log n \rceil$ "instances" of the center cover data structure where the p-th instance has parameters $R^{\mathrm{c}} = \epsilon 2^p$ and $R^{\mathrm{d}} = 2^{p+2}$ and is responsible for the distance range from 2^p to 2^{p+1} (for all $0 \leq p \leq \lfloor \log n \rfloor$). Suppose that after the i-th deletion we want to answer a query for the approximate distance between the nodes x and y. For every p, we first query for a center covering x from the p-th instance of the center cover data structure. Denote the *location* of this center by z_p. The distance estimate provided by the p-th instance is $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$. We will output $\min_p d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ as an estimate of $d_{G_i}(x, y)$. (Note that is possible that $z_p = \perp$, i.e., there is no center covering x in the p-th instance. This might happen if x is in a connected component of size less than R^{c}. In this case we set $d_{G_i}(z_p, x) + d_{G_i}(z_p, y) = \infty$.)

To see the approximation guarantee, let p^* be such that $2^{p^*} \leq d_{G_i}(x, y) \leq 2^{p^*+1}$. Observe that if $p = p^*$, then $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is a $(1+O(\epsilon), 0)$-approximate distance estimate (due to Equation (2.4)), and if $p \neq p^*$ then $d_{G_i}(z_p, x) + d_{G_i}(z_p, y) \geq d_{G_i}(x, y)$ (by the triangle inequality). Thus, $\min_p d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is a $(1+O(\epsilon), 0)$-approximate value of $d_{G_i}(x, y)$. The query time, which is the time to compute $\min_p d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$, is $O(\log n)$.

The query time can be reduced to $O(\log \log n)$ as follows. Observe that for any $p < p^*$, the distance $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ might be ∞ if $d_{G_i}(z_p, y) > R^{\mathrm{d}}$; however, if it is finite, it will provide a $(1+\epsilon, 0)$-approximation (since $d_{G_i}(z_p, x) \leq \epsilon p$). In other words, it suffices to find the smallest index p^{**} for which $d_{G_i}(z_{p^{**}}, x) + d_{G_i}(z_{p^{**}}, y)$ is finite; this value will be a $(1 + O(\epsilon), 0)$-approximate value of $d_{G_i}(x, y)$. To find this index, observe further that for any $p > p^*$, either $z_p = \perp$ or $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is finite. So, we can find p^{**} by a binary search (since for any p, if $z_p = \perp$ or $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is finite, then we know that $p^{**} \leq p$).

Theorem 2.2.14 ([113]). *Assume that for all parameters R^c and R^d such that $R^c \leq R^d$ there is a center cover data structure that has constant query time and a total update time of $T(R^c, R^d)$. Then, for every $\epsilon \leq 1$, there is a $(1+\epsilon, 0)$-approximate decremental APSP*

data structure with $O(\log\log n)$ query time and a total update time of $\sum_p T(R^c_p, R^d_p)$ where $R^c_p = \epsilon 2^p$ and $R^d_p = 2^{p+2}$ (for $0 \le p \le \lfloor \log n \rfloor$).

As shown before, Roditty and Zwick [113] obtain a randomized center cover data structure with constant query time and a total update time of $\tilde{O}(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. By Theorem 2.2.14 they get a $(1+\epsilon, 0)$-approximate decremental APSP data structure with a total update time of $\tilde{O}(mn/\epsilon)$ and a query time of $O(\log\log n)$. Note that the query time can sometimes be reduced further to $O(1)$, and this is the case for their algorithm as well as our randomized algorithm in Section 2.3. This is essentially because there is a $(3,0)$-approximation randomized algorithm for APSP, which can be used to approximate p^* (we defer details to Lemma 2.3.19). To analyze the total update time of their data structure for k deletions observe that

$$\sum_{p=0}^{\lfloor \log n \rfloor} \tilde{O}(mnR^{\mathrm{d}}_p/R^{\mathrm{c}}_p) = \sum_{p=0}^{\lfloor \log n \rfloor} \tilde{O}(mn2^{p+1}/(2^p\epsilon)) = \sum_{p=0}^{\lfloor \log n \rfloor} \tilde{O}(mn/\epsilon)$$
$$= \tilde{O}(mn\log n/\epsilon) = \tilde{O}(mn/\epsilon)\,.$$

In Section 2.3 we show that we can maintain an *approximate* version of the center cover data structure in time $\tilde{O}(n^{5/2}R^{\mathrm{d}}/(\epsilon R^{\mathrm{c}}))$. Using this data structure, we will get a $(1+\epsilon, 2)$-approximate decremental APSP data structure with a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time. In Section 2.4 we show how to maintain an exact *deterministic* center cover data structure with a total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. By Theorem 2.2.14 this immediately implies a deterministic $(1+\epsilon, 0)$-approximate decremental APSP data structure with a total update time of $O(mn\log n)$ and a query time of $O(\log\log n)$.

2.3 $\tilde{O}(n^{5/2})$-Total Time $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-Approximation Algorithms

In this section, we present a data structure for maintaining all-pairs shortest paths under edge deletions with multiplicative error $1+\epsilon$ and additive error 2 that has a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$. The data structure is correct with high probability. We also show a variant of this data structure with multiplicative error $2+\epsilon$ and no additive error. In doing this, we introduce the notion of a *persevering path* (see Definition 2.3.1) and a *locally persevering emulator* (Definition 2.3.2). In Section 2.3.1, we then present the locally persevering emulator that we will use to obtain our result. Then, in Section 2.3.2 we explain our main technique, called *monotone Even-Shiloach tree*, where we maintain the distances from a single node to all other nodes, up to some distance R^{d}, in a locally persevering emulator. (Recall that R^{d} is a parameter called "distance range".) In Section 2.3.3 we show how approximate decremental SSSP helps in solving approximate decremental APSP. Finally, in Section 2.3.4, we show how to put the results in Sections 2.3.1 to 2.3.3 together to obtain the desired $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-approximate decremental APSP data structures.

Definition 2.3.1 (Persevering path). *Let $\mathscr{H} = (H_i, w_i)_{0\le i\le k}$ be a dynamic weighted graph. We say that a path $\pi = \langle v_0, v_1, \dots, v_\ell \rangle$ is* persevering up to time t *(where $t \le k$) if for all $0 \le i \le \ell - 1$,*

$$\forall 0 \le j \le t \ : \ (v_i, v_{i+1}) \in E(H_j) \quad \text{and} \quad \forall 0 \le j < t \ : \ w_j(v_i, v_{i+1}) \le w_{j+1}(v_i, v_{i+1}).$$

In other words, edges in π always exist in $\mathscr{H}$ up to time t and their weights never decrease.

We now introduce the notion of a *locally persevering emulator.* An (α, β)-emulator of a dynamic graph $\mathscr{G} = (G_i)_{0\le i\le k}$ is usually another dynamic weighted graph $\mathscr{H} = (H_i, w_i)_{0\le i\le k}$ with the same set of nodes as $\mathscr{G}$ that preserves the distance of the original dynamic graph, i.e., for all $i \le k$ and all nodes x and y, there is a path π_{xy} in H_i such that $d_{G_i}(x, y) \le w_i(\pi_{xy}) \le \alpha d_{G_i}(x, y) + \beta$. The notion of a *locally persevering* emulator has another parameter τ. It requires the condition $d_{G_i}(x, y) \le w_i(\pi_{xy}) \le \alpha d_{G_i}(x, y) + \beta$ to hold only when $d_{G_i}(x, y) \le \tau$. More importantly, it puts an additional restriction that the path π_{xy} must be either a shortest path in G_i or a persevering path.

Definition 2.3.2 (Locally persevering emulator). *Consider parameters $\alpha \ge 1$, $\beta \ge 0$ and $\tau \ge 1$, a dynamic graph $\mathscr{G} = (G_i)_{0\le i\le k}$, and a dynamic weighted graph $\mathscr{H} = (H_i, w_i)_{0\le i\le k}$. For every $i \le k$, we say that a path π in G_i is* contained *in (H_i, w_i) if every edge of π is contained in H_i and has weight 1. We say that $\mathscr{H}$ is an* (α, β, τ)-locally persevering emulator *of $\mathscr{G}$ if for for all nodes x and y we have*

(1) $d_{G_i}(x, y) \le d_{H_i, w_i}(x, y)$ for all $0 \le i \le k$, and

(2) there are t_1 and t_2 with $0 \le t_1 < t_2 \le k + 1$ such that the following holds:

- *a) There is a path π from x to y in $\mathscr{H}$ that is persevering (at least) up to time t_1 and satisfies $w_t(\pi) \le \alpha d_{G_t}(x, y) + \beta$.*
- *b) For every $t_1 < i \le t_2$, a shortest path from x to y in G_i is contained in (H_i, w_i).*
- *c) For every $i \ge t_2$, $d_{G_i}(x, y) > \tau$.*

Condition (1) simply says that $\mathscr{H}$ does not underestimate the distances in $\mathscr{G}$. Condition (2) says that the distance between x and y must be preserved in $\mathscr{H}$ in the following specific way: In the beginning (see (2)a), it must be approximately preserved by a single path π (thus π is a persevering path). Whenever π disappears, the shortest path between x and y must appear in $\mathscr{H}$ (see (2)b). However, we can remove all these conditions whenever the distance between x and y is more than τ (see (2)c).

2.3.1 $(1, 2, \lceil 2/\epsilon \rceil)$-Locally Persevering Emulator of Size $\tilde{O}(n^{3/2})$

In the following we present the locally persevering emulator that we will use to achieve a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$ for decremental approximate APSP. Roughly

speaking, we can replace the running time of $\tilde{O}(mn/\epsilon)$ by $\tilde{O}(n^{5/2}/\epsilon^2)$ because this emulator always has $\tilde{O}(n^{3/2})$ edges. However, to be technically correct, we have to use the stronger fact that the number of edges ever contained in the emulator is $\tilde{O}(n^{3/2})$, as in the following statement.

Lemma 2.3.3 (Existence of $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator with $\tilde{O}(n^{3/2})$ edges)**.** *For every $0 < \epsilon \leq 1$ and every decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$, there is data structure that maintains a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ in $O(mn^{1/2} \log n/\epsilon)$ total time such that $\mathcal{H}$ is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator with high probability. Moreover, the number of edges ever contained in the emulator is $|E_k(\mathcal{H})| = O(n^{3/2} \log n)$ and the total number of updates in $\mathcal{H}$ is $\phi_k(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.*

We construct a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ as follows. We pick a set D of nodes by including every node independently with probability $(a \ln n)/\sqrt{n}$ for a large enough constant a. Note that the size of D is $O(\sqrt{n} \log n)$ in expectation. It is well-known that by this type of sampling every node with degree more than $\sqrt{n}$ has a neighbor in D with high probability (see, e.g., [41, 127]), i.e., D *dominates* all high-degree nodes. This is even true for every version G_i of a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$. For every $0 \leq i \leq k$, we define that the graph H_i contains the following two types of edges. For every node $x \in D$ and every node y such that $d_{G_i}(x, y) \leq \lceil 2/\epsilon \rceil + 1$, H_i contains an edge (x, y) of weight $d_{G_i}(x, y)$. For every node x such that $\deg_{G_i}(x) \leq \sqrt{n}$, H_i contains every edge (x, y) of G_i.[15] Note that as edges are deleted from $\mathcal{G}$ distances between nodes might increase which in turn increases the weights of the corresponding edges in $\mathcal{H}$. When the distance between x and y in $\mathcal{G}$ exceeds $\lceil 2/\epsilon \rceil + 1$, the edge (x, y) is deleted from $\mathcal{H}$.

In the following we prove Lemma 2.3.3 by arguing that the dynamic graph $\mathcal{H}$ described above has the following four desired properties:

- $\mathcal{H}$ is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathcal{G}$.
- The expected number of edges ever contained in the emulator is $|E_k(\mathcal{H})| = O(n^{3/2} \log n)$.
- The expected total number of updates in $\mathcal{H}$ is $\phi_k(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.
- The edges of $\mathcal{H}$ can be maintained in expected total time $O(mn^{1/2} \log n/\epsilon)$ for k deletions in $\mathcal{G}$.

The last item refers to the time needed to determine, after every deletion in $\mathcal{G}$ which edges are contained in $\mathcal{H}$ and what their weight is.

[15] Our construction is very similar to the classic $(1, 2)$-emulator given by Dor, Halperin, and Zwick [41]. The main difference is that we can only insert edges of limited weight into the emulator; in particular, we only have edges of weight $O(1/\epsilon)$ in the emulator. One reason for this choice is that it is not known whether the $(1, 2)$-emulator of Dor et al. can be maintained in $\tilde{O}(mn)$ time under edge deletions.

Lemma 2.3.4 (Locally persevering)**.** *The dynamic graph $\mathscr{H}$ described above is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathscr{G}$ with high probability.*

Proof. Let $t \leq k$ and let x and y be a pair of nodes. We first argue that $d_{G_t}(x, y) \leq d_{H_t, w_t}(x, y)$. It is clear from the construction of (H_t, w_t) that every edge in (H_t, w_t) corresponds to an edge in G_t or to a path in G_t. Therefore no path in (H_t, w_t) from x to y can be shorter than the distance $d_{G_t}(x, y)$ of x to y in G_t.

We now argue that $\mathscr{H}$ fulfills part two of the definition of a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathscr{G}$. Assume that $d_{G_t}(x, y) \leq \lceil 2/\epsilon \rceil$ and that no shortest path from x to y in G_t is also contained in (H_t, w_t). Let π be an arbitrary shortest path from x to y in G_t. Since π is not contained in H_t, there must be some edge (u, v) on π such that $(u, v) \notin E(H_t)$. This can only happen if u has degree more than $\sqrt{n}$ in G_t. With high probability u has a neighbor $z \in D$ in G_t (see, e.g., [41, 127]). Now consider any $i \leq t$. Note that $d_{G_t}(x, u) \leq d_{G_t}(x, y) \leq \lceil 2/\epsilon \rceil$ and that $d_{G_i}(x, z) \leq d_{G_t}(x, z)$ because distances never decrease in a decremental graph. By the triangle inequality we get

$$d_{G_i}(x, z) \leq d_{G_t}(x, z) \leq d_{G_t}(x, u) + d_{G_t}(u, z) \leq \lceil 2/\epsilon \rceil + 1 \,.$$

Therefore, for every $i \leq t$, H_i contains an edge (x, z) of weight $w_i(x, z) = d_{G_i}(x, z)$, which means that the edge (x, z) is persevering up to time t. The same argument shows that H_i also contains an edge (z, y) of weight $w_i(z, y) = d_{G_i}(z, y)$ for every $i \leq t$, i.e., (z, y) is also persevering up to time t. Now consider the path π' in H_t consisting of the edges (x, z) and (z, y). Since both edges are persevering up to time t, also the path π' is persevering up to time t. Furthermore, π' guarantees the desired approximation:

$$\begin{aligned} w_t(\pi') = w_t(x, z) + w_t(z, y) &= d_{G_t}(x, z) + d_{G_t}(z, y) \\ &\leq d_{G_t}(x, u) + d_{G_t}(u, z) + d_{G_t}(z, u) + d_{G_t}(u, y) \\ &= d_{G_t}(x, u) + d_{G_t}(u, y) + 2 \\ &= d_{G_t}(x, y) + 2 \,. \end{aligned}$$

To explain the last equation, remember that u lies on a shortest path from x to y and therefore $d_{G_t}(x, y) = d_{G_t}(x, u) + d_{G_t}(u, y)$. Thus, $\mathscr{H}$ is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathscr{G}$. □

Lemma 2.3.5 (Number of edges)**.** *The number of edges ever contained in the dynamic graph $\mathscr{H}$ is $|E_k(\mathscr{H})| = O(n^{3/2} \log n)$ in expectation.*

Proof. Every edge in H_i either was inserted at some time or it is an edge that is also contained in H_0. Thus, it is sufficient to bound number of inserted edges and the number of edges in H_0 by $O(n^{3/2} \log n)$.

We first show that the number of edges in H_0 is $|E(H_0)| = O(n^{3/2} \log n)$. We can charge each edge in H_0 either to a node in D or to a node with degree at most $\sqrt{n}$. For every node $x \in D$ there might be $O(n)$ edges adjacent to x in H_0. Since there

are $O(\sqrt{n}\log n)$ many nodes in D, the number of edges charged to these nodes is $O(n^{3/2}\log n)$. For nodes with degree at most $\sqrt{n}$ there are $O(\sqrt{n})$ edges adjacent to x in H_0. Since there are $O(n)$ such nodes, the number of edges charged to these nodes is $O(n^{3/2})$. In total, we get

$$|E(H_0)| = O(n^{3/2}\log n) + O(n^{3/2}) = O(n^{3/2}\log n)\,.$$

We now show that the number of edges inserted into $\mathscr{H}$ over all deletions in $\mathscr{G}$ is $O(n^{3/2})$. Every time the degree of a node x changes from $\deg_{G_i}(x) > \sqrt{n}$ to $\deg_{G_{i+1}}(x) = \sqrt{n}$ (for some $0 \le i < k$) we insert all $\sqrt{n}$ edges adjacent to x in G_{i+1} into H_{i+1}. In a decremental graph it can happen at most once for every node that the degree of a node drops to $\sqrt{n}$. Therefore at most $n^{3/2}$ edges are inserted in total. □

Lemma 2.3.6 (Number of updates)**.** *The total number of updates in the dynamic graph $\mathscr{H}$ described above is $\phi_k(\mathscr{H}) = O(n^{3/2}\log n/\epsilon)$ in expectation.*

Proof. Only the following kinds of updates appear in $\mathscr{H}$: edge insertions, edge deletions, and edge weight increases. Every edge that is inserted or deleted has to be contained in $\mathscr{H}$ at some time. Thus, we can bound the number of insertions and deletions by $|E_k(\mathscr{H})|$, which is $O(n^{3/2}\log n)$ by Lemma 2.3.5

It remains to bound the number of edge weight increases by $O(n^{3/2}\log n/\epsilon)$. All weighted edges are incident to at least one node in D. The maximum weight of these edges is $\lceil 2/\epsilon\rceil + 1$ and the minimum weight is 1. As all edge weights are integer, the weight of such an edge can increase at most $\lceil 2/\epsilon\rceil + 1$ times. As there are $O(\sqrt{n}\log n)$ nodes in D, each having $O(n)$ weighted edges, the total number of edge weight increases is $O(n^{3/2}\log n/\epsilon)$. □

Lemma 2.3.7 (Running time)**.** *The edges of the dynamic graph $\mathscr{H}$ described above can be maintained in expected total time $O(mn^{1/2}\log n/\epsilon)$ over all k edge deletions in $\mathscr{G}$.*

Proof. We use the following data structures: (A) For every node, we maintain its incident edges in $\mathscr{H}$ with a dynamic dictionary using dynamic perfect hashing [40] or cuckoo hashing [103]. This graph representation allows us to perform insertions and deletions of edges as well as edge weight increases. (B) For every node x, we maintain the degree of x in $\mathscr{G}$. (C) For every node $x \in D$ we maintain a (classic) ES-tree (see Section 2.2.2) rooted at x up to distance $\lceil 2/\epsilon\rceil + 1$.

We now explain how to process the i-the edge deletion in $\mathscr{G}$ of, say, the edge (u, v). First of all we update (B) by decreasing the number that stores the degree of u in $\mathscr{G}$ and then do the same for v. If the degree of u (or v) drops to $\sqrt{n}$ we insert all edges incident to u (or v) in G_i into H_i in (A). After this procedure, for every node $x \in D$, we do the following to update (C): First of all, we report the deletion of (u, v) to the ES-tree rooted at x. Every node y has a level in this ES-tree. If the level of y increases to ∞, then $d_{G_i}(x, y) > \lceil 2/\epsilon\rceil + 1$ and therefore we remove the edge (x, y) from $\mathscr{H}$ in (A). If the level of y increases, but does not reach ∞, then $d_{G_i}(x, y) \le \lceil 2/\epsilon\rceil + 1$ and we update the weight of the edge (x, y) in (A).

We can perform each deletion, insertion and edge weight increase expected amortized constant time. As there are $O(n^{3/2} \log n/\epsilon)$ updates in $\mathscr{H}$ in expectation by Lemma 2.3.6, the expected total time for maintaining (A) is $O(n^{3/2} \log n/\epsilon)$. We need constant time per deletion in $\mathscr{G}$ to update (B) and thus time $O(m)$ in total. Maintaining the ES-tree takes total time $O(m/\epsilon)$ for each node in D (see Section 2.2.2). Since in expectation there are $O(n^{1/2} \log n)$ nodes in D, the expected total time for maintaining (A) is $O(mn^{1/2} \log n/\epsilon)$ in total.

Thus, the expected total update time for maintaining $\mathscr{H}$ under deletions in $\mathscr{G}$ is $O(n^{3/2} \log n/\epsilon + m + mn^{1/2} \log n/\epsilon)$, which is $O(mn^{1/2} \log n/\epsilon)$. □

2.3.2 Maintaining Distances Using Monotone Even-Shiloach Tree

In this section, we show how to use a locally persevering emulator to maintain the distances from a specific node r (called *root*) to all other nodes, up to distance R^{d}, for some parameter R^{d}. The hope of using an emulator is that the total update time will be smaller since an emulator has a smaller number of edges. In particular, recall that if we run an ES-tree on an input graph, the total update time is $\tilde{O}(mR^{\mathrm{d}})$. Now consider running an ES-tree on an emulator $\mathscr{H}$ instead, we might hope to get a running time of $\tilde{O}(m'R^{\mathrm{d}})$, where m' is the number of edges ever appearing in $\mathscr{H}$. This is beneficial when $m' \ll m$ (for example, the emulator we construct in the previous section has $m' = \tilde{O}(n^{1.5})$ which is less than m when the input graph is dense). The main result of this section is that we can achieve exactly this when $\mathscr{H}$ is a locally-persevering emulator and we run a variant of ES-tree called *monotone ES-tree* on $\mathscr{H}$.

Lemma 2.3.8 (Monotone ES-tree + Locally Persevering Emulator)**.** *For every distance range parameter R^d, every source node r, and every decremental graph $\mathscr{G} = (G_i)_{0 \le i \le k}$ with an (α, β, τ)-locally persevering emulator $\mathscr{H} = (H_i, w_i)_{0 \le i \le k}$, the monotone ES-tree on $\mathscr{H}$ is an $(\alpha + \beta/\tau, \beta)$-approximate decremental SSSP data structure for $\mathscr{G}$. It has constant query time and a total update time of*

$$O(\phi_k(\mathscr{H}) + |E_k(\mathscr{H})| \cdot ((\alpha + \beta/\tau)R^d + \beta)),$$

where $\phi_k(\mathscr{H})$ is the total number of updates in $\mathscr{H}$ up to time k and $E_k(\mathscr{H})$ is the set of all edges ever contained in $\mathscr{H}$ up to time k.

Note that we need to modify the ES-tree because although the input graph undergoes only edge deletions, the emulator might have to undergo some edge *insertions*. If we straightforwardly extend the ES-tree to handle insertions, we will have to keep the level of any node y at $\ell(y) = \min_z(\ell(z) + w(y, z))$ as in Line 15 of Algorithm 2.1. This might cause the level $\ell(y)$ of some node y in the ES-tree to *decrease*. This *destroys the monotonicity* of levels of nodes, which is the key to guarantee the running time of the ES-tree, as shown in Section 2.2.2. The monotone ES-tree is a variant that insists on keeping the nodes' levels monotone (thus the name); it never decreases the level of any node.

Implementation of Monotone ES-Tree Our monotone ES-tree data structure is a modification of the ES-tree, which always maintains the level $\ell(x)$ of every node x in a shortest paths tree rooted at r up to depth R^{d}, as presented in Section 2.2.2. Algorithm 2.2 shows the pseudocode of the monotone ES-tree. Our modification can deal with edge insertions, but does this in a *monotone* manner: it will never decrease the level of any node. In doing so, it will lose the property of providing a shortest paths tree of the underlying dynamic graph, which in our case is the emulator $\mathscr{H}$. However, due to special properties of the emulator, we can still guarantee that the level provided by the monotone ES-tree is an approximation of the distance in the original decremental graph $\mathscr{G}$. The distance estimate provided for a node x is the level of x in the monotone ES-tree.

The overall algorithm now is as follows (see Algorithm 2.2 for details). We initialize the monotone ES-tree by computing a shortest paths tree in the emulator H_0 up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. For every node x in this tree we set $\ell(x) = d_{H_0}(x, r)$ and for every other node x we set $\ell(x) = \infty$. Starting with these levels, we maintain an ES-tree rooted at r up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ on the graph $\mathscr{H}$. This ES-tree alone cannot deal with edge insertions and edge weight increases. Our additional procedure that is called after the insertion of an edge (u, v) only updates the value of v in the heap $N(u)$ of u to $\ell(v) + w(u, v)$. In particular, the level of u is not changed after such an insertion.

Order of Updates Before we start analyzing our algorithm we clarify a crucial detail about the order of updates in the locally persevering emulator $\mathscr{H}$. Consider an edge deletion in the graph G_i that results in the graph G_{i+1}. In the emulator $\mathscr{H}$, it might be the case that several updates are necessary to obtain (H_{i+1}, w_{i+1}) from (H_i, w_i). There could be several insertions, edge weight increases and edge deletions at once.[16] Our algorithm will process these updates (using the monotone ES-tree) in a specific order: First, it processes the edge insertions, one after the other. Afterwards, we process the edge deletions and edge weight increases (also one after the other). This order is crucial for the correctness of our algorithm.

Analysis We first argue about the correctness of the monotone ES-tree and afterwards argue about its running time. In the following we let $\ell_i(u)$ be the level of u in the monotone ES-tree after it has processed the i-th edge deletion in $\mathscr{G}$ (which could mean that it has processed a whole series of insertions, weight increases and deletions of the emulator $\mathscr{H}$). Remember that (H_i, w_i) denotes the emulator after all updates caused by the i-th deletion in $\mathscr{G}$. We say that an edge (u, v) is *stretched* if $\ell_i(u) \neq \infty$ and $\ell_i(u) > \ell_i(v) + w_i(u, v)$. We say that a node u is *stretched* if it is incident to an edge (u, v) that is stretched. Note that for a node u that is not stretched we either have $\ell_i(u) = \infty$ or $\ell_i(u) \leq \ell_i(v) + w_i(u, v)$ for every edge $(u, v) \in E(H_i)$. Our analysis uses four simple observations about the algorithm. (Recall that a tree edge

[16] We could also allow edge weight decreases and handle them in exactly the same way as edge insertions. For simplicity, we omit this case from our description.

Algorithm 2.2: Monotone ES-tree

```
// The algorithm is like the usual ES-tree (Algorithm 2.1)
   with three modifications:
// 1. The algorithm runs on ℋ = (H_0, H_1, ...) instead of 𝒢.
// 2. The depth of the tree is (α + β/τ)R^d + β instead of R^d
// 3. There are additional procedures for the insertion of
   edges and edge weight increases.
// Procedures Delete and Increase are the same as before.
// Line numbers in the form i* indicate lines that are
   different from Algorithm 2.1. Blue color marks the changes.
```

1 **Procedure** Initialize()
2* Compute shortest paths tree from r in (H_0, w_0) up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$
3 **foreach** *node* u **do**
4* Set $\ell(u) = d_{H_0}(u, r)$
5 **for** *every edge* (u, v) **do** insert v into heap $N(u)$ of u with key $\ell(v) + w(u, v)$

6 **Procedure** Insert(u, v, $w(u, v)$)
7 Insert v into heap $N(u)$ with key $\ell(v) + w(u, v)$ and u into heap $N(v)$ with key $\ell(u) + w(u, v)$

8 **Procedure** UpdateLevels()
9 **while** *heap* Q *is not empty* **do**
10 Take node y with minimum key $\ell(y)$ from heap Q and remove it from Q
11 $\ell'(y) \leftarrow \min_z(\ell(z) + w(y, z))$
// $\min_z(\ell(z) + w(y, z))$ can be retrieved from the heap $N(y)$. $\arg\min_z(\ell(z) + w(y, z))$ is y's parent in the ES-tree.
12 **if** $\ell'(y) > \ell(y)$ **then**
13 $\ell(y) \leftarrow \ell'(y)$
14* **if** $\ell'(y) > (\alpha + \beta/\tau)R^{d} + \beta$ **then** $\ell(y) \leftarrow \infty$
15 **foreach** *neighbor* x *of* y **do**
16 update key of y in heap $N(x)$ to $\ell(y) + w(x, y)$
17 insert x into heap Q with key $\ell(x)$ if Q does not already contain x

is an edge between any node y and its parent as in Line 11 of Algorithm 2.2; i.e., its an edge (y, z') for some node $z' = \arg\min_z(\ell(z) + w(y, z))$.)

Observation 2.3.9. *The following holds for the monotone ES-tree:*

(1) The level of a node never decreases.

(2) An edge can only become stretched when it is inserted.

(3) As long as a node x is stretched, its level does not change.

(4) For every tree edge (u, v) (where v is the parent of u), $\ell(u) \geq \ell(v) + w(u, v)$.

Proof. The only places in the algorithm where the level of a node is modified are in Line 4 during the initialization and in Line 13. The if-condition in Line 12 guarantees that the level of a node never decreases and thus (1) holds. Furthermore, whenever the level of a node y increases in Line 13 we have $\ell(y) = \min_z(\ell(z) + w(y, z)) \leq \ell(z') + w(y, z')$ for every neighbor z' of y. Thus, after such a level increase the edge (y, z') is non-stretched for every neighbor z' of y and so is the node y.

To prove (2), consider an edge (x, y) that becomes stretched. This can only happen if the edge (x, y) was not contained in the graph before and is inserted or if the edge changes from non-stretched to stretched. When (x, y) is non-stretched we have $\ell(x) \leq \ell(y) + w(x, y)$. For (x, y) to become stretched (i.e., for $\ell(x) > \ell(y) + w(x, y)$ to hold) either the left hand side of this inequality has to increase or the right hand side has to decrease. When the left hand side increases, the level of x changes and, as argued above, this implies that (x, y) will be non-stretched. As the level of y is non-decreasing, the right hand side can only decrease when the weight of the edge (x, y) decreases. This can only happen after inserting this edge with a smaller weight.

We now prove (3). Consider a node x that is stretched. As long as it is stretched, the level of x does not increase because, as argued above, each level increase immediately makes x non-stretched.

Finally, we prove (4). Consider an edge (u, v) such that v is the parent of u. It is easy to see that $\ell(u) \geq \ell(v) + w(u, v)$ as long as v stays the parent of u because the level of u increases if and only if the level of v or the weight of the edge (u, v) increases. In such a case we have $\ell(u) = \ell(v) + w(u, v)$. The only other possibility for the right hand side of the inequality to change is when the weight of the edge (u, v) decreases, which can happen after an insertion. But decreasing this value does not invalidate the inequality. □

We now prove that the monotone ES-tree provides an $(\alpha + \beta/\tau, \beta)$-approximation of the true distance if it runs on an (α, β, τ)-locally persevering emulator. We use an inductive argument to show that, after having processed the i-th deletion of an edge in $\mathcal{G}$, the level of every node x is a $(\alpha + \beta/\tau, \beta)$-approximation of the distance of x to the root, i.e., $d_{G_i}(x, r) \leq \ell_i(x) \leq (\alpha + \beta/\tau) d_{G_i}(x, r) + \beta$. The intuition why this should be correct is as follows: If the monotone ES-tree gives the desired approximation before

a deletion in $\mathscr{G}$ and the deletion does not cause an edge in $\mathscr{H}$ to become stretched, then the structure of the monotone ES-tree is similar to the ES-tree and the same argument that we use for the ES-tree should show that the monotone ES-tree still gives the desired approximation. If, however, an edge becomes stretched in $\mathscr{H}$, then the level of the affected node does not change anymore and, thus, as distances in decremental graphs never decrease, should still give the desired approximation. This intuition is basically correct, but the correctness proof also requires the emulator to be persevering, as a persevering path does not contain any inserted edge and, thus, no stretched edges.

Remember that processing an edge deletion in $\mathscr{G}$ might mean processing a series of updates in $\mathscr{H}$. We will first show that the approximation guarantee holds for every node that is *stretched* after the monotone ES-tree has processed the i-th deletion. Afterwards we will show that it holds for *every* node.

Lemma 2.3.10. *Let $0 < i \leq k$ and assume that $\ell_{i-1}(x') \leq (\alpha + \beta/\tau) \cdot d_{G_{i-1}}(x', r) + \beta$ for every node x' with $\ell_{i-1}(x') \neq \infty$. Then $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$ for every stretched node x.*

Proof. Here we need the assumption that the monotone ES-tree sees the updates in the emulator caused by a single edge deletion in a specific order, namely such that all edge insertions can be processed before the edge weight increases and edge deletions. Since x is stretched, there must have been a previous insertion of an edge (x, y) incident to x such that x is stretched since the time this edge was inserted (see Observation 2.3.9(2)). Let $\ell'(x)$ denote the level of x after the insertion of (x, y) has been processed. By Observation 2.3.9, nodes do not change their level as long as they are stretched and therefore $\ell_i(x) = \ell'(x)$.

We now show that $\ell_i(x) = \ell'(x) = \ell_{i-1}(x)$. The insertion of (x, y) could either happen at time i or at some earlier time (i.e., either it was caused by the i-th edge deletion or by a previous edge deletion). If the insertion was caused by a previous edge deletion we clearly have $\ell_{i-1}(x) = \ell'(x)$ because the level of x has not changed since this insertion. Consider now the case that the insertion was caused by the i-th edge deletion. Recall that all insertions caused by the i-th deletion are processed *before* any other updates of the emulator are processed. Since edge insertions do not change the level of any node, we have $\ell'(x) = \ell_{i-1}(x)$. In both cases we have $\ell'(x) = \ell_{i-1}(x)$ and thus $\ell_i(x) = \ell_{i-1}(x)$. Since $\ell_i(x) \neq \infty$, we have $\ell_{i-1}(x) \neq \infty$. It follows that

$$\ell_i(x) = \ell_{i-1}(x) \leq (\alpha + \beta/\tau) \cdot d_{G_{i-1}}(x, r) + \beta \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$$

as desired. The first inequality above follows from the assumptions of the lemma, and the second one is because $\mathscr{G}$ is a decremental graph in which distances never decrease. □

In order to prove the approximation guarantee for non-stretched nodes we have to exploit the properties of the (α, β, τ)-locally persevering emulator $\mathscr{H}$. In the classic

ES-tree the level of two nodes differs by at most the weight of a path connecting them—modulo some technical conditions that arise for ES-trees of limited depth. In the monotone ES-tree this is only true for persevering paths (see Lemma 2.3.12). Before we can show this we need an even simpler property of the monotone ES-tree: If two nodes are connected by an edge that is not stretched, then their levels differ by at most the weight of the edge connecting them. Again, in the classic ES-tree this holds for any edge.

Lemma 2.3.11. *Consider any $0 \leq i \leq k$ and any $(x, y) \in E(H_i)$. We have*

$$\ell_i(x) \leq \ell_i(y) + w_i(x, y)$$

if $\ell_i(y) + w_i(x, y) \leq (\alpha + \beta/\tau)R^d + \beta$ and either (a) $i = 0$ or (b) $i \geq 1$, $\ell_{i-1}(x) \neq \infty$ and (x, y) is not stretched.

Proof. Note that no edge in H_0 is stretched. Thus, (x, y) is not stretched for $i \geq 0$. Hence, we either have $\ell_i(x) \leq \ell_i(y) + w_i(x, y)$ as desired or $\ell_i(x) = \infty$. Thus, we only have to argue that $\ell_i(x) \neq \infty$.

Assume by contradiction that $\ell_i(x) = \infty$. As $\ell_{i-1}(x) \neq \infty$, the level of x is not changed while the monotone ES-tree processes the insertions in $\mathscr{H}$ caused by the i-th deletion in $\mathscr{G}$. Thus, the only possibility for the level to be increased to ∞ is when the monotone ES-tree processes the edge deletions and edge weight increases. For every node v, let $\ell'(v)$ and $w'(u, v)$ denote the level of every node v and the weight of every edge (u, v) directly after the level of x has been increased to ∞. Since $\ell'(x) = \infty$ it must be the case that $\min_z(\ell'(z) + w'(x, z)) > (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ and therefore also $\ell'(y) + w'(x, y) > (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. But since levels and edge weights never decrease, we also have $\ell'(y) + w'(x, y) \leq \ell_i(y) + w_i(x, y) \leq (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ which contradicts the inequality we just derived. Therefore it cannot be the case that $\ell'(x) = \infty$. □

Lemma 2.3.12. *For every path π from a node x to a node z that (1) is persevering up to time i and (2) has the property that $\ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^d + \beta$, we have $\ell_i(x) \leq \ell_i(z) + w_i(\pi)$.*

Proof. The proof is by induction on i and the length of the path π. The claim is clearly true if $i = 0$ or the path has length 0. Consider now the induction step. Let (x, y) denote the first edge on the path. Let π' denote the subpath of π from y to z. Note that $\ell_i(z) + w_i(\pi') \leq \ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. Therefore we may apply the induction hypothesis on y and get that $\ell_i(y) \leq \ell_i(z) + w_i(\pi')$. Thus, we get

$$\ell_i(y) + w_i(x, y) \leq \ell_i(z) + w_i(\pi') + w(x, y) = \ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta\,.$$

By the definition of persevering paths, every edge (u, v) on π has always existed in $\mathscr{H}$ since the beginning. Therefore the edge (x, y) has never been inserted which means that (x, y) is not stretched by Observation 2.3.9(2). Since levels and edge weights are non-decreasing we have $\ell_{i-1}(z) + w_{i-1}(\pi) \leq \ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. By the induction hypothesis for $i - 1$ this implies that $\ell_{i-1}(x) \leq \ell_{i-1}(z) + w_{i-1}(\pi) \neq \infty$. We therefore may apply Lemma 2.3.11 and get that $\ell_i(x) \leq \ell_i(y) + w_i(x, y) \leq \ell_i(z) + w_i(\pi)$. □

Using the property above, we would ideally like to do the following: We would like to split a shortest path from x to the root r into subpaths of length $\leq \tau$ and replace each subpath by a persevering path such that the length of each subpath and the persevering path by which it is replaced are approximately the same. Repeated applications of the inequality of Lemma 2.3.12 would then allow us to bound the level of x. However, this approach alone does not work because the definition of a locally persevering emulator does not always guarantee the existence of a persevering path. Instead of a persevering path, the locally persevering emulator might also provide us with a shortest path of G_i that is contained in the current emulator H_i. In principle this is a nice property because a shortest path is even better than an approximate shortest path. But the problem now is that nodes on this path could be stretched and only for non-stretched nodes the difference in levels of two nodes can be bounded by the weight of the edge between them. We can resolve this issue by induction on the distance to r, which allows us to use the contained path only partially.

Lemma 2.3.13 (Correctness)**.** *For every node x and every $0 \leq i \leq k$, $\ell_i(x) \geq d_{G_i}(x, r)$ and if $d_{G_i}(x, r) \leq R^d$, then $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$.*

Proof. We start with a proof of the first inequality $\ell_i(x) \geq d_{G_i}(x, r)$. Consider the (weighted) path π from x to the root r in the monotone ES-tree. Recall that the parent of node v is a node $u = \arg\min_z(\ell(z) + w(y, z))$ as in Line 11 in Algorithm 2.2. For every edge (u, v) on this path, where v is the parent of u, we have $\ell_i(u) \geq \ell_i(v) + w_i(u, v)$ by Observation 2.3.9(4). By repeated applications of this inequality for every edge on π we get $\ell_i(x) \geq w_i(\pi) + \ell_i(r) = w_i(\pi)$ (since the level of the root r is always 0). Since π is a path in H_i we have $w_i(\pi) \geq d_{G_i}(x, r)$ because a locally persevering emulator never underestimates the true distance by definition.

We now prove the second inequality $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$ if $d_{G_i}(x) \leq R^{\mathrm{d}}$. The proof is by induction on i and the distance of x to r in G_i.

The claim is clearly true if x is the root node r itself. If $i \geq 1$, then note that $d_{G_{i-1}}(x, r) \leq d_{G_i}(x, r) \leq R^{\mathrm{d}}$ and therefore, by the induction hypothesis for $i - 1$, we have $\ell_{i-1}(x) \neq \infty$. Therefore we may apply Lemma 2.3.10 which means that the desired inequality holds if x is stretched. Thus, from now on we assume that $x \neq r$ and that x is not stretched. We distinguish two cases.

Case 1: Consider first the case that there is a shortest path from x to r in G_i such that its first edge (x, y) is contained in (H_i, w_i). Note that $d_{G_i}(y, r) < d_{G_i}(x, r)$. Therefore we may apply the induction hypothesis and get $\ell_i(y) \leq (\alpha+\beta/\tau)\cdot d_{G_i}(y, r)+\beta$. We now want to argue that $\ell_i(x) \leq \ell_i(y) + w_i(x, y)$ by applying Lemma 2.3.11. The edge (x, y) is contained in (H_i, w_i) with weight $w_i(x, y) = d_{G_i}(x, y)$ and thus

$$\begin{aligned}
\ell_i(y) + w_i(x, y) &= \ell_i(y) + d_{G_i}(x, y) \\
&\leq (\alpha + \beta/\tau) \cdot d_{G_i}(y, r) + \beta + d_{G_i}(x, y) \\
&\leq (\alpha + \beta/\tau) \cdot (d_{G_i}(x, y) + d_{G_i}(y, r)) + \beta \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta \qquad (2.5) \\
&\leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta\,. \qquad (2.6)
\end{aligned}$$

Remember that (x, y) is not stretched and if $i \geq 1$, then $\ell_{i-1}(x) \neq \infty$ (as argued above). Using (2.6) we may now apply Lemma 2.3.11 and together with (2.5) get that

$$\ell_i(x) \leq \ell_i(y) + w_i(x, y) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$$

as desired.

Case 2: Consider now the case that for every shortest path from x to r in G_i its first edge is not contained in (H_i, w_i). Define the node z as follows. If $d_{G_i}(x, r) < \tau$, then $z = r$. If $d_{G_i}(x, r) \geq \tau$, then z is a node on a shortest path from x to r in G_i whose distance to x is τ, i.e., $d_{G_i}(x, z) = \tau$ and $d_{G_i}(x, r) = d_{G_i}(x, z) + d_{G_i}(z, r)$. In both cases there is no shortest path from x to z in G_i that is also contained in (H_i, w_i) because every shortest path from x to z can be extended to a shortest path from x to r in G_i and (H_i, w_i) does not contain the first edge of such a path. Since $\mathscr{H}$ is an (α, β, τ)-locally persevering emulator, we know that there is a path π from x to z in (H_i, w_i) that is persevering up to time i such that $w_i(\pi) \leq \alpha d_{G_i}(x, z) + \beta$.

If $z = r$, we have $\ell_i(z) = 0$ and therefore we get

$$\begin{aligned}\ell_i(z) + w_i(\pi) = w_i(\pi) &\leq \alpha d_{G_i}(x, z) + \beta \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta \\ &\leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta\end{aligned}$$

as desired. Consider now the case that $z \neq r$. Since $d_{G_i}(z, r) < d_{G_i}(x, r)$, we may apply the induction hypothesis on z and get that $\ell_i(z) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(z, r) + \beta$. Together with $d_{G_i}(x, z) = \tau$, we get

$$\begin{aligned}\ell_i(z) + w_i(\pi) &\leq (\alpha + \beta/\tau) \cdot d_{G_i}(z, r) + \beta + \alpha d_{G_i}(x, z) + \beta \\ &= (\alpha + \beta/\tau) \cdot d_{G_i}(z, r) + \beta + \alpha d_{G_i}(x, z) + \beta \cdot d_{G_i}(x, z)/\tau \\ &= (\alpha + \beta/\tau) \cdot d_{G_i}(z, r) + \beta + (\alpha + \beta/\tau) \cdot d_{G_i}(x, z) \\ &= (\alpha + \beta/\tau) \cdot (d_{G_i}(x, z) + d_{G_i}(z, r)) + \beta \\ &= (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta \\ &\leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta\,.\end{aligned}$$

The last equation follows from the definition of z.

In both cases we have $\ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta$. Since π is persevering up to time i, we may apply Lemma 2.3.12 and get the approximation guarantee:

$$\ell_i(x) \leq \ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta\,. \qquad \square$$

Finally, we provide the running time analysis. In principle we use the same charging argument as for the classic ES-tree. We only have to deal with the fact that the degree of a node might change over time in the dynamic emulator.

Lemma 2.3.14 (Running Time)**.** *For k deletions in $\mathscr{G}$, the monotone ES-tree has a total update time of $O(\phi_k(\mathscr{H}) \log n + |E_k(\mathscr{H})| \cdot ((\alpha + \beta/\tau)R^d + \beta) \log n)$, where $E_k(\mathscr{H})$ is the set of all edges ever contained in $\mathscr{H}$ up to time k.*

Proof. We first bound the time needed for the initialization. Using Dijkstra's algorithm, the shortest paths tree can be computed in time $O(|E(H_0)| + n \log n)$, which is $O(|E_k(\mathscr{H})| \log n)$

We now bound the time for processing all edge deletions in $\mathscr{G}$. Remember that the monotone ES-tree runs on the emulator $\mathscr{H}$. An edge deletion in $\mathscr{G}$ could result in several updates in the emulator $\mathscr{H}$. All of these updates have to be processed by the monotone ES-tree with time $O(\log n)$ per update plus the time needed for running the procedure UpdateLevels. Therefore the total update time is $O(\phi_k(\mathscr{H}) \log n)$, where $\phi_k(\mathscr{H})$ is the total number of updates in $\mathscr{H}$, plus the cumulated time for running the reconnection procedure.

We now bound the running time of the procedure UpdateLevels. Here, the well-known level-increase argument works. We define the *dynamic degree* of a node x by $\deg_{\mathscr{H}}(x) = |\{(x, y) \mid (x, y) \in E_k(\mathscr{H})\}|$. Clearly, the dynamic degree never underestimates the current degree of a node in the emulator. We charge time $O(\deg_{\mathscr{H}}(x) \log(x))$ to every level increase of a node x and time $O(\log n)$ to every update in $\mathscr{H}$.

We now argue that this charging covers all costs in the procedure UpdateLevels. Consider a node x that is processed in the while loop of the procedure UpdateLevels after some update in the emulator. Now the following holds: If the level of x increases, the monotone ES-tree has to spend time $O(\deg_{\mathscr{H}}(x) \log(x))$ because $\deg_{\mathscr{H}}(x)$ bounds the current degree of x in the emulator. If the level of x does not increase, the monotone ES-tree has to spend time $O(\log n)$. We now only have to argue that the cost of $O(\log n)$ in the second case is already covered by our charging scheme.

There are two possibilities why x is in the heap. The first one is that x is processed directly after the deletion or weight increase of an edge (x, y). The second one is that it was put there by one of its neighbors. In the first situation we can charge the running time of $O(\log n)$ to the weight increase (or delete) operation. Consider now the second situation: the level of a node y increases and its neighbor x is put into the heap for later processing. Later on x is processed but its level does not increase. Then we can charge the running time of $O(\log n)$ to the time $O(\deg_{\mathscr{H}}(x) \log n)$ that we already charge to y.

Since the monotone ES-tree is only maintained up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$, at most $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ level increases are possible for every node. Thus, the total update time of the monotone ES-tree is

$$O(\phi_k(\mathscr{H}) \log n + \sum_{x \in U} \deg_{\mathscr{H}}(x)((\alpha + \beta/\tau)R^{\mathrm{d}} + \beta) \log n) .$$

As $\sum_{x \in U} \deg_{\mathscr{H}}(x) \leq 2|E_{\mathscr{H}}(U)|$, this becomes

$$O(\phi_k(\mathscr{H}) + E_{\mathscr{H}}(U)((\alpha + \beta/\tau)R^{\mathrm{d}} + \beta) \log n) . \qquad \square$$

Eliminating the $\log n$**-factor** The factor $\log n$ in the running time of Lemma 2.3.14 comes from using a heap Q and, for every node u, a heap $N(u)$. We now want to avoid using these heaps and only charge $O(\deg_{\mathscr{H}}(u))$ to every level increase of a

node u and time $O(1)$ to every update in $\mathscr{H}$. King [79] explained how to eliminate the $\log n$-factor for the classic ES-tree. However, we cannot use the same modified data structures as King because of the possibility of insertions and edge weight increases.

First we explain how to avoid the heap Q. Observe that, every time we increase the level of a node, it suffices to increase the level by only 1. Thus, instead of a heap for Q we can also use a simple queue, implemented with a list that allows us to retrieve and remove its first element and to append an element at its end.

Now we explain how to avoid the heap $N(u)$ of every node u. Remember that we only want to increase the level of a node u if there is no neighbor v of u such that

$$\ell(v) + w(u, v) \leq \ell(u)\,. \tag{2.7}$$

Therefore we maintain a counter $c(u)$ for every node u such that $c(u) = |\{v \mid \ell(v) + w(u, v) \leq \ell(u)\}|$.[17] If the counters are correctly maintained, we can simply check whether $c(u)$ is 0 to determine whether the level of u has to increase (which replaces Lines 11 and 12 of Algorithm 2.2). For a node u and its neighbor v the status of Inequality (2.7) only changes (i.e., the inequality starts or stops being satisfied) in the following cases:

- The level of u or the level of v increases.
- The weight of the edge (u, v) increases.
- The edge (u, v) is inserted (thus v becomes a neighbor of u).
- The edge (u, v) is deleted (thus v stops being a neighbor of v).

Note that for two nodes u and v we can check whether they satisfy Inequality (2.7) in constant time. Thus, we can efficiently maintain the counters as follows:

- Every time we update an edge (u, v) (by an insertion, deletion, or weight increase), we check in constant time whether Inequality (2.7) holds before the update and whether it holds after the update. Then we increase or decrease $c(v)$ and $c(u)$ if necessary. These operations take constant time, which we charge to the update in $\mathscr{H}$.
- Every time $\ell(u)$ increases, we recompute $c(u)$. This takes time $O(\deg_{\mathscr{H}}(u))$. Furthermore, for every neighbor v of u, we check in constant time whether Inequality (2.7) holds before the update and whether it holds after the update. Then we increase or decrease $c(v)$ if necessary. This takes constant time for every neighbor of u and thus time $O(\deg_{\mathscr{H}}(u))$ for all of them. We can charge the running time $O(\deg_{\mathscr{H}}(u))$ to the level increase of u.

Having explained how to maintain the counters, the remaining running time analysis is the same as in Lemma 2.3.14. The improved running time can therefore be stated as follows.

[17] The idea of maintaining this kind of counter has previously been used by Brim et al. [27] in the context of mean-payoff games.

Lemma 2.3.15 (Improved Running Time)**.** *For k deletions in $\mathscr{G}$, the monotone ES-tree can be implemented with a total update time of $O(\phi_k(\mathscr{H}) + |E_k(\mathscr{H})| \cdot ((\alpha + \beta/\tau)R^d + \beta))$, where $E_k(\mathscr{H})$ is the set of all edges ever contained in $\mathscr{H}$ up to time k.*

Note that the solution proposed above does not allow us to retrieve the parent of every node in the tree in constant time. This would be desirable because then, for every node v, we could not only get the approximate distance of v to the root in constant time, but also a path of corresponding or smaller length in time proportional to the length of this path.

We can achieve this property as follows. For every node u we maintain a list $L(u)$ of nodes. *Every time* a node u and one of its neighbors v start to satisfy Inequality (2.7), v is appended to $L(u)$. Note that it is *not* always the case that u and all nodes v in the list $L(u)$ satisfy Inequality (2.7). We just have the guarantee that they satisfied it at some previous point in time. However, the converse is true: if u and its neighbor v currently satisfy Inequality (2.7), then v is contained in $L(u)$. Using the same argument as above for maintaining the counters, the running time for appending nodes to the lists is paid for by charging $O(1)$ to every update in $\mathscr{H}$ and $O(\deg_{\mathscr{H}}(u))$ to every level increase of a node u.

We can now decide whether the level of a node u has to increase as follows (this replaces Lines 11 and 12 of Algorithm 2.2). Look at the first node v in the list $L(u)$. If u and v *still* satisfy Inequality (2.7), the level of u does not have to increase. Otherwise, we retrieve and remove the first element from the list until we find a node v such that u and v satisfy Inequality (2.7). If no such node v can be found in the list, then the list will be empty after this process and we know that the level of u has to increase. Otherwise, the first node in the list $L(u)$ serves as the parent of u in the tree. The constant running time for reading and removing the first node can be charged to the previous appending of this node to $L(u)$.

Note that the list $L(u)$ of each node u might require a lot of space because some nodes might appear several times. If we want to save space, we can do the following. For every node u we maintain a set $S(u)$ that stores for every neighbor of u whether it is contained in $L(u)$. Every time we add or remove a node from $L(u)$ we also add or remove it from $S(u)$. Before adding a node to $L(u)$ we additionally check whether it is already contained in $S(u)$ and thus also in $L(u)$. We implement $S(u)$ with a dynamic dictionary using dynamic perfect hashing [40] or cuckoo hashing [103]. This data structure needs time $O(1)$ for look-ups and expected amortized time $O(1)$ for insertions and deletions. Thus, the running time bound of Lemma 2.3.15 will still hold in expectation. Furthermore, for every node u, the space needed for $L(u)$ and $S(u)$ is bounded by $O(\deg_{\mathscr{H}}(u))$. However, this solution is not deterministic anymore.

2.3.3 From Approximate SSSP to Approximate APSP

In the following, we show how a combination of approximate decremental SSSP data structures can be turned into an approximate decremental APSP data structure. We

follow the ideas of Roditty and Zwick [113], who showed how to obtain approximate APSP from *exact* SSSP. We remark that one can obtain an efficient APSP data structure from this reduction, if the running time of the (approximate) SSSP data structure depends on the distance range that it covers in a specific way.

We first define an approximate version of the center cover data structure and show how such a data structure can be obtained from an approximate decremental SSSP data structure by marginally worsening the approximation guarantee. We slightly modify the notions of a center cover and a center cover data structure we gave in Section 2.2.3, where we reviewed the algorithmic framework of Roditty and Zwick [113]. The main idea behind their APSP data structure is to maintain $\log n$ instances of center cover data structures such that the instance p can answer queries for the approximate distance of two nodes x and y if the distance between them is in the range from 2^p to 2^{p+1}. Arbitrary distance queries can then be answered by performing binary search over the instances to determine p. We will follow this approach using approximate instead of exact data structures.

Definition 2.3.16 (Approximate center cover)**.** *Let U be a set of nodes in a graph G, let R^c be a positive integer, the* cover range*, and let $\alpha \geq 1$ and $\beta \geq 0$. We say that a node x is (α, β)-*covered *by a node $c \in U$ in G if $d_G(x, c) \leq \alpha R^c + \beta$. We say that U is an (α, β)-*approximate center cover of G with parameter R^c *if every node x that is in a connected component of size at least R^c is (α, β)-covered by some node $c \in U$ in G.*

Definition 2.3.17. *An (α, β)-*approximate center cover data structure *with* cover range parameter R^c *and* distance range parameter R^d *for a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ maintains, for every $0 \leq i \leq k$, a set of* centers *$C_i = \{1, 2, \ldots, l\}$ and a set of nodes $U_i = \{c_i^1, c_i^2, \ldots, c_i^l\}$ such that U_i is an (α, β)-approximate center cover of G_i with parameter R^c. For every center $j \in C_i$ and every $0 \leq i \leq k$, we call c_i^j the* location *of center j in G_i and for every node x we say that x is (α, β)-covered by j in G_i if x is (α, β)-covered by c_i^j in G_i. After the i-th edge deletion (where $0 \leq i \leq k$), the data structure provides the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G_i.*
- Distance(j, x)*: Return an estimate $\delta_i(c_i^j, x)$ of the distance between the location c_i^j of center j and the node x such that $\delta_i(c_i^j, x) \leq \alpha d_{G_i}(c_i^j, x) + \beta$, provided that $d_{G_i}(c_i^j, x) \leq R^d$. If $d_{G_i}(c_i^j, x) > R^d$, then either return $\delta_i(c_i^j, x) = \infty$ or return $\delta_i(c_i^j, x) \leq \alpha d_{G_i}(c_i^j, x) + \beta$.*
- FindCenter(x)*: If x is in a connected component of size at least R^c, then return a center j (with current location c_i^j) such that $d_{G_i}(x, c_i^j) \leq \alpha R^c + \beta$. If x is in a connected component of size less than R^c, then either return $\perp$ or return a center j such that $d_{G_i}(x, c_i^j) \leq \alpha R^c + \beta$.*

The total update time *is the total time needed for performing all k delete operations and the initialization and the* query time *is the worst-case time needed to answer a single distance or find center query.*

We now show how to obtain an approximate center cover data structure that is correct with high probability, which means that, with small probability, the operation FindCenter(x) might return $\perp$ although x is in a connected component of size at least R^{c}.

Lemma 2.3.18 (Approximate SSSP implies approximate center cover)**.** *Let R^c and R^d be parameters such that $R^c \leq R^d$. If there are (α, β)-approximate decremental SSSP data structures with distance range parameters R^c and R^d for some $\alpha \geq 1$ and $\beta \geq 0$ that have constant query times and total update times of $T(R^c)$ and $T(R^d)$, respectively (where $T(R^d)$ is $\Omega(n)$), then there is an (α, β)-approximate center cover data structure that is correct with high probability, has constant query time and an expected total update time of $O((T(R^d)n \log n)/R^c)$.*

Proof. Let $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph. It is well-known (see Lemma 1.3.2) that, by random sampling, we can obtain a set $U = \{c^1, c^2, \ldots, c^l\}$ of expected size $O(n \log n/R^{\mathrm{c}})$ that is a center cover of G_i for every $i \leq k$ with high probability. Clearly, every center cover is also an (α, β)-approximate center cover. Thus, U is an (α, β)-approximate center cover of G_i for every $0 \leq i \leq k$. Throughout all deletions, the set $C = \{1, 2, \ldots, l\}$ will serve as the set of centers and each center j will always be located at the same node c^j.

We use the following data structures: For every center j, we maintain two (α, β)-approximate decremental SSSP data structures with source c^j: for the first one we use the parameter R^{c} and for the second one we use the parameter R^{d}. As there are $O(n \log n/R^{\mathrm{c}})$ centers, the total update time for all these SSSP data structures is $O(T(R^{\mathrm{d}})(n \log n)/R^{\mathrm{c}})$. For every node x and every center j, let $\delta_i(x, c^j)$ denote the estimate of the distance between x and the location of center j returned by the second SSSP data structure with source c^j after the i-th edge deletion. For every node x we maintain a set S_x of centers that cover x such that (a) if $d_{G_i}(x, c^j) \leq R^{\mathrm{c}}$, then $j \in S_x$ and (b) for all $j \in S_x$, $\delta_i(x, c^j) \leq \alpha R^{\mathrm{c}} + \beta$.

The set S_x can be implemented by using an array of size $|C| = O((n \log n)/R^{\mathrm{c}})$ for every node x. We initialize S_x in time $O((n \log n)/R^{\mathrm{c}})$ as follows: for every center j, we query $\delta_0(x, c^j)$ and insert j into S_x if $\delta_0(x, c^j) \leq \alpha R^{\mathrm{c}} + \beta$. Since $\delta_0(x, c^j) \leq \alpha d_{G_0}(x, c^j) + \beta$, this includes every center j such that $d_{G_0}(x, c^j) \leq R^{\mathrm{c}}$. To maintain the sets of centers we do the following after every deletion. Remember that for every center j, the first SSSP data structure with source c^j returns every node x such that $\delta_i(x, c^j) \leq \alpha R^{\mathrm{c}} + \beta$ and $\delta_{i+1}(x, c^j) > \alpha R^{\mathrm{c}} + \beta$. For every such node x we remove j from S_x. Note that every center j with $\delta_t(x, c^j) > \alpha R^{\mathrm{c}} + \beta$ (for $0 \leq t \leq k$) can safely be removed S_x because $\delta_t(x, c^j) > \alpha R^{\mathrm{c}} + \beta$ implies $d_{G_t}(x, c^j) > R^{\mathrm{c}}$ and $d_{G_i}(x, c^j) \geq d_{G_t}(x, c^j)$ for all $i \geq t$. We can charge the running time for maintaining the sets of centers to the delete operations in the SSSP data structures. Thus, this running time is already included in the total update time stated above. For every node x, no center is ever added to S_x after the initialization. Thus, in the array representing S_x, we can maintain a pointer to the left-most center time proportional to the size of the array, which is $|C| = O((n \log n)/R^{\mathrm{c}})$.

We now show how to perform the operations of an approximate center cover data structure, as specified in Definition 2.3.17, in constant time. Let i be the index of the last deletion. Given a center j and a node x we answer a query for the distance of x to c^j by returning $\delta_i(x, c^j)$ from the second SSSP data structure of c^j, which gives an (α, β)-approximation of the true distance. Given a node x, we answer a query for finding a nearby center by returning any center j in the set of centers S_x of x. If S_x is empty, we return $\perp$. Note that for every center j in S_x we know that $d_{G_i}(x, c^j) \leq \alpha R^c + \beta$ as required because $d_{G_i}(x, c^j) \leq \delta_i(x, c^j)$. If x is in a connected component of size at least R^c we can ensure that we find a center j in S_x because, by our random choice of centers, we have $d_{G_i}(x, c^j) \leq R^c$ for some center j with high probability. If $d_{G_i}(x, c^j) \leq R^c$, then, S_x contains j. □

We now show why the approximate center cover data structure is useful. If one can obtain an approximate center cover data structure, then one also obtains an approximate decremental APSP data structure with slightly worse approximation guarantee. The proof of this observation follows Roditty and Zwick [113]. In their algorithm, Roditty and Zwick keep a set of nodes U (which we call centers) such that every node (that is in a sufficiently large connected component) is "close" to some node in U. To be able to efficiently find a close center for every node, they maintain, for every node, the nearest node in the set of centers. However, it is sufficient to return *any* center that is close.

Lemma 2.3.19 (Approximate center cover implies approximate APSP)**.** *Assume that for all parameters R^c and R^d such that $R^c \leq R^d$ there is an (α, β)-approximate center cover data structure that has constant query time and a total update time of $T(R^c, R^d)$. Then, for every $0 < \epsilon \leq 1$, there is an $(\alpha + 2\epsilon\alpha^2, 2\beta + 2\alpha\beta)$-approximate decremental APSP data structure with $O(\log \log n)$ query time and a total update time of $\hat{T} = \sum_{p=0}^{\lfloor \log n \rfloor} T(R_p^c, R_p^d)$ where $R_p^c = \epsilon 2^p$ and $R_p^d = \alpha\epsilon 2^p + \beta + 2^{p+1}$ (for $0 \leq p \leq \lfloor \log n \rfloor$).*

The query time can be reduced to $O(1)$ if there is an (α', β')-approximate decremental APSP data structure for some constants α' and β' with constant query time and a total update time of $\hat{T}$.

Proof. The data structure uses $\lceil \log n \rceil$ many instances where the p-th instance is responsible for the distance range from 2^p to 2^{p+1}. For the p-th instance we maintain a center cover data structure using the parameters $R_p^c = \epsilon 2^p$ and $R_p^d = 2^{p+1} + \alpha\epsilon 2^p + \beta$. For every center j and every node x, let $\delta_i^p(c^j, x)$ denote the estimate of the distance between c^j and x provided by the p-th center cover data structure. Let $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph and let i be the index of the last deletion.

For every instance p, we can compute a distance estimate $\hat{\delta}_i^p(x, y)$ for all nodes x and y as follows. Using the center cover data structure, we first check whether there is some center j with location c^j that (α, β)-covers x, i.e., $d_{G_i}(x, c^j) \leq \alpha R_p^c + \beta$. If x is not (α, β)-covered by any center we set $\hat{\delta}_i^p(x, y) = \infty$. Otherwise we query the center cover data structure to get estimates $\delta_i^p(c^j, x)$ and $\delta_i^p(c^j, y)$ of the distances between c^j and x and between c^j and y, respectively. (Remember that these distance

estimates might be ∞.) We now set $\hat{\delta}_i^p(x,y) = \delta_i^p(c^j,x) + \delta_i^p(c^j,y)$. Note that, given p, we can compute $\hat{\delta}_i^p(x,y)$ in constant time. The query procedure will rely on three properties of the distance estimate $\hat{\delta}_i^p(x,y)$.

1. The distance estimate never underestimates the true distance, i.e., $\hat{\delta}_i^p(x,y) \geq d_{G_i}(x,y)$.
2. If $d_{G_i}(x,y) \geq 2^p$ and $\hat{\delta}_i^p(x,y) \neq \infty$, then $\hat{\delta}_i^p(x,y) \leq (\alpha + 2\alpha^2) d_{G_i}(x,y) + 2\beta + 2\alpha\beta$.
3. If x is in a connected component of size at least R_p^{c} and $d_{G_i}(x,y) \leq 2^{p+1}$, then $\hat{\delta}_i^p(x,y) \neq \infty$.

The first property is clearly true if $\hat{\delta}_i^p(x,y) = \infty$ and otherwise follows by applying the triangle inequality (note that $d_{G_i}(c^j,y) \leq \delta_i^p(c^j,y)$ in any case):

$$d_{G_i}(x,y) \leq d_{G_i}(c^j,x) + d_{G_i}(c^j,y) \leq \delta_i^p(c^j,x) + \delta_i^p(c^j,y) = \hat{\delta}_i^p(x,y)\,.$$

Thus, $\hat{\delta}_i^p(x,y)$ never underestimates the true distance. For the second property we remark that if $\hat{\delta}_i^p(x,y) \neq \infty$, it must be the case that we have found a center j with location c^j that (α,β)-covers x. Therefore $d_{G_i}(x,c^j) \leq \alpha R_p^{\mathrm{c}} + \beta$. Furthermore it must be the case that $\delta_i^p(x,c^j) \neq \infty$ and $\delta_i^p(c^j,y) \neq \infty$ and therefore $\delta_i^p(x,c^j) \leq \alpha d_{G_i}(x,c^j) + \beta$ and $\delta_i^p(c^j,y) \leq \alpha d_{G_i}(c^j,y) + \beta$. Now simply consider the following chain of inequalities:

$$\begin{aligned}
\hat{\delta}_i^p(x,y) = \delta_i^p(x,c^j) + \delta_i^p(c^j,y) &\leq \alpha(d_{G_i}(c^j,x) + d_{G_i}(c^j,y)) + 2\beta \\
&\leq \alpha(d_{G_i}(c^j,x) + d_{G_i}(c^j,x) + d_{G_i}(x,y)) + 2\beta \\
&= \alpha(2d_{G_i}(c^j,x) + d_{G_i}(x,y)) + 2\beta \\
&\leq \alpha(2\alpha R_p^{\mathrm{c}} + 2\beta + d_{G_i}(x,y)) + 2\beta \\
&= \alpha(2\alpha\epsilon 2^p + 2\beta + d_{G_i}(x,y)) + 2\beta \\
&\leq \alpha(2\alpha\epsilon d_{G_i}(x,y) + 2\beta + d_{G_i}(x,y)) + 2\beta \\
&= (\alpha + 2\epsilon\alpha^2) d_{G_i}(x,y) + 2\beta + 2\alpha\beta
\end{aligned}$$

We now prove the third property. If x is in a component of size at least R_p^{c}, then, with high probability, it is covered by some center j with location c^j and we have

$$d_{G_i}(c^j,x) \leq \alpha R_p^{\mathrm{c}} + \beta = \alpha\epsilon 2^p + \beta \leq R_p^{\mathrm{d}}\,.$$

Therefore we get $\delta_i^p(c^j,x) \leq \alpha d_{G_i}(c^j,x) + \beta < \infty$. Furthermore, we have

$$d_{G_i}(c^j,y) \leq d_{G_i}(c^j,x) + d_{G_i}(x,y) \leq \alpha\epsilon 2^p + \beta + 2^{p+1} = R_p^{\mathrm{d}}\,.$$

which gives $\delta_i^p(c^j,y) \leq \alpha d_{G_i}(c^j,y) + \beta < \infty$. As both of its components are not ∞, the sum $\hat{\delta}_i^p(x,y) = \delta_i^p(c^j,x) + \delta_i^p(c^j,y)$ is also not ∞, as desired.

A query time of $O(\log n)$ is immediate as we can simply return the minimum of all distance estimates $\hat{\delta}_i^p(x,y)$. A query time of $O(\log\log n)$ is possible because

of the following idea: If $d_{G_i}(x,y) \neq \infty$, it is sufficient to find the minimum index p such that $\hat{\delta}_i^p(x,y) \neq \infty$. This minimum index can be found by performing binary search over all $\log n$ possible indices. Furthermore, the query time can be reduced to $O(1)$ if there is a second (α',β')-approximate decremental APSP data structure with constant query time for some constants α' and β', We first compute the distance estimate $\delta_i'(x,y)$ of the second data structure for which we know that $d_{G_i}(x,y) \in [\delta_i'(x,y)/\alpha' - \beta', \delta_i'(x,y)]$. Now there is only a constant number of indices p such that $\{2^p,\dots,2^{p+1}\} \cap [\delta_i'(x,y)/\alpha' - \beta', \delta_i'(x,y)] \neq \emptyset$. For every such index we compute $\hat{\delta}_i^p(x,y)$ and return the minimum distance estimate obtained by this process. □

Finally, we show how to obtain an approximate decremental APSP data structure from an approximate decremental SSSP data structure if the approximation guarantee is of the form $(\alpha+\epsilon,\beta)$. In that case we can avoid the worsening of the approximation guarantee of Lemma 2.3.18.

Lemma 2.3.20. *Assume that for some $\alpha \geq 1$ and $\beta \geq 0$, every $0 < \epsilon \leq 1$, and all $0 \leq R^d$ there is an $(\alpha+\epsilon,\beta)$-approximate decremental SSSP data structure with distance range parameter R^d that has constant query time and a total update time of $T'(R^d,\epsilon)$. Then there is an $(\alpha+\epsilon,\beta)$-approximate decremental APSP data structure with a query time of $O(\log\log n)$ and a total update time of*

$$\hat{T} = \sum_{p=0}^{\lfloor \log n \rfloor} (T'(R_p^d,\hat{\epsilon})n\log n)/R_p^c + nT'(\hat{R}^d,\hat{\epsilon})$$

where $\hat{\epsilon} = \epsilon/(18\alpha^2)$, $\hat{R}^d = (4\alpha+8\beta)/\hat{\epsilon}$, $R_p^c = \hat{\epsilon}2^p$ and $R_p^d = \alpha\hat{\epsilon}2^p + \beta + 2^{p+1}$ (for $0 \leq p \leq \lfloor \log n \rfloor$).

The query time can be reduced to $O(1)$ if there is an (α',β')-approximate decremental APSP data structure for some constants α' and β' with constant query time and a total update time of $\hat{T}$.

Proof. By combining Lemma 2.3.18 with Lemma 2.3.19 the approximate decremental SSSP data structure implies that there is an $(\hat{\alpha},\hat{\beta})$-approximate decremental APSP data structure where $\hat{\alpha} = (\alpha+\hat{\epsilon}) + 2\hat{\epsilon}(\alpha+\hat{\epsilon})^2$ and $\hat{\beta} = 2\beta + 2(\alpha+\hat{\epsilon})\beta$. This APSP data structure has a query time of $O(\log\log n)$ and a total update time of

$$\sum_{p=0}^{\lfloor \log n \rfloor} (T'(R_p^{\mathrm{d}},\hat{\epsilon})(n\log n)/R_p^{\mathrm{c}}\,.$$

By Lemma 2.3.19 the query time can be reduced to $O(1)$ if, for some constants α' and β', there is an (α',β')-approximate decremental APSP data structure with constant query time and a total update time of $\hat{T}$.

The data structure above provides, for every decremental graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$ and all nodes x and y, a distance estimate $\delta_i(x,y)$ such that $d_{G_i}(x,y) \leq \delta_i(x,y) \leq \hat{\alpha}d_{G_i}(x,y) + \hat{\beta}$ after the i-th deletion. By our choice of $\hat{\epsilon} = \epsilon/(18\alpha^2)$ we get

$$\hat{\alpha} = (\alpha+\hat{\epsilon}) + 2\hat{\epsilon}(\alpha+\hat{\epsilon})^2 \leq \alpha + \hat{\epsilon}\alpha^2 + 2\hat{\epsilon}(\alpha+\alpha)^2 = \alpha + 9\hat{\epsilon}\alpha^2 = \alpha + \epsilon/2$$

and

$$\hat{\beta} = 2\beta + 2(\alpha + \hat{\epsilon})\beta \leq 2\beta + 2(\alpha + 1)\beta = (2\alpha + 4)\beta$$

Thus, if $d_{G_i}(x, y) \geq (4\alpha + 8\beta)/\epsilon$, then

$$\begin{aligned}\delta_i(x, y) &\leq (\alpha + \epsilon/2)d_{G_i}(x, y) + (2\alpha + 4)\beta \leq (\alpha + \epsilon/2)d_{G_i}(x, y) + \epsilon d_{G_i}(x, y)/2 \\ &= (\alpha + \epsilon)d_{G_i}(x, y)\,.\end{aligned}$$

Additionally, we use a second approximate decremental APSP data structure to deal with distances that are smaller than $(4\alpha + 8\beta)/\epsilon$ (which is less than $(4\alpha + 8\beta)/\hat{\epsilon}$). For this data structure we simply maintain an $(\alpha + \hat{\epsilon}, \beta)$-approximate decremental SSSP data structure for every node with distance range parameter $\hat{R}^{\mathrm{d}} = (4\alpha + 8\beta)/\hat{\epsilon}$. We answer distance queries by returning the minimum of the distance estimates provided by both APSP data structures. As both APSP data structures never underestimate the true distance, the minimum of both distance estimates gives the desired $(\alpha + \epsilon, \beta)$-approximation. □

2.3.4 Putting Everything Together

In the following we show how the monotone ES-tree of Lemma 2.3.8 together with the locally persevering emulator of Lemma 2.3.3 can be used to obtain $(1 + \epsilon, 2)$- and $(2 + \epsilon, 0)$-approximate decremental APSP data structures with $\tilde{O}(n^{5/2}/\epsilon^2)$ total update time. These results are direct consequences of the previous parts of this section. We first show how to obtain a $(1 + \epsilon, 2)$-approximate decremental SSSP data structure. Using Lemma 2.3.20 we then immediately obtain a $(1 + \epsilon, 2)$-approximate decremental APSP data structure.

Corollary 2.3.21 ($(1+\epsilon, 2)$-approximate monotone ES-tree)**.** *Given a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator $\mathscr{H}$ (as in Lemma 2.3.3), there is a $(1 + \epsilon, 2)$-approximate decremental SSSP data structure for every distance range parameter R^d that is correct with high probability, has constant query time, and an expected total update time of $O(n^{3/2} \log n/\epsilon + n^{3/2}R^d \log n)$, where the time for maintaining $\mathscr{H}$ is not included.*

Proof. Let $\mathscr{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph and let $\mathscr{H}$ be the $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of Lemma 2.3.3. By Lemma 2.3.8 there is an approximate decremental SSSP data structure for every source node r and every distance range parameter R^{d}. Let $\delta_i(x, r)$ denote the estimate of the distance between x and r provided after the i-th edge deletion in $\mathscr{G}$. By Lemma 2.3.8 we have $d_{G_i}(x, c) \leq \delta_i(x, c)$, and furthermore, if $d_{G_i}(x, c) \leq R^{\mathrm{d}}$, then

$$\delta_i(x, c) \leq (1 + 2/(\lceil 2/\epsilon \rceil))d_{G_i}(x, c) + 2 \leq (1 + \epsilon)d_{G_i}(x, c) + 2\,.$$

By Lemma 2.3.3, the number of edges ever contained in the emulator is $|E_k(\mathscr{H})| = O(n^{3/2} \log n)$ and the total number of updates in $\mathscr{H}$ is $\phi_k(\mathscr{H}) = O(n^{3/2} \log n/\epsilon)$. Therefore, by Lemma 2.3.8, the total update time of the approximate decremental SSSP

data structure is

$$O(\phi_k(\mathscr{H}) + |E_k(\mathscr{H})| \cdot ((\alpha + \beta/\tau)R^{\mathrm{d}} + \beta)) = O((n^{3/2}\log n)/\epsilon + (n^{3/2}\log n)\cdot((1+\epsilon)R^{\mathrm{d}} + 2)) = O((n^{3/2}\log n)/\epsilon + n^{3/2}R^{\mathrm{d}}\log n)\,. \quad \square$$

Theorem 2.3.22 (Main result of Section 2.3: Randomized $(1+\epsilon, 2)$-approximation with truly-subcubic total update time)**.** *For any $0 < \epsilon \leq 1$, there is a $(1+\epsilon, 2)$-approximate decremental APSP data structure with constant query time and an expected total update time of $O((n^{5/2}\log^3 n)/\epsilon)$ that is correct with high probability.*

Proof. We set $\hat{\epsilon} = \epsilon/18$. Let $\mathscr{H}$ denote the $(1, 2, 2/\hat{\epsilon})$-locally persevering emulator of Lemma 2.3.3. The total update time for maintaining $\mathscr{H}$ is $O(mn^{1/2}\log n/\epsilon)$. Since $m \leq n^2$ this is within the claimed total update time. By Corollary 2.3.21 we can use $\mathscr{H}$ to maintain, for every distance range parameter R^{d}, a $(1+\hat{\epsilon}, 2)$-approximate decremental SSSP data structure that has constant query time and a total update time of $T(R^{\mathrm{d}}) = O((n^{3/2}\log n)/\epsilon + n^{3/2}R^{\mathrm{d}}\log n)$.

Using $\alpha = 1$ and $\beta = 2$, it follows from Lemma 2.3.20 that there is a $(1+\epsilon, 2)$-approximate decremental APSP data structure whose total update time is proportional to

$$\sum_{p=0}^{\lfloor \log n \rfloor} (T(R_p^{\mathrm{d}})n\log n)/R_p^{\mathrm{c}} + T(\hat{R}^{\mathrm{d}})n = \sum_{p=0}^{\lfloor \log n \rfloor} ((n^{3/2}\log n)/\hat{\epsilon} + n^{3/2}R_p^{\mathrm{d}}\log n)(n\log n)/R_p^{\mathrm{c}}) + ((n^{3/2}\log n)/\hat{\epsilon} + n^{3/2}\hat{R}^{\mathrm{d}}\log n)n)$$

where $\hat{\epsilon} = \epsilon/18$, $\hat{R}^{\mathrm{d}} = 12/\hat{\epsilon}$, $R_p^{\mathrm{c}} = \hat{\epsilon}2^p$, and $R_p^{\mathrm{d}} = \alpha\hat{\epsilon}2^p + 2^{p+1} + 2$ (for $0 \leq p \leq \lfloor \log n \rfloor$). Note that $1/\hat{\epsilon} = O(1/\epsilon)$, $\hat{R}^{\mathrm{d}} = O(1/\epsilon)$ and $R_p^{\mathrm{d}}/R_p^{\mathrm{c}} = O(1/\epsilon)$. Therefore the total update time is $O((n^{5/2}\log^3 n)/\epsilon)$.

The query time of the APSP data structure provided by Lemma 2.3.20 can be reduced to $O(1)$. The reason is that Bernstein and Roditty [24] provide, for example, a $(5+\epsilon', 0)$-approximate decremental APSP data structure for some constant ϵ'. The total update time of this data structure is

$$\tilde{O}\left(n^{2+1/3+O(1/\sqrt{\log n})}\right)$$

which is well within $O(n^{5/2})$. $\square$

The $(2+\epsilon, 0)$-approximate decremental APSP data structure now follows as a corollary. We simply need the following observation: if the distance between two nodes is 1, then we can answer queries for their distance exactly by checking whether they are connected by an edge.

Corollary 2.3.23 (Randomized $(2+\epsilon, 0)$-approximation with truly-subcubic total update time)**.** *For every $0 < \epsilon \leq 1$, there is a $(2+\epsilon, 0)$-approximate decremental APSP data structure with constant query time and an expected total update time of $O((n^{5/2}\log^3 n)/\epsilon)$ that is correct with high probability.*

Proof. By using the data structure of Theorem 2.3.22 we can, after the i-th edge deletion in a decremental graph $\mathcal{G} = (G_i)_{0 \le i \le k}$ and for all nodes x and y, query for a distance estimate $\delta_i(x, y)$ in constant time that satisfies:

$$d_{G_i}(x, y) \le \delta_i(x, y) \le (1+\epsilon) d_{G_i}(x, y) + 2$$

Note that if $d_{G_i}(x, y) \ge 2$, then

$$\delta_i(x, y) \le (1+\epsilon) d_{G_i}(x, y) + 2 \le (1+\epsilon) d_{G_i}(x, y) + d_{G_i}(x, y) = (2+\epsilon) d_{G_i}(x, y)\,.$$

If $d_{G_i}(x, y) < 2$, then we actually have $d_{G_i}(x, y) \le 1$ because G_i is an unweighted graph. A distance of 1 simply means that there is an edge connecting x and y in G_i. Since the adjacency matrix of $\mathcal{G}$ is maintained anyway, we can find out in constant time whether $d_{G_i}(x, y) = 1$. By setting, for all nodes x and y,

$$\delta'_i(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } (x, y) \in E(G_i) \\ \delta_i(x, y) & \text{otherwise} \end{cases}$$

we get $d_{G_i}(x, y) \le \delta'_i(x, y) \le (2+\epsilon) d_{G_i}(x, y)$. Clearly, this data structure can answer queries in constant time by returning the distance estimate $\delta'_i(x, y)$ and has the same total update time as the $(1+\epsilon, 2)$-approximate decremental APSP data structure, namely $O((n^{5/2} \log^3 n)/\epsilon)$. □

2.4 Deterministic Decremental $(1+\epsilon)$-Approximate APSP with $O(mn \log n)$ Total Update Time

In this section, we present a deterministic decremental $(1+\epsilon)$-approximate APSP algorithm with $O(mn \log n/\epsilon)$ total update time.

Theorem 2.4.1 (Main result of Section 2.4: Deterministic $O((mn \log n)/\epsilon)$ total update time)**.** *For every $0 < \epsilon \le 1$, there is a deterministic $(1+\epsilon, 0)$-approximate decremental APSP data structure with a total update time of $O((mn \log n)/\epsilon)$ and a query time of $O(\log \log n)$.*

Using known reductions, we show in Section 2.4.3 that this decremental algorithm implies a deterministic fully dynamic algorithm with an amortized running time of $\tilde{O}(mn/(\epsilon t))$ per update and a query time of $\tilde{O}(t)$ for every $t \le n$.

The main task in proving Theorem 2.4.1 is to design a deterministic version of the *center cover data structure* (see Section 2.2.3) with a total deterministic update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$ and constant query time. Once we have this data structure, Theorem 2.4.1 directly follows as a corollary from Theorem 2.2.14. Note that we cannot use the same idea is in [113] to reduce the query time from $O(\log \log n)$ to $O(1)$. This would require a *deterministic* (α, β)-approximate decremental APSP data structure for some constants α and β with *constant* query time and a total update

time of $O((mn \log n)/\epsilon)$. To the best of our knowledge such a data structure has not yet been developed.

Recall that R^c and R^d are the *coverage range* and *distance range* parameters where we want (a) every node (in a connected component of size at least R^c) to be within distance of at most R^c from some center, and we want (b) to maintain the distance from each center to every node within distance at most R^d. Roditty and Zwick [113], following an argument of Ullman and Yannakakis [127] (see also Lemma 1.3.2), observed that making each node a center independently at random with probability $(a \ln n)/R^c$, where a is a constant and $1 \leq R^c \leq n$, gives a set C of centers such that with probability at least $1 - n^{-(a-1)}$ the conditions of a center cover with parameter R^c are fulfilled by C in the initial graph and the expected size of C is $O((n \log n)/R^c)$. The randomized decremental APSP algorithm of [113] simply chooses a large enough value of a so that with high probability C fulfills the center cover properties with parameter R^c not only in the initial graph but continues to fulfill them in all the $O(n^2)$ graphs generated during $O(n^2)$ edge deletions. This is only possible because it is assumed that the "adversary" that generates the deletions is oblivious, i.e., does not know the location of the centers. The main challenge for the *deterministic* algorithm is to *dynamically adapt* the location and number of the centers so that (i) the center cover properties with size R^c continue to hold, while the graph is modified, and (ii) the total cost incurred is $O(mn)$. Once we have such a data structure, we can use the approach of Roditty and Zwick, as discussed in Section 2.2.3, to obtain an algorithm for maintaining decremental approximate shortest paths.

The *new feature* of our deterministic center cover data structure is that it sometimes *moves* centers to avoid opening too many centers (which are expensive to maintain). As we described in Section 2.1, the key technique behind the new data structure is what we call a *moving Even-Shiloach tree*. We note that the moving Even-Shiloach tree is actually a concept rather than a new implementation: we implement it in a straightforward way by building a new Even Shiloach tree every time we have to move it. However, analyzing the total update time needs new insights and a careful charging argument. To separate the analysis of the moving Even-Shiloach tree from the charging argument, we describe the data structure in two pieces:

(1) First, in Section 2.4.1, we give the *moving centers* data structure that can answer Distance and FindCenter operations, but needs to be told *where* to move a center, when a center has to be moved. This data structure is basically an implementation of *several* moving Even-Shiloach trees.[18]

(2) Then, in Section 2.4.2, we show how to determine when a center (with a moving Even-Shiloach tree rooted at it) has to be moved and, if so, where it has to move.

Combining these two pieces gives the center cover data structure.

[18]Later on we want to use the moving centers data structure, and not directly the moving Even-Shiloach trees, because we will need an additional operation which is not directly provided by the Even-Shiloach trees (in particular, the FindCenter operation defined in Section 2.4.1).

2.4.1 Deterministic Moving Centers Data Structure

In the following, we design a deterministic data structure, called *moving centers data structure*, and analyze its cost in terms of the number of centers opened ($\mathfrak{O}$) and the *moving distance* ($\mathfrak{D}$). When a center is created, it is given a unique identifier j. The data structure can handle the following operations.

Definition 2.4.2 (Moving centers data structure)**.** *A* moving centers data structure *with* cover range parameter R^{c} *and* distance range parameter R^{d} *for a decremental graph $\mathcal{G} = (G_i)_{0\le i\le k}$ maintains, for every $0 \le i \le k$, a set of* centers $C_i = \{1, 2, \dots, l\}$ *and a set of nodes $U_i = \{c_i^1, c_i^2, \dots, c_i^l\}$. For every center $j \in C$ and every $0 \le i \le k$, we call $c_i^j \in U_i$ the* location *of center j in G_i. After the i-th edge deletion (where $0 \le i \le k$), the data structure provides the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G_i.*
- Open(x)*: Open a new center at node x and return the ID of the opened center for later use.*
- Move(j, x)*: Move the center j from its current location c_i^j to node x.*
- Distance(j, x)*: Return the distance $d_{G_i}(c_i^j, x)$ between the location c_i^j of center j and the node x, provided that $d_{G_i}(c_i^j, x) \le R^d$. If $d_{G_i}(c_i^j, x) > R^d$, then return ∞.*
- FindCenter(x)*: Return a center j (with location c_i^j) such that $d_{G_i}(x, c_i^j) \le R^c$. If no such center exists, return $\perp$.*

The total update time *is the total time needed for performing the all delete, open, and move operations and the initialization. The* query time *is the worst-case time needed to answer a single distance or find center query.*

The moving centers data structure is a first step towards implementing the center cover data structure: It can answer all query operations that are posed to the center cover data structure, but, unlike the center cover data structure, it needs to be told where to place the centers and where to open new centers. This information is determined by the data structure in the next section.

In the rest of Section 2.4.1 we use the following notation: The decremental graph $\mathcal{G}$ undergoes a sequence of k edge deletions. By G_i we denote the graph after the i-th deletion (for $0 \le i \le k$). Each deletion in the graph is reported to the moving centers data structure by a delete operation. By C_i we denote the set of centers at the time of the i-th delete operation and by c_i^j we denote the location of center $j \in C_i$ at the time of the i-th delete operation.

Definition 2.4.3 (Moving distance ($\mathfrak{D}$))**.** *The* total moving distance*, denoted by $\mathfrak{D}$, is defined as $\mathfrak{D} = \sum_{0\le i<k} \sum_{j\in C_i} d_{G_i}(c_i^j, c_{i+1}^j)$.*

The main result of this section is that we can maintain a moving centers data structure in $O(m(\mathfrak{O}R^{\mathrm{d}} + \mathfrak{D}))$ time, as in Proposition 2.4.4 below. The data structure is actually very simple: we maintain an Even-Shiloach tree of depth at most R^{d} at every node for which we open a center and for every node to which we move a center. Note that our algorithm treats an Even-Shiloach tree at each such node as a new tree, regardless of whether we open a center or move a center there. While the algorithm can naively treat each Even-Shiloach tree as a new one, the analysis cannot: if we do so, we will get a total update time of $O((\mathfrak{O} + \mathfrak{M})mR^{\mathrm{d}})$, where $\mathfrak{M}$ is the total number of move-operations (since maintaining each Even-Shiloach tree takes $O(mR^{\mathrm{d}})$ total update time). Instead, we bound the cost incurred by the move-operation based on how far a center is moved, i.e., the moving distance $\mathfrak{D}$. This argument allows us to replace the unfavorable term $\mathfrak{M}mR^{\mathrm{d}}$ by $\mathfrak{D}m$. The deterministic center cover data structure of Section 2.4.2 will generate a sequence of center open and move requests so that $\mathfrak{D} = O(n)$. For simplifying the analysis, we state the following result under a technical assumption, that will always be fulfilled by the intended use of the moving centers data structure in Section 2.4.2.[19]

Proposition 2.4.4 (Main result of Section 2.4.1: deterministic moving centers data structure)**.** *Let R^c and R^d be parameters such that $R^c \leq R^d$. Under the assumption that between two consecutive delete operations there can be at most one open or delete operation for each center, there is a moving centers data structure with a total deterministic update time of $O((\mathfrak{O}R^d + \mathfrak{D})m)$, where $\mathfrak{O}$ is the number of open operations and $\mathfrak{D}$ is the total moving distance.*[20] *The data structure can answer each query in constant time.*

Proof. Our data structure maintains (1) An ES-tree of depth R^{d} rooted at every node that currently hosts a center; and (2) for every node a doubly-linked *center list* of centers by which it is covered. Recall that a node is covered by a center iff the node is contained in the ES-tree of depth R^{c} of the center. For every center j and node x we keep a pointer of the node representing x in the ES-tree of j to the list element representing j in the center list of x.

The data structure is updated as follows: Every time we open a center j at some node x we build an ES-tree of depth R^{d} rooted at x. Additionally we add j to the center list of all nodes covered by j and set the pointers from the ES-tree to the center lists.

When we move a center j from a node x to another node y we build an ES-tree of depth R^{d} rooted at y and stop maintaining the ES-tree rooted at x. Additionally we use the pointers from the ES-tree rooted at x into the center lists to remove j from all the center lists of the nodes in the ES-tree rooted x. Then we add j to the suitable center lists for all nodes in the ES-tree of y and add pointers into these lists from the ES-tree of y.

[19] Without this assumption the total update time will be $O((\mathfrak{O}R^{\mathrm{d}} + \mathfrak{M} + \mathfrak{D})m)$, where $\mathfrak{O}$ is the number of open operations, $\mathfrak{M}$ is the number of move operations, and $\mathfrak{D}$ is the total moving distance.

[20] Note that the total moving distance might be ∞ if, after some deletion i, a center j is moved from c_i^j to c_{i+1}^j such that there is not path between c_i^j and c_{i+1}^j in G_i. In this case we our analysis cannot bound the total update time of the moving centers data structure.

After deleting an edge we update all ES-trees of depth R^{d}. If a node x reaches a level larger than R^{d} in the ES-tree of j, it is removed from the ES-tree of j and we use its pointer in the ES-tree to remove j from x's center list. The total work of this operation is proportional to the amount of time spent updating all the ES-trees.

To answer a distance query for center j and node x we return the distance of x to the root of the ES-tree of j. To answer a find center query for node u we simply return the first element of the center list of node u. Both query operations take constant worst-case time.

We now bound the running time for maintaining the ES-trees of the centers. First we bound the initialization costs. For each open operation and each move operation of a center j we spend time $O(m)$ for (re-)initializing the ES-tree of center j. This leads to a total running time of $O(\mathfrak{O}m + \mathfrak{M}m)$ for all initializations where $\mathfrak{M}$ is the total number of move operations. Note that we can ignore every move operation that does not change the location of any center. Every other move operation increases the total moving distance by at least 1. Therefore we can charge the initialization cost of $O(m)$ for moving a center to the moving distance which means that the quantity $O(\mathfrak{O}m + \mathfrak{M}m)$ will be absorbed by $O((\mathfrak{O}R^{\mathrm{d}} + \mathfrak{D})m)$, the projected total update time.

We are left to bound the time spent for processing the deletions in the ES-trees of centers. For every center j, we denote by $T(i,j)$ the running time for processing the i-th edge deletion in the ES-tree of center j. Furthermore, we denote by o_j the index of the delete operation before which the center j has been opened, i.e., center j was opened before the o_j-th and after the (o_j-1)-th delete operation. Remember that the set of centers never shrinks, i.e., $C_i \subseteq C_k$ for every $0 \leq i \leq k$. We will show that $\sum_{0<i\leq k}\sum_{j\in C_i} T(i,j) = O((\mathfrak{O}R^{\mathrm{d}} + \mathfrak{D})m)$.

The basic idea is that the time spent up to deletion i for node x in the ES-tree of center j is $O(\deg_{G_0}(x)\cdot d_{G_i}(x,c_i^j))$. After a move operation the distance of x to the new root c_{i+1}^j is at most $d_{G_i}(c_{i+1}^j, c_i^j)$ smaller than the previous distance and, thus, at most $\sum_{0\leq i<k}\deg_{G_0}(x)\cdot d_{G_i}(c_i^j, c_{i+1}^j)$ additional time will be spent for updating x in the ES-tree of center j.

Consider the $(i+1)$-th edge deletion and let $j \in C_{i+1}$ be a center. By Corollary 2.2.11, the total time for processing this deletion in the ES-tree of center j is

$$T(i+1,j) = \sum_{x\in V}\deg_{G_0}(x)\cdot\left(\min\left(d_{G_{i+1}}(x,c_{i+1}^j), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x,c_{i+1}^j), R^{\mathrm{d}}\right)\right) \quad (2.8)$$

If j has already been opened before the i-th edge deletion, then, by the triangle inequality, we get $d_{G_i}(x,c_i^j) \leq d_{G_i}(x,c_{i+1}^j) + d_{G_i}(c_i^j, c_{i+1}^j)$ which is equivalent to $d_{G_i}(x,c_{i+1}^j) \geq d_{G_i}(x,c_i^j) - d_{G_i}(c_i^j, c_{i+1}^j)$. It follows that

$$\begin{aligned}\min\left(d_{G_i}(x,c_{i+1}^j), R^{\mathrm{d}}\right) &\geq \min\left(d_{G_i}(x,c_i^j) - d_{G_i}(c_{i+1}^j, c_i^j), R^{\mathrm{d}}\right)\\ &\geq \min\left(d_{G_i}(x,c_i^j), R^{\mathrm{d}}\right) - d_{G_i}(c_i^j, c_{i+1}^j)\,.\end{aligned}$$

Therefore we get

$$\begin{aligned}
T(i+1,j) &\leq \sum_{x\in V} \deg_{G_0}(x) \cdot \left(\min\left(d_{G_{i+1}}(x, c_{i+1}^j), R^{\mathrm{d}} \right) - \min\left(d_{G_i}(x, c_i^j), R^{\mathrm{d}} \right) \right. \\
&\quad \left. + \; d_{G_i}(c_i^j, c_{i+1}^j) \right) \\
&= \sum_{x\in V} \deg_{G_0}(x) \cdot \left(\min\left(d_{G_{i+1}}(x, c_{i+1}^j), R^{\mathrm{d}} \right) - \min\left(d_{G_i}(x, c_i^j), R^{\mathrm{d}} \right) \right) \\
&\quad + \sum_{x\in V} \deg_{G_0}(x) \cdot d_{G_i}(c_i^j, c_{i+1}^j) \\
&\leq \sum_{x\in V} \deg_{G_0}(x) \cdot \left(\min\left(d_{G_{i+1}}(x, c_{i+1}^j), R^{\mathrm{d}} \right) - \min\left(d_{G_i}(x, c_i^j), R^{\mathrm{d}} \right) \right) \\
&\quad + 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \,.
\end{aligned}$$

Summing up all $T(i,j)$ for every deletion $i > o_j$ gives a telescoping sum that results in the following term:

$$\begin{aligned}
\sum_{o_j < i \leq k} T(i,j) = \sum_{x\in V} \deg_{G_0}(x) \cdot \min\left(d_{G_k}(x, c_k^j), R^{\mathrm{d}} \right) \\
- \sum_{x\in V} \deg_{G_0}(x) \cdot \min\left(d_{G_{o_j}}(x, c_{o_j}), R^{\mathrm{d}} \right) + \sum_{o_j \leq i < k} 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \,.
\end{aligned}$$

Consider now a center j and the o_j-th edge deletion. By (2.8) we can bound the running time $T(o_j, j)$ as follows:

$$T(o_j, j) \leq \sum_{x\in V} \deg_{G_0}(x) \cdot \min\left(d_{G_{o_j}}(x, c_{o_j}^j), R^{\mathrm{d}} \right) \,.$$

Therefore the total time for maintaining the moving ES-tree of center j over all deletions is

$$\begin{aligned}
\sum_{o_j \leq i \leq k} T(i,j) &= T(o_j, j) + \sum_{o_j < i \leq k} T(i,j) \\
&\leq \sum_{x\in V} \deg_{G_0}(x) \cdot \min\left(d_{G_k}(x, c_k^j), R^{\mathrm{d}} \right) + \sum_{o_j \leq i < k} 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \\
&\leq \sum_{x\in V} \deg_{G_0}(x) \cdot R^{\mathrm{d}} + \sum_{o_j \leq i < k} 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \\
&\leq 2mR^{\mathrm{d}} + \sum_{o_j \leq i < k} 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \,.
\end{aligned}$$

By summing up this quantity over all centers, and switching the order of the double

sum, we arrive at the following total time:

$$\begin{aligned}\sum_{0<i\le k}\sum_{j\in C_i} T(i,j) &= \sum_{j\in C_k}\sum_{o_j\le i\le k} T(i,j)\\ &\le \sum_{j\in C_k} 2mR^{\mathrm{d}} + \sum_{j\in C_k}\sum_{o_j\le i<k} 2m\cdot d_{G_i}(c_i^j, c_{i+1}^j)\\ &= \sum_{j\in C_k} 2mR^{\mathrm{d}} + 2m\cdot\sum_{0\le i<k}\sum_{j\in C_i} d_{G_i}(c_i^j, c_{i+1}^j)\\ &= 2\mathfrak{O}mR^{\mathrm{d}} + 2m\mathfrak{D}\end{aligned}$$

Therefore the total update time for maintaining the moving centers data structure over all operations is $O((\mathfrak{O}R^{\mathrm{d}} + \mathfrak{D})m)$. □

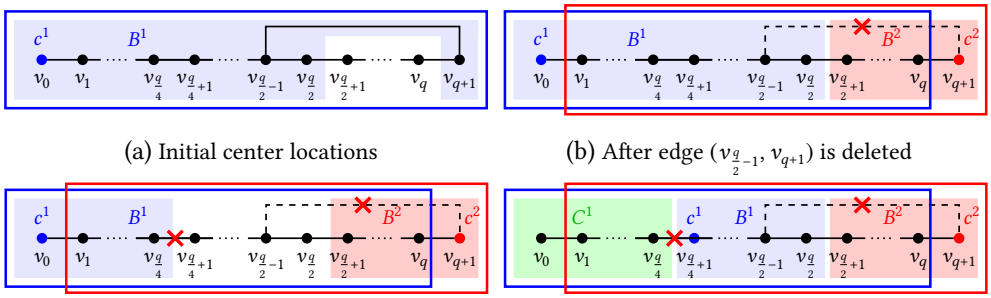
2.4.2 Deterministic Center Cover Data Structure

In this section, we present a deterministic algorithm for maintaining the center cover data structure CenterCover, as defined in Definition 2.2.13. That is, for parameters R^{c} and R^{d}, we show that we can maintain a set of centers with the following two properties. First, all nodes in a connected component of size at least R^{c} are *covered* by some center, i.e., each of them is at distance at most R^{c} to some center. Second, for every center, the distance to every node up to distance R^{d} is maintained. This section is devoted to proving the following.

Proposition 2.4.5 (Main result of Section 2.4.2)**.** *For every cover range parameter R^c and every distance range parameter R^d such that $R^c \le R^d$, there is a center cover data structure with a total* deterministic *update time of $O(mnR^d/R^c)$ and constant query time.*

High-Level Ideas

Our algorithm will internally use the moving centers data structure from Section 2.4.1 (called MovingCenter). It has to determine how to open and move centers in a way that ensures that at any time every node in a connected component of size at least R^{c} is covered by some center, i.e., its distance to the nearest center is at most R^{c}. At a high level, our algorithm is very simple (see Figure 2.3 for an example; note that $q = R^{\mathrm{c}}$): For each center j, it maintains two sets B^j and C^j, where B^j is always defined to be the set of nodes whose distance to center j is at most $R^{\mathrm{c}} - |C^j|$. Initially, the algorithm sets $C^j = \emptyset$ and chooses a set of centers such that all sets B^j are disjoint (see Figure 2.3a). The sets C^j will never decrease during the algorithm. After an edge deletion, if a node in a large connected component (size $\ge R^{\mathrm{c}}$) that is not covered by any center anymore (e.g., v_{q+1} in Figure 2.3b). In this case, the algorithm simply opens a new center at that node. However, before doing so it has to check whether $|B^j \cup C^j| < R^{\mathrm{c}}/2$ for some existing center j. (For example, after edge $(v_{q/4}, v_{q/4+1})$ is deleted as in Figure 2.3c, $|B^1 \cup C^1| = (q/4+1) < q/2 = R^{\mathrm{c}}/2$.) If this is the case for



(a) Initial center locations

(b) After edge $(v_{\frac{q}{2}-1}, v_{q+1})$ is deleted

(c) Deleting edge $(v_{\frac{q}{4}}, v_{\frac{q}{4}+1})$ without moving c^1

(d) After moving c^1

Figure 2.3: Example of our algorithm for maintaining the center cover data structure using the moving centers data structure, as in Proposition 2.4.5. We use $q = R^c$ and, for any j, we let c^j denote the location of center j. Boxes filled with colors show sets B^j and C^j. (a) shows a possible initial location of center c^1. This makes $B^1 = \{v_0, \dots, v_{q/2}\} \cup \{v_{q+1}\}$ and $C^1 = \emptyset$. All nodes are covered by center c^1. (b) shows what our algorithm does when edge $(v_{q/2-1}, v_{q+1})$ is deleted. In this case, v_{q+1} is not covered by c^1 anymore so we open a center c^2 at v_{q+1}. (c) shows what B^1 will look like after edge $(v_{q/4}, v_{q/4+1})$ is deleted, if we do not move center c^1. In particular, $|B^1 \cup C^1| < q/2$. (d) shows what our algorithm will do after edge $(v_{q/4}, v_{q/4+1})$ is deleted to maintain the largeness property (i.e., to make sure that $|B^1 \cup C^1| \geq q/2$): it moves nodes $v_0, \dots, v_{q/4}$ from B^1 to C^1 and moves the first center from v_0 to $v_{q/4+1}$.

center j, it will add all nodes of B^j to C^j and move the center j to the end-node of the deleted edge that is in a different connected component than the old location of j. As we will show, the nodes in B^j at the new location are *not* contained in $B^{j'}$ for any center $j' \neq j$, i.e., the invariant that all sets B^j are disjoint remains valid. For example, in Figure 2.3d, the algorithm puts nodes $v_0, \dots, v_{q/4}$ to C^1 and moves c^1 to node $v_{q/4+1}$, which is the end-node of the deleted edge $(v_{q/4}, v_{q/4+1})$ that is in a connected component different from center c^1.)

We now give the intuition behind this algorithm and its analysis before going into details. Recall from Proposition 2.4.4 that opening and maintaining a center together cost $O(mR^c)$ time in total and a move-operation incurs a total time of $O(m)$ per one unit moving distance. So, to get the desired $O(mnR^d/R^c)$ total time bound, we will make sure that our algorithm uses a limited number of open-operations and a limited moving distance; in particular, we will make sure that

$$\mathfrak{O} = O(n/R^c) \quad \text{and} \quad \mathfrak{D} = O(n).$$

To guarantee that we open at most $O(n/R^c)$ centers, we imagine that each node holds a coin at the beginning of the algorithm, which it can give to at most one center during the algorithm, and we require that each center must receive at least $R^c/2$ coins from some nodes in the end. Clearly, this will automatically ensure that at most $2n/R^c$ centers will be opened. Since the graph keeps changing, it is hard to say which node should give a coin to which center at the beginning. Instead, our algorithm will maintain two sets for each center j: the set B^j of *borrowed* nodes from

which center j has borrowed coins but might have to return the coins back, and the set C^j of *collected* nodes from which center j has collected coins that it will never return. After all edge deletions, j will hold the coins of all nodes in $B^j \cup C^j$. Our algorithm will maintain $B^j \cup C^j$ with two properties:

1. (Largeness) $|B^j \cup C^j| \geq R^c/2$ at any time (so that j gets enough coins in the end), and
2. (Disjointness) $B^j \cup C^j$ is disjoint from $B^{j'} \cup C^{j'}$ for all centers $j \neq j'$ (so that no node gives a coin to more than one center).

These two properties easily imply that every center will get at least $R^c/2$ coins in the end – center j simply collects coins from the nodes in $B^j \cup C^j$; consequently, they guarantee that $\mathfrak{D} = O(n/R^c)$ as desired. Note that B^j and C^j are only introduced for the analysis, our algorithm does *not* need to maintain them explicitly. If the location of center j is moved from x to y then we say for every node u on a shortest path between x to y that the center has been *moved through* u. To guarantee that the total moving distance is $O(n)$, we need one more property:

3. (Confinement) The location of center j is moved *only through nodes that are added to* C^j.

By the disjointness property, no two centers are moved along the same node if the confinement property is satisfied. So, the total moving distance will be $\mathfrak{D} = O(n)$ as desired.

It is left to check that the algorithm we have sketched earlier satisfies all three properties above. The largeness property can be guaranteed using the fact that after every edge deletion, the algorithm will move every center j such that $|B^j \cup C^j| < R^c/2$ to a new node; the only non-obvious property we have to prove is that B^j will be large enough after the move, and the key to this proof is the fact that the connected component containing the new location of center j has size at least $R^c/2 - |C^j|$. For the disjointness property, we will show two further properties.

(P1) (Initial-disjointness) When we open a center j, B^j is disjoint from $B^{j'} \cup C^{j'}$ for all other centers j'.

(P2) (Shrinking) We never add any node to $B^j \cup C^j$. (For example, $B^1 \cup C^1$ in Figure 2.3a is a subset of $B^1 \cup C^1$ in Figure 2.3d.)

These two properties are sufficient to guarantee the disjointness property because if two sets $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint at the beginning (by Property (P1)), they will remain disjoint if we never add a node to them (by Property (P2)). The shrinking property (Property (P2)) can be checked simply by observing the behavior of the algorithm (see Lemma 2.4.12 for detail). To show the initial-disjointness property (Property (P1)), we use the fact that j is of distance at least R^c from other centers when j is opened which implies that $B^j \cap B^{j'} = \emptyset$. Additionally, we will prove that $C^{j'}$ contains only nodes in connected components of size less than R^c whereas any

new center j is opened in a connected component of size at least R^c. This implies that $B^j \cap C^{j'} = \emptyset$ when j is opened.

Finally, for the confinement property, just observe that before the algorithm moves a center j, it puts all nodes in the connected component containing the center j to C^j and moves j to a node outside of this connected component. For example, in Figure 2.3d the algorithm puts nodes $v_1, \ldots, v_{q/4}$ to C^1 before moving the first center through $v_1, \ldots, v_{q/4}$ to $v_{q/4+1}$.

Algorithm Description

Our algorithm is outlined in Algorithm 2.3. For each center j, the algorithm maintains its location c^j, which could change over time since centers can be moved. Besides, it also maintains the set C^j and the number r^j, which are set to $\emptyset$ and $R^c/2$, respectively, when center j is opened. The intended value of r^j is $r^j = R^c/2 - |C^j|$ and the algorithm always updates r^j in a way that this is ensured. The algorithm also uses the moving centers data structure (denoted by MovingCenter and explained in Section 2.4.1) to maintain the distance between each center j to other nodes in the graph, up to distance R^d. This helps us to implement CenterCover.FindCenter and CenterCover.Distance queries in a straightforward way: the algorithm just invokes the same queries from the moving centers data structure.

Initially, on G_0 (i.e., before the graph changes), our algorithm initializes the moving centers data structure by opening centers in a greedy manner: as long as there is a node x that is not covered by any center, it opens a center at x. This process will also be used every time an edge is deleted, to make sure that every node remains covered by a center. Procedure CenterCover.GreedyOpen proceeds as follows. For every node x, it checks whether x is *not covered*; this is the case if CenterCover.FindCenter(x) returns $\perp$ and the size of the connected component containing x is at least R^c (we refer to Lemma 2.4.21 for how to compute the size of this component). If x is not covered, the algorithm opens a center at x, stores the index j of this new center, and initializes the values of C^j, r^j and c^j, as in Line 6. This completes the GreedyOpen procedure.

The main work of Algorithm 2.3 lies in the Delete operation, since it has to make sure that all nodes are still covered by some centers after the deletion. Procedure CenterCover.Delete proceeds as follows. Let us assume that the $(i+1)$-th edge (u, v) is deleted from G_i and let G_{i+1} denote the resulting graph. First, the procedure checks whether there is any center j that is in a large component in G_i and in a small connected component in G_{i+1}; i.e., the size of the connected component of c^j is at least r^j in G_i and less than r^j in G_{i+1} (see Line 16 of Algorithm 2.3). Next, if such a center j in a small connected component exists (in fact, we will show that there exists at most one such j – see Lemma 2.4.15), we will *move* j to a different component and update the values of C^j, r^j and c^j. It is crucial in our analysis that j must be moved carefully. In particular, we will move j to either u or v, depending on which node is in a *different* component from c^j, the current location of j. (Note that one of u and v will be in the same connected component as j and the other will be in a different

Algorithm 2.3: CenterCover (Deterministic Center Cover Data Structure)

```
// Given a decremental graph 𝒢 = (G_i)_{0≤i≤k} and integers R^c and R^d,
   this data structure maintains a set of centers such that
   every node (that is in a connected component of size at
   least R^c) has distance at most R^c to at least one center
   and we can query the distance between a center and a node
   if their distance is at most R^d (otherwise, we will get ∞
   in return). It allows four operations: Initialize, Delete,
   FindCenter and Distance, as defined in Definition 2.2.13.
1  Procedure CenterCover.GreedyOpen()
2      Let G_i denote the current graph
3      foreach node x do
           // The if-statement checks if x is not covered by a
              center and the size of the connected component
              containing it is larger than R^c. See Lemma 2.4.21 for
              the implementation detail.
4          if CenterCover.FindCenter(x) = ⊥ and |Comp_{G_i}(x)| ≥ R^c then
               // Tell moving centers data structure to open new
                  center at x. Let j be the index of this center.
5              j ← MovingCenter.Open(x)
6              Set C^j ← ∅, r^j ← R^c/2, and c^j ← x

   // Parameters: Initial version G_0 of decremental graph 𝒢,
      integers R^c and R^d.
7  Procedure CenterCover.Initialize(G_0, R^c, R^d)
       // Initialize the moving centers data structure.
8      MovingCenter.Initialize(G_0, R^c, R^d)
9      CenterCover.GreedyOpen()

10 Procedure CenterCover.FindCenter(v)          // Parameter: Node v
11     return MovingCenter.FindCenter(v)

   // Parameters: Center index j and node v
12 Procedure CenterCover.Distance(j, v)
13     return MovingCenter.Distance(j, v)

   // Parameter: (i+1)-th deleted edge (u, v)
14 Procedure CenterCover.Delete(u, v)
15     Let G_i denote the graph before deleting (u, v) and let G_{i+1} denote the graph
       afterwards.
       // Find a center j for which the connected component
          containing it becomes smaller than r^j. See Lemma 2.4.22
          for how to find such a center j. (Actually, there will
          be at most one such center, see Lemma 2.4.15.)
16     Find a center j such that |Comp_{G_{i+1}}(c^j)| < r^j.
17     if such a center j exists then
           // Move j to either u or v depending on who is in a
              different connected component than c^j.
18         if u and c^j are not connected in G_{i+1} then y ← u else y ← v
19         Set C^j ← C^j ∪ Comp_{G_{i+1}}(c^j), r^j ← r^j − |Comp_{G_{i+1}}(c^j)|, and c^j ← y
20         MovingCenter.Move(j, y)
       // Report edge deletion to moving centers data structure.
21     MovingCenter.Delete(u, v)
22     CenterCover.GreedyOpen()
```

component.) We use a variable $y \in \{u, v\}$ to refer to the new location to which we move center j (see Line 18). We then update the values of C^j, r^j and c^j. In particular, we put *all* nodes in the connected component that previously contained center j (before we move it to y) into C^j and update r^j to $R^c/2 - |C^j|$ and c^j to y. Then we report the move of center j to y to the moving centers data structure. Afterwards, we report the deletion of the edge (u, v) to the moving centers data structure so that it updates the distances between centers and nodes to the new distances in G_{i+1}. Finally, we execute the CenterCover.GreedyOpen procedure to make sure that every node remains covered: if there is a node x that is not covered, we open a center at x. This completes the deletion operation.

Analysis

The correctness of Algorithm 2.3 is immediate. As the procedure GreedyOpen is called after every edge deletion, every node in a connected component of size at least R^c will always be covered. In the following we analyze the running time of Algorithm 2.3.

Our main task is to bound the running time of the moving centers data structure internally used by the algorithm. In particular we want to use the running time bound stated in Proposition 2.4.4 which requires to bound the number $\mathfrak{O}$ of open-operations performed by the algorithm and the total moving distance $\mathfrak{D}$. As outlined in Section 2.4.2 we assign to each center j the set $B^j \cup C^j$. The set C^j contains all nodes of connected components in which the center j once was located, as shown in Algorithm 2.3. The set B^j is the set of all nodes that are at distance at most r^j from the center j in the current graph. We first show that the sets $B^j \cup C^j$ fulfill two properties: disjointness and largeness. Disjointness says that for all centers $j \neq j'$ the sets $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint. Largeness says that the set $B^j \cup C^j$ has size at least $R^c/2$ for each center j. Using these two properties, we will prove that there are at most $\mathfrak{O} = O(n/R^c)$ open-operations and that the total moving distance is $\mathfrak{D} = O(n)$. These bounds will then allow us to obtain a total update time of $O(mnR^d/R^c)$ for the moving centers data structure used by Algorithm 2.3. Afterwards we will show that all other operations of the algorithm can also be carried out within this total update time. To make our arguments precise we will use the following notation.

Definition 2.4.6. *Let $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph for which Algorithm 2.3 maintains a center cover data structure. The graph undergoes a sequence of k deletions and by G_i we denote the graph after the i-th deletion. We use the following notation:*

- *For all nodes x and y we denote by $d_i(x, y) = d_{G_i}(x, y)$ the distance between x and y in the graph G_i.*
- *For every node x, we denote by $\mathrm{Comp}_i(x) = \mathrm{Comp}_{G_i}(x)$ the nodes in the connected component of x in the graph G_i.*

- *For every center j, we denote by c_i^j, r_i^j, and C_i^j the values of c^j, r^j, and C^j after the algorithm has processed the i-th deletion, respectively (equivalently: the values before the $(i+1)$-th deletion).*
- *For every center j, we define the set B_i^j by $B_i^j = \{x \in V \mid d_i(c_i^j, x) \leq r_i^j\}$, i.e., B_i^j is the set of all nodes that are within distance r_i^j to the location c_i^j of center j in the graph G_i.*

Preliminary observations We first state some simple observations that will be helpful later on.

Observation 2.4.7. *Let x be a node, let $i \leq k$, and let B' be the set $B' = \{y \in V \mid d_i(x,y) \leq r\}$ for some integer r. If $|\operatorname{Comp}_i(x)| < r$, then $\operatorname{Comp}_i(x) = B'$. Furthermore, $|\operatorname{Comp}_i(x)| < r$ if and only if $|B'| < r$.*

Proof. Clearly $B' \subseteq \operatorname{Comp}_i(x)$ and thus $|B'| \leq |\operatorname{Comp}_i(x)|$. Therefore, if $|\operatorname{Comp}_i(x)| < r$ also $|B'| < r$. Now assume that $|B'| < r$. We first show that $\operatorname{Comp}_i(x) \subseteq B'$. Let y be a node in $\operatorname{Comp}_i(x)$ and assume by contradiction that $d_i(x,y) > r$. Since $y \in \operatorname{Comp}_i(x)$, x and y are connected and therefore the shortest path from x to y has to contain some node z such that $d_i(x,z) = r$. The shortest path π from x to z contains $d_i(x,z) = r$ edges and $r+1$ nodes. For every node z' on π we have $d_i(x,z') \leq r$ and thus $\pi \subseteq B'$. Since $|\pi| = r+1$ we get $|B'| \geq r+1$ which contradicts our assumption. Therefore $d_i(x,y) \leq r$ which means that $\operatorname{Comp}_i(x) \subseteq B'$. Now $|\operatorname{Comp}_i(x)| \leq |B'| < r$ as desired. □

Observation 2.4.8. *For every center j and every $i \leq k$, $r_i^j = R^c/2 - |C_i^j|$.*

Proof. When the center j is opened the algorithm sets $r_i^j = R^c/2$ and $C_i^j = \emptyset$. Therefore $r_i^j = R^c/2 - |C_i^j|$ trivially holds. Afterwards the algorithm only modifies r^j and C^j when a center is moved. Since r^j is increased by exactly the amount by which $|C^j|$ is decreased, the equation remains true. □

Observation 2.4.9. *For every center j and every $i \leq k$, $|\operatorname{Comp}_i(x)| < R^c$ for every node $x \in C_i^j$.*

Proof. For every node x that is put into C_i^j after the i-th edge deletion we have $|\operatorname{Comp}_i(x)| < r_i^j$. Since the size of the connected component of x never increases in a decremental graph and $r_i^j \leq R^c$ for all $i \leq k$ by Observation 2.4.8, the claim is true. □

Observation 2.4.10. *For every center j and every $i \leq k$, the sets B_i^j and C_i^j are disjoint.*

Proof. The set C_i^j only contains nodes in connected components from which the center j has been moved away, i.e., that do not contain c_i^j. No center will ever be moved back into such a connected component. Since $B_i^j \subseteq \operatorname{Comp}_i(c_i^j)$, we conclude that B_i^j and C_i^j are disjoint. □

Disjointness property We now want to prove the disjointness property. We will proceed as follows: first we show that, for every center j that is opened, the set $B^j \cup C^j$ is disjoint from the set $B^{j'} \cup C^{j'}$ of every other existing center j'. Afterwards we show that the algorithm never adds any nodes to $B^j \cup C^j$. These two facts will imply that all the sets $B^j \cup C^j$ are disjoint.

Lemma 2.4.11 (Initial disjointness)**.** *When the algorithm opens a center j after the i-th edge deletion, the set $B_i^j \cup C_i^j$ is disjoint from the set $B_i^{j'} \cup C_i^{j'}$ for every other center $j \neq j'$.*

Proof. Let j be the center that is opened and let $j' \neq j$ be an existing center. The algorithm sets $C_i^j = \emptyset$ and therefore we only have to argue that B_i^j and $B_i^{j'} \cup C_i^{j'}$ are disjoint. Note that c_i^j is in a connected component of size at least R^c because otherwise the algorithm would not have opened a center at c_i^j. Observe that the set B_i^j is contained in the connected component of c_i^j. By Observation 2.4.9 all nodes of $C_i^{j'}$ are in a connected component of size less than R^c and therefore $B_i^j \cap C_i^{j'} = \emptyset$. We now argue that $B_i^j \cap B_i^{j'} = \emptyset$. Suppose that there is some node x contained in both B_i^j and $B_i^{j'}$. By the definition of B_i^j and $B_i^{j'}$ we get $d_i(c_i^j, x) \leq r_i^j = R^c/2 - |C_i^j| \leq R^c/2$ as well as $d_i(c_i^{j'}, x) \leq R^c/2$. By the triangle inequality we get

$$d_i(c_i^j, c_i^{j'}) \leq d_i(c_i^j, x) + d_i(x, c_i^{j'}) \leq R^c/2 + R^c/2 = R^c .$$

But then c_i^j is covered by $c_i^{j'}$. This means that the algorithm would not have opened a new center at c_i^j which contradicts our assumption. □

Lemma 2.4.12 (Shrinking property)**.** *For every center j and every $i < k$, we have $B_i^j \cup C_i^j \subseteq B_{i+1}^j \cup C_{i+1}^j$.*

Proof. Let (u, v) be the $(i+1)$-th deleted edge. We only have to argue that the claim holds for centers that the algorithm has already opened before this deletion. If the algorithm does not move j, then the values of C^j, r^j, and c^j are not changed at all and thus $C_{i+1}^j = C_i^j$, $r_{i+1}^j = r_i^j$, and $c_{i+1}^j = c_i^j$. Furthermore, since distances never decrease in a decremental graph we also have $B_{i+1}^j \subseteq B_i^j$ and the claim follows.

Now consider the case that the algorithm moves the center j from $x = c_i^j$ to c_{i+1}^j, where either $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Assume without loss of generality that $c_{i+1}^j = v$. To simplify notation, let A denote the set $A = \mathrm{Comp}_{i+1}(x)$. The fact that the algorithm moves the center implies that $|A| < r_i^j$. Note that the algorithm sets $C_{i+1}^j = C_i^j \cup A$ and $r_{i+1}^j = r_i^j - |A|$.

The observation needed for proving the shrinking property is $B_{i+1}^j \cup A \subseteq B_i^j$. From this observation we get $B_{i+1}^j \cup C_{i+1}^j = B_{i+1}^j \cup C_i^j \cup A \subseteq B_i^j \cup C_i^j$ as desired. We first prove $A \subseteq B_i^j$ and then we prove $B_{i+1}^j \subseteq B_i^j$. Let B' be the set $B' = \{z \in V \mid d_{i+1}(x, z) \leq r_i^j\}$. Since $|A| < r_i^j$ we get $A \subseteq B'$ by Observation 2.4.7 and since $d_i(x, z) \leq d_{i+1}(x, z)$ for every node z we have $B' \subseteq B_i^j$. Now $A \subseteq B'$ and $B' \subseteq B_i^j$, and we may conclude that $A \subseteq B_i^j$.

Finally, we prove that $B^j_{i+1} \subseteq B^j_i$. Since we move the center j from x to v it must be the case, by the way the algorithm works, that $|A| = |\operatorname{Comp}_{i+1}(x)| < |\operatorname{Comp}_i(x)|$ and that v is not connected to x in G_{i+1}. This can only happen if v is connected to x in G_i. Let z be a node in B^j_{i+1} which means that $d_{i+1}(v,z) \leq r^j_{i+1}$. Consider a shortest path π from x to v in G_i consisting of $d_i(x,v)$ many edges. Every edge on π except for the last one (which is (u,v)) is also contained in G_{i+1} and therefore all nodes on π except for v are contained in A. Therefore we get $|A| \geq |\pi \setminus \{v\}| = d_i(x,v)$. We now get $z \in B^j_i$ by observing that $d_i(x,z) \leq r^j_i$, which can be seen from the following chain of inequalities:

$$d_i(x,z) \leq d_i(x,v) + d_i(v,z) \leq d_i(x,v) + d_{i+1}(v,z) \leq |A| + r^j_{i+1} = r^j_i\,. \qquad \square$$

Lemma 2.4.13 (Disjointness)**.** *Algorithm 2.3 maintains the following invariant: for all centers $j \neq j'$ and every $i \leq k$, $B^j_i \cup C^j_i$ is disjoint from $B^{j'}_i \cup C^{j'}_i$.*

Proof. By Lemma 2.4.11 the invariant holds after the initialization. Now consider the $(i+1)$-th edge deletion. Let $j \neq j'$ be two different existing centers. By the induction hypothesis $B^j_i \cup C^j_i$ and $B^{j'}_i \cup C^{j'}_i$ are disjoint. Since $B^j_{i+1} \cup C^j_{i+1} \subseteq B^j_i \cup C^j_i$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1} \subseteq B^{j'}_i \cup C^{j'}_i$ by Lemma 2.4.12, also $B^j_{i+1} \cup C^j_{i+1}$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1}$ are disjoint. Now let j be an existing center and let j' be a center that is opened in the procedure CenterCover.GreedyOpen (called at the end of the procedure CenterCover.Delete). By Lemma 2.4.11 we also have that $B^j_{i+1} \cup C^j_{i+1}$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1}$ are disjoint. This shows that, for *all* centers j and j' such that $j \neq j'$, $B^j_{i+1} \cup C^j_{i+1}$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1}$ are disjoint. $\square$

Largeness property We now want to prove the largeness property which states that for every center j the size of the set $B^j \cup C^j$ is always at least $R^c/2$. The largeness property will follow from the invariant $|B^j| \geq r^j$. Before we can prove this invariant we have to argue that our algorithm really moves every center j that fulfills the "moving condition" $|\operatorname{Comp}_i(c^j)| \geq r^j$ and $|\operatorname{Comp}_{i+1}(c^j)| < r^j$. Remember that the algorithm only moves *one* such center after each deletion. We show that there actually is at most one center fulfilling the moving condition and therefore it is not necessary that the algorithm also moves any other center.

Observation 2.4.14. *Let (u,v) be the $(i+1)$-th deleted edge. If $|\operatorname{Comp}_i(c^j_i)| \geq r^j_i$ and $|\operatorname{Comp}_{i+1}(c^j_i)| < r^j_i$, then $u \in B^j_i$ and $v \in B^j_i$.*

Proof. Suppose that $d_i(u,c^j_i) > r^j_i$. Let π a shortest path from c^j_i to u in G_i consisting of $d_i(u,c^j_i) > r^j_i$ many edges and thus at least $r^j_i + 1$ nodes. The edge (u,v) can only appear as the last edge on the shortest path π. Therefore, after deleting it, there are still r^j_i nodes connected to c^j_i which contradicts the assumption that $\operatorname{Comp}_{i+1}(c^j_i) < r^j_i$. Thus, $d_i(u,c^j_i) \leq r^j_i$ which means that $u \in B^j_i$. Since the edge (u,v) is undirected the same argument works for v. $\square$

Lemma 2.4.15 (Uniqueness of center to move)**.** *Let (u, v) be the $(i+1)$-th deleted edge. If $|B_i^j| \geq r_i^j$ for every center j, then there is at most one center j such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$ and, in G_{i+1}, either u is connected to c_i^j (and v is disconnected from c_i^j) or v is connected to c_i^j (and u is disconnected from c_i^j).*

Proof. Let j be a center such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. As $|B_i^j| \geq r_i^j$, we also have $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$ by Observation 2.4.7. The size of the connected component of c_i^j can only decrease if the deletion of (u, v) disconnects at least one node from $\operatorname{Comp}_i(c^j)$. For this to happen, u and v must be connected to c_i^j in G_i and furthermore one of these nodes (either u or v) must be disconnected from c_i^j in G_{i+1} while the other node stays connected to c_i^j.

Now suppose that there are two centers $j \neq j'$ such that $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$ and $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$, and $|\operatorname{Comp}_i(c_i^{j'})| \geq r_i^{j'}$ and $|\operatorname{Comp}_{i+1}(c_i^{j'})| < r_i^{j'}$. By Observation 2.4.14, we get $u \in B_i^j$ and $u \in B_i^{j'}$ which contradicts the disjointness property of Lemma 2.4.13. We conclude that there cannot be two such centers $j \neq j'$. □

Lemma 2.4.16. *For every center j and every $i \leq k$, Algorithm 2.3 maintains the invariant $|B_i^j| \geq r_i^j$.*

Proof. We first argue that the invariant holds for every center j that we open at some node x in the greedy open procedure after the i-th deletion. The algorithm only opens the center if x is in a connected component of size at least R^c. Since $r_i^j = R^c/2 - |C_i^j| \leq R^c$ (Observation 2.4.8) we have $|\operatorname{Comp}_i(x)| \geq r_i^j$. Therefore we get $|B_i^j| \geq r_i^j$ by Observation 2.4.7.

We now show that the invariant is maintained for all centers that have already been opened before we delete the $(i+1)$-th edge (u, v). Consider first the case that $|\operatorname{Comp}_{i+1}(c_i^j)| \geq r_i^j$. In that case the center j will not be moved and we have $C_{i+1}^j = C_i^j$, $B_{i+1}^j = B_i^j$, and $r_{i+1}^j = r_i^j$. Since $|\operatorname{Comp}_{i+1}(c_{i+1}^j)| \geq r_{i+1}^j$ we get $|B_{i+1}^j| \geq r_{i+1}^j$ as desired by Observation 2.4.7.

Now consider the case that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. Since the invariant holds for i, Lemma 2.4.15 applies and thus we can be sure that the algorithm will move center j from node x to node y (where either $y = u$ or $y = v$). Remember that we have $c_i^j = x$, $c_{i+1}^j = y$, and $r_{i+1}^j = r_i^j - |\operatorname{Comp}_{i+1}(x)|$ in that case. Since x and y were connected in G_i but are not connected anymore in G_{i+1} we get $\operatorname{Comp}_{i+1}(y) = \operatorname{Comp}_i(x) \setminus \operatorname{Comp}_{i+1}(x)$. Due to $\operatorname{Comp}_{i+1}(x) \subseteq \operatorname{Comp}_i(x)$ it follows that

$$|\operatorname{Comp}_{i+1}(y)| = |\operatorname{Comp}_i(x)| - |\operatorname{Comp}_{i+1}(x)| \geq r_i^j - |\operatorname{Comp}_{i+1}(x)| = r_{i+1}^j .$$

By Observation 2.4.7, the fact that $|\operatorname{Comp}_{i+1}(y)| \geq r_{i+1}^j$ implies that $|B_{i+1}^j| \geq r_{i+1}^j$ as desired. □

Lemma 2.4.17 (Largeness)**.** *For every center j and every $i \leq k$, Algorithm 2.3 maintains the invariant $|B_i^j \cup C_i^j| \geq R^c/2$.*

Proof. By Observation 2.4.10, B_i^j and C_i^j are disjoint and by Observation 2.4.8 we have $r_i^j = R^c/2 - |C_i^j|$. By Lemma 2.4.16 we have $|B_i^j| \geq r_i^j$. Therefore we get the desired bound as follows:

$$|B_i^j \cup C_i^j| = |B_i^j| + |C_i^j| \geq r_i^j + |C_i^j| = R^c/2 - |C_i^j| + |C_i^j| = R^c/2$$

where the inequality above follows from Lemma 2.4.16. □

Bounding the number of open operations Now that we have established the disjointness and the largeness property for the sets $B^j \cup C^j$ of every center j, we can bound the number of open-operations by $\mathfrak{O} = O(n/R^c)$. This will be useful for our goal of limiting the total update time of the moving centers data structure to $O(mnR^d/R^c)$.

Lemma 2.4.18 (Number of open operations)**.** *Over all edge deletions, Algorithm 2.3 performs $O(n/R^c)$ open-operations in its internal moving centers data structure.*

Proof. Let C_k denote the set of centers after all k deletions. Note that moving a center does not change the number of centers. Therefore, the size of C_k is equal to the total number of centers opened. Due to the disjointness property (Lemma 2.4.13) the sets $B_k^j \cup C_k^j$ after all k edge deletions are disjoint for all centers j. When we sum up over all these sets we do not count any node twice. Therefore we get

$$\sum_{j \in C_k} |B_k^j \cup C_k^j| = \left| \bigcup_{j \in C_k} (B_k^j \cup C_k^j) \right| \leq n$$

By the largeness property (Lemma 2.4.17) every set $B_k^j \cup C_k^j$ has size at least $R^c/2$, i.e., $|B_k^j \cup C_k^j| \geq R^c/2$. We now combine both inequalities and get

$$n \geq \sum_{j \in C} |B_k^j \cup C_k^j| \geq \sum_{j \in C} R^c/2 = |C| R^c/2$$

which gives $|C| \leq 2n/R^c$ as desired. □

Bounding the total moving distance Finally, we prove that the total moving distance of the moving centers data structure used by our algorithm is $O(n)$. For this proof we will use a property of the algorithm that we call *confinement*: every center j will be moved only through nodes that are added to C^j.

Lemma 2.4.19 (Confinement)**.** *For every move of center j from c_i^j to c_{i+1}^j after the $(i+1)$-th edge deletion, let π_i^j be the set of nodes on a shortest path from c_i^j to c_{i+1}^j in G_i. Then, for every center j and every $0 \leq i < k$, $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$.*

Proof. Let (u, v) be the $(i+1)$-th deleted edge. Consider the situation that the algorithm moves some center j from c_i^j to c_{i+1}^j. By the rules of the algorithm for

moving centers we either have $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Due to Observation 2.4.14 we have $c_{i+1}^j \in B_i^j$ which means that $d_i(c_i^j, c_{i+1}^j) \leq r_i^j$.

Now let π_i^j be a shortest path from c_i^j to c_{i+1}^j in G_i. All nodes in π_i^j, except for c_{i+1}^j, are connected to c_i^j in G_{i+1} since the edge (u, v) only appears as the last edge on the shortest path due to $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Therefore we have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq \mathrm{Comp}_{i+1}(c_i^j)$. Since $C_{i+1}^j = C_i^j \cup \mathrm{Comp}_{i+1}(c_i^j)$, we get $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j$. We also have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq B_i^j$ because $d_i(c_i^j, c_{i+1}^j) \leq r_i^j$. Since B_i^j and C_i^j are disjoint (Observation 2.4.10), also $\pi_i^j \setminus \{c_{i+1}^j\}$ and C_i^j are disjoint. It therefore follows that $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$. □

Lemma 2.4.20 (Total moving distance)**.** *The total moving distance of the moving centers data structure used by Algorithm 2.3 is* $\mathfrak{D} = O(n)$.

Proof. We let C_k denote the set of centers after the algorithm has processed all deletions. Furthermore, we denote by o_j the index of the edge deletion before which the center j has been opened, i.e., center j was opened before the o_j-th and after the $(o_j - 1)$-th deletion.

Consider the situation that the algorithm moves a center j from c_i^j to c_{i+1}^j after the $(i+1)$-th edge deletion and let π_i^j be a shortest path from c_i^j to c_{i+1}^j in G_i as in Lemma 2.4.19. The shortest path π_i^j consists of $d_i(c_i^j, c_{i+1}^j)$ many edges and $d_i(c_i^j, c_{i+1}^j) + 1$ many nodes. Therefore we get $d_i(c_i^j, c_{i+1}^j) = |\pi_i^j \setminus \{c_{i+1}^j\}|$. By Lemma 2.4.19 we have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$ for every center j and every $0 \leq i < k$. The value of the set C^j after all edge deletions is given by C_k^j for every center j. By the disjointness property (Lemma 2.4.13) we have $\sum_{j \in C} |C_k^j| = \left|\bigcup_{j \in C} C_k^j\right|$. We now obtain the bound $\mathfrak{D} \leq n$ as follows.

$$\begin{aligned}\mathfrak{D} &= \sum_{j \in C_k} \sum_{o_j \leq i < k} d_i(c_i^j, c_{i+1}^j) = \sum_{j \in C_k} \sum_{o_j \leq i < k} |\pi_i^j \setminus \{c_{i+1}^j\}| \\ &\leq \sum_{j \in C_k} \sum_{o_j \leq i < k} |C_{i+1}^j \setminus C_i^j| = \sum_{j \in C_k} |C_k^j| = \left|\bigcup_{j \in C} C_k^j\right| \leq n\,. \end{aligned}$$ □

Implementation details Before we finish this section we clarify two implementation details of Algorithm 2.3 and argue that they can be carried out within the total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$.

There are two places in the algorithm where we have to compute the sizes of connected components. First, in the procedure GreedyOpen, we have to check for every node that is not covered by any center whether it is in a connected of size at least R^{c}. Second, in the procedure Delete, we have to check whether the size of the connected component of some center j drops below r^j. So far we have not explained explicitly how to carry out these steps. If we could obtain the size of the connected component deterministically in linear time, the running time analysis we have given so far would suffice. Remember that the moving centers data structure internally maintains an ES-tree for every center. Thus, it would seem intuitive to use the ES-trees for counting the number of nodes in the current component of each center.

However, we do not report edge deletions to the moving centers data structure immediately. Therefore it is not clear how to use these ES-trees to determine the size of the connected components of a centers.

Instead, we do the following. In parallel to our own algorithm we use the deterministic (fully) dynamic connectivity data structure of Henzinger and King [62].[21] This data structure can answer queries of the form "are the nodes x and y connected?" in constant time. Its amortized time per deletion is $O(n^{1/3}\log n)$. Thus, its total update time over all deletions is $O(mn^{1/3}\log n)$. Trivially, this data structure allows us to compute the size of the connected component of a node x in time $O(n)$: we simply iterate over all nodes and count how many of them are connected to x. We now explain how to perform the two tasks listed above using the dynamic connectivity data structure.

Lemma 2.4.21 (Detail of Line 4 in Algorithm 2.3)**.** *Given a dynamic connectivity data structure with constant query time, performing the check in the if-condition of Line 4 takes time $O((n+\mathfrak{O})n)$ over all deletions, where $\mathfrak{O}$ is the total number of open-operations.*

Proof. Given a node x we have to check whether CenterCover.FindCenter(x) $= \perp$ and $|\operatorname{Comp}_{G_i}(x)| \geq R^c$. We first check whether x is covered by any center by querying the moving centers data structure (if x is covered, the procedure returns a center covering x, otherwise it returns $\perp$.) This check takes constant time. If a node x is not covered we additionally have to check whether $|\operatorname{Comp}_i(x)| < R^c$. Note that if $|\operatorname{Comp}_i(x)| < R^c$ for some node x we do not have to consider this node anymore after future deletions because connected components never increase their size in a decremental graph. Therefore we may spend time $O(n)$ for every node x to determine $|\operatorname{Comp}_i(x)|$. If $|\operatorname{Comp}_i(x)| < R^c$, then we charge this time to the node and will never process the node again in the future and if $|\operatorname{Comp}_i(x)| \geq R^c$, then we charge this time to the open-operation. Therefore the total running time over all deletions for performing this check in the if-condition is $O((n+\mathfrak{O})n)$, where $\mathfrak{O}$ is the total number of open-operations. □

Lemma 2.4.22 (Detail of Line 16 of Algorithm 2.3)**.** *Given a dynamic connectivity data structure with constant query time, we can, after the $(i+1)$-th deletion, find a center j such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$ if it exists in time $O(n)$.*

Proof. Let (u,v) be the $(i+1)$-th deleted edge. For any center j such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$, we have $u \in B_i^j$ by Observation 2.4.14. Moreover, by the disjointness property (Lemma 2.4.13), there can only be at most one center j such that $u \in B_i^j$. The algorithm for finding this center now is simple: We find a center j such that $u \in B_i^j$, which is unique if it exists; then, we compute the size of the connected component containing c_i^j using the dynamic connectivity data structure [62]. In particular, we

[21] This is the fastest known deterministic dynamic connectivity data structure with *constant* query time.

iterate over all centers in time $O(n)$ to find a candidate center j such that $d_i(u, c_i^j) \leq r^j$ (i.e., $u \in B_i^j$) (as argued above at most one such center exists). We can determine the distance $d_i(u, c_i^j)$ in constant time by querying the moving centers data structure. For the candidate center j we now have to check whether $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. We determine the size of $\operatorname{Comp}_{i+1}(c_i^j)$ in time $O(n)$ by using the dynamic connectivity data structure with constant query time. Thus, the running time for this algorithm is $O(n)$ per deletion. □

Total update time Now we state the total update time of Algorithm 2.3. The bounds on the number of centers opened and the total moving distance of the centers allow us to bound the running time of the moving centers data structure used by the algorithm.

Theorem 2.4.23. *The deterministic center cover data structure of Algorithm 2.3 has constant query time and a total update time of $O(mnR^d/R^c)$.*

Proof. By Proposition 2.4.4 the moving centers data structure internally used by Algorithm 2.3 has constant query time and a total deterministic update time of $O(\mathfrak{O}mR^{\mathrm{d}} + \mathfrak{D}m)$ where $\mathfrak{O}$ is the total number of open operations and $\mathfrak{D}$ is the total moving distance. Algorithm 2.3 delegates all queries to the moving centers data structure and therefore also has constant query time. By Lemma 2.4.18 the number of open operations is $O(n/R^{\mathrm{c}})$ and by Lemma 2.4.20 the total moving distance is $O(n)$. Therefore the total update time of the moving centers data structure is

$$O(\mathfrak{O}mR^{\mathrm{d}} + \mathfrak{D}m) = O(mnR^{\mathrm{d}}/R^{\mathrm{c}} + mn) = O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$$

because $R^{\mathrm{c}} \leq R^{\mathrm{d}}$. As argued in Lemma 2.4.21 and Lemma 2.4.22, all other operations of the algorithm can be implemented within a total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. Therefore the claimed running time follows. □

2.4.3 Deterministic Fully Dynamic Algorithm

There is a well-known reduction by Henzinger and King [61] for converting a decremental algorithm into a fully dynamic algorithm. A similar approach has been used by Roditty and Zwick [113], using the decremental algorithm we derandomized above as the starting point. In the following we sketch the deterministic fully dynamic algorithm implied by Theorem 2.4.1. The fully dynamic algorithm allows two update operations: deleting an arbitrary set of edges and inserting a set of edges touching a node v, called the center of the insertion.

Theorem 2.4.24. *For every $0 < \epsilon \leq 1$ and every $t \leq \sqrt{n}$ there is a deterministic fully dynamic $(1+\epsilon, 0)$-approximate APSP data structure with amortized update time $O(mn/(\epsilon t))$ and query time $\tilde{O}(t)$.*

Proof. The algorithm works in phases. After each t update operations we start a new phase. At the beginning of each phase we re-initialize the decremental algorithm of

Theorem 2.4.1. We report to this algorithm all future deletions of the phase, but no insertions. For all nodes u and v let $\delta_1(u, v)$ denote the $(1 + \epsilon)$-approximate distance estimate obtained by the decremental algorithm. Additionally, after every update in the graph, we do the following: Let I denote the set of centers of all insertions that so far happened in the current phase. For every $v \in I$, we compute the shortest paths from v to all nodes in the current graph, i.e., where all insertions and deletions are considered. We use Dijkstra's algorithm for this task and denote by $\delta_2(u, v)$ the distance from u to v computed in this way.

To answer a query for the approximate distance between nodes u and v we compute and return the following value:

$$\delta(u, v) = \min(\delta_1(u, v), \min_{x \in I}(\delta_2(u, x) + \delta_2(x, v))) .$$

Let $d(u, v)$ denote the distance from u to v in the current graph. If there is a shortest path from u to v that that does not use any edge inserted in the current phase, then the decremental algorithm provides a $(1 + \epsilon)$-approximation of the distance between u and v, i.e., $\delta_1(u, v) \leq (1 + \epsilon)d(u, v)$. Otherwise the shortest paths from u to v contains an inserted node $x \in I$. In that case we have $d(u, v) = \delta_2(u, x) + \delta_2(x, v)$ and thus $d(u, v) = \min_{x \in I}(\delta_2(u, x) + \delta_2(x, v)))$. This means that $\delta(u, v) \leq (1 + \epsilon)d(u, v)$. As both $\delta_1(u, v)$ and $\min_{x \in I}(\delta_2(u, x) + \delta_2(x, v)))$ never underestimate the true distance we also have $\delta(u, v) \geq d(u, v)$.

As the query time of the decremental algorithm is $O(\log \log n)$, the query time of the fully dynamic algorithm is $O(t + \log \log n) = \tilde{O}(t)$. The decremental approximate all-pairs shortest paths data structure has a total update time of $\tilde{O}(mn/\epsilon)$. Amortized over the whole phase, we have to pay $\tilde{O}(mn/(\epsilon t))$ per update for this data structure. Computing the shortest paths from the inserted nodes takes time $\tilde{O}(tm)$ per update. This gives an amortized update time of $\tilde{O}(mn/(\epsilon t) + tm)$. If $t \leq \sqrt{n}$, the term tm is dominated by the term mn/t and thus the amortized update time is $\tilde{O}(mn/(\epsilon t))$. □

We remark that the fully dynamic result of Roditty and Zwick [113] is still a bit stronger. Their trade-off is basically the same, but it holds for a larger range of t, namely $t \leq m^{1/2-\delta}$ for every fixed $\delta > 0$. The reason is that they use randomization not only for the decremental algorithm but also for some other part of the fully dynamic algorithm.

2.5 Conclusion

We obtained two new algorithms for solving the decremental approximate APSP algorithm in unweighted undirected graphs. Our first algorithm is $(1 + \epsilon, 2)$-approximate and has a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time. The main idea behind this algorithm is to run an algorithm of Roditty and Zwick [113] on a sparse dynamic emulator. In particular, we modify the central shortest paths tree data structure of Even and Shiloach [49, 79] to deal with edge insertions in a monotone manner. Our approach is conceptually different from the approach of Bernstein and

Roditty [24] who also maintain an ES-tree in a sparse dynamic emulator. The sparsification techniques used here and at other places only work for undirected graphs. Using a new sampling technique, we recently obtained a $(1+\epsilon, 0)$-approximation for decremental SSSP in *directed* graphs with constant query time and a total update time of $o(mn)$ (see Chapter 4).

Our second algorithm provides a $(1+\epsilon, 0)$-approximation and has a *deterministic* total update time of $O(mn \log n/\epsilon)$ and constant query time. We obtain it by derandomizing the algorithm of [113] using a new amortization argument based on the idea of relocating Even-Shiloach trees.

Our results directly motivate the following directions for further research. It would be interesting to extend our derandomization technique to other randomized algorithms. In particular, we ask whether it is possible to derandomize the *exact* decremental APSP algorithm of Baswana, Hariharan and Sen [16] (total update time $\tilde{O}(n^3)$).

Another interesting direction is to check whether our monotone ES-tree approach also works for other dynamic emulators, in particular for weighted graphs. One of the tools that we used was a dynamic $(1+\epsilon, 2)$-emulator for unweighted undirected graphs. Is it also possible to obtain *purely additive* dynamic emulators or spanners with small additive error?

Maybe the most important open problem in this field is a faster APSP algorithm for the fully dynamic setting. The fully dynamic algorithm of Demetrescu and Italiano [36] provides exact distances and takes time $\tilde{O}(n^2)$ per update, which is essentially optimal. Is it possible to get a faster fully dynamic algorithm that still provides a good approximation, for example a $(1+\epsilon)$-approximation?

CHAPTER 3

Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time

In the decremental single-source shortest paths (SSSP) problem we want to maintain the distances between a given source node s and every other node in an n-node m-edge graph G undergoing edge deletions. While its static counterpart can be easily solved in near-linear time, this decremental problem is much more challenging even in the *undirected unweighted* case. In this case, the classic $O(mn)$ total update time of Even and Shiloach [49] has been the fastest known algorithm for three decades. At the cost of a $(1+\epsilon)$-approximation factor, the running time was recently improved to $O(n^{2+o(1)})$ by Bernstein and Roditty [24]. In this chapter, we bring the running time down to near-linear: We give a $(1+\epsilon)$-approximation algorithm with $O(m^{1+o(1)})$ total update time, thus obtaining *near-linear time*. Moreover, we obtain $O(m^{1+o(1)} \log W)$ time for the weighted case, where the edge weights are integers from 1 to W. The only prior work on weighted graphs in $o(mn \log W)$ time is the $O(mn^{9/10+o(1)})$-time algorithm presented in Chapter 4 which works for the general weighted directed case.

In contrast to the previous results which rely on maintaining a sparse emulator, our algorithm relies on maintaining a so-called *sparse* (h, ϵ)*-hop set* introduced by Cohen [32] in the PRAM literature. An (h, ϵ)-hop set of a graph $G = (V, E)$ is a set F of weighted edges such that the distance between any pair of nodes in G can be $(1+\epsilon)$-approximated by their h-hop distance (given by a path containing at most h edges) on $G' = (V, E \cup F)$. Our algorithm can maintain an $(n^{o(1)}, \epsilon)$-hop set of near-linear size in near-linear time under edge deletions. It is the first of its kind to the best of our knowledge. To maintain approximate distances using this hop set,

we extend the monotone Even-Shiloach tree of Chapter 2 and combine it with the bounded-hop SSSP technique of Bernstein [22, 23] and Mądry [95]. These two new tools might be of independent interest.

3.1 Introduction

Dynamic graph algorithms refer to data structures on graphs that support update and query operations. They are classified according to the type of update operations they allow: *decremental* algorithms allow only edge deletions, *incremental* algorithms allow only edge insertions, and *fully dynamic* algorithms allow both insertions and deletions. In this chapter, we consider decremental algorithms for the *single-source shortest paths (SSSP)* problem on *undirected* graphs. The *unweighted* case of this problem allows the following operations.

- DELETE(u, v): delete the edge (u, v) from the graph, and
- DISTANCE(x): return the distance $d_G(s, x)$ between node s and node x in the current graph G.

The *weighted* case allows an additional operation INCREASE(u, v, Δ) which increases the weight of the edge (u, v) by Δ. We allow positive integer edge weights in the range from 1 to W, for some parameter W. For any $\alpha \geq 1$, we say that an algorithm is an *α-approximation* algorithm if, for any distance query DISTANCE(x), it returns a distance estimate $\delta(s, x)$ such that $d_G(s, x) \leq \delta(s, x) \leq \alpha d_G(s, x)$. There are two time complexity measures associated with this problem: *query time* denoting the time needed to answer *each* distance query, and *total update time* denoting the time needed to process *all* edge deletions. The running time will be in terms of n, the number of nodes in the graph, and m, the number of edges *before* the first deletion. For the weighted case, we additionally have W, the maximum edge weight. We use $\tilde{O}$-notation to hide $O(\text{polylog } n)$ terms. In this chapter, we focus on algorithms with small ($O(1)$ or $O(\text{polylog } n)$) query time, and the main goal is to minimize the total update time, which will simply be referred to as *time* when the context is clear.

Related Work The static version of SSSP can be easily solved in $\tilde{O}(m)$ time using, e.g., Dijkstra's algorithm. Moreover, due to the deep result of Thorup [123], it can even be solved in linear ($O(m)$) time in undirected graphs with positive integer edge weights. This implies that in our setting we can naively solve decremental SSSP in $O(m^2)$ time by running the static algorithm after every deletion. The first non-trivial decremental algorithm is due to Even and Shiloach [49] from 1981 and takes $O(mn)$ time in unweighted undirected graphs. This algorithm will be referred to as *ES-tree* throughout this chapter. It has many applications such as for decremental strongly-connected components [109] and multicommodity flow problems [95]; yet, the ES-tree has resisted many attempts of improving it for decades. Roditty and Zwick [115] explained this phenomenon by providing evidence that the ES-tree is optimal for maintaining exact distances even on *unweighted undirected* graphs,

unless there is a major breakthrough for Boolean matrix multiplication and many other long-standing problems [128]. After the preliminary version [64] of this work appeared, Henzinger et al. [70] showed that $O(mn)$ is essentially the best possible total update time for maintaining exact distances under the assumption that there is no "truly subcubic" algorithm for a problem called online Boolean matrix-vector multiplication. It is thus natural to shift the focus to *approximation algorithms*.

The first improvement for unweighted undirected graphs was due to Bernstein and Roditty [24] who presented a $(1+\epsilon)$-approximation algorithm an expected total update time of $O(n^{2+O(1/\sqrt{\log n})})$.[1] This time bound is only slightly larger than quadratic and beats the $O(mn)$ time of the ES-tree unless the input graph is very sparse. For the more general cases, Henzinger and King [61] observed that the ES-tree can be easily adapted to directed graphs. King [79] later extended the ES-tree to an $O(mnW)$-time algorithm for weighted directed graphs. A scaling techniques used in recent algorithms of Bernstein [22, 23] and Mądry [95], as well as earlier papers on approximate shortest paths [31, 130], gives a $(1+\epsilon)$-approximate $\tilde{O}(mn \log W)$-time algorithm for weighted directed graphs. Very recently, we obtained a $(1+\epsilon)$-approximation algorithm with total update time $O(mn^{9/10+o(1)})$ for decremental approximate SSSP in weighted directed graphs if $W \leq 2^{\log^c n}$ for some constant c (see Chapter 3). This gives the first $o(mn)$ time algorithm for the directed case, as well as other important problems such as single-source reachability and strongly-connected components [90, 109, 114]. Also very recently, Abboud and Vassilevska Williams [1] showed that "deamortizing" our $O(mn^{9/10+o(1)})$-time algorithms might not be possible: a combinatorial algorithm with *worst case* update time and query time of $O(n^{2-\delta})$ (for some $\delta > 0$) per deletion implies a faster combinatorial algorithm for Boolean matrix multiplication and, for the more general problem of maintaining the number of reachable nodes from a source under deletions (which our algorithms can do) a worst case update and query time of $O(m^{1-\delta})$ (for some $\delta > 0$) will falsify the strong exponential time hypothesis.

Our Results Given the significance of the decremental SSSP problem, it is important to understand its time complexity. In this chapter, we obtain a near-linear time algorithm for decremental $(1+\epsilon)$-approximate SSSP in weighted undirected graphs. It has a total update time of $O(m^{1+O(\sqrt{\log\log n/\log n})} \log W)$ and maintains an estimate of the distance between the source node and every other node, guaranteeing constant worst-case query time. The algorithm is randomized and assumes an oblivious adversary who fixes the sequence of updates in advance; it is correct with high probability and the bound on its total update time holds in expectation. In the unweighted case, our algorithm significantly improves our previous algorithm in [63] as discussed above. There was no previous algorithm designed specifically for weighted undirected graphs, and the previous best running time for this case comes from our $O(mn^{9/10+o(1)})$ time for weighted directed graphs (Chapter 3).

[1] To enhance readability we assume that ϵ is a constant when citing related work, thus omitting the dependence on ϵ in the running times.

As a consequence of our techniques we also obtain an algorithm for the all-pairs shortest paths (APSP) problem. For every integer $k \geq 2$ and every $0 < \epsilon \leq 1$, we obtain a decremental $((2+\epsilon)^k - 1)$-approximate APSP algorithm with query time $O(k^k)$ and total update time $O(m^{1+1/k+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log^2 W)$. We remark that for $k = 2$ and $1/\epsilon = O(\text{polylog } n)$ our result gives a $(3+\epsilon)$-approximation with constant query time and total update time $O(m^{1+1/2+o(1)} \log W)$. For very sparse graphs with $m = \Theta(n)$, this is almost optimal in the sense that it almost matches the static running time [125] of $O(m\sqrt{n})$, providing stretch of $3+\epsilon$ instead of 3 as in the static setting. Our result on approximate APSP has to be compared with the following prior work. For weighted directed graphs Bernstein [23] gave a decremental $(1+\epsilon)$-approximate APSP algorithm with constant query time and total update time $\tilde{O}(mn \log W)$. For unweighted undirected graphs there are two previous results that improve upon this update time at the cost of larger approximation error. First, for any fixed integer $k \geq 2$, Bernstein and Roditty gave a decremental $(2k-1+\epsilon)$-approximate APSP algorithm with constant query time and total update time $\tilde{O}(mn^{1/k})$. Second, for any integer $k \geq 2$, Abraham, Chechik, and Talwar [3] gave a decremental $2^{O(\rho k)}$-approximate APSP algorithm for unweighted undirected graphs with query time $O(k\rho)$ and total update time $\tilde{O}(mn^{1/k})$, where $\rho = (1 + \lceil (\log n^{1-1/k})/\log(m/n^{1-1/k}) \rceil)$.

3.2 Preliminaries

In this chapter we want to maintain approximate shortest paths in an undirected graph $G = (V, E)$ with positive integer edge weights in the range from 1 to W, for some parameter W. The graph undergoes a sequence of *updates*, which might be edge deletions or edge weight increases. This is called the *decremental setting*. We denote by V the set of nodes of G and by E the set of edges of G before the first edge deletion. We set $n = |V|$ and $m = |E|$.

For every weighted undirected graph $G = (V, E)$, we denote the weight of an edge (u, v) in G by $w_G(u, v)$. The *distance* $d_G(u, v)$ between a node u and a node v in G is the weight of the shortest path, i.e., the minimum-weight path, between u and v in G. If there is no path between u and v in G, we set $d_G(x, y) = \infty$. For every set of nodes $U \subseteq V$ we denote by $E[U]$ the set of edges incident to the nodes of U, i.e., $E[U] = \{(u, v) \in E \mid u \in U\}$.[2] Furthermore, for every set of nodes $U \subseteq V$, we denote by $G|U$ the subgraph of G induced by the nodes in U, i.e., $G|U$ contains all edges (u, v) such that (u, v) is contained in E and u and v are both contained in U, or short: $G|U = (U, E \cap U^2)$. Similarly, for every set of edges $F \subseteq V^2$ and every set of nodes $U \subseteq V$ we denote by $F|U$ the subset of F induced by U.

We say that a distance estimate $\delta(u, v)$ is an (α, β)-approximation of the true distance $d_G(u, v)$ if $d_G(u, v) \leq \delta(u, v) \leq \alpha d_G(u, v) + \beta$, i.e., $\delta(u, v)$ never underestimates the true distance and overestimates it with a multiplicative error of at most α and an additive error of at most β. If there is no additive error, we simply say α-approximation instead of $(\alpha, 0)$-approximation.

[2] Since G is undirected, this definition is equivalent to $E[U] = \{(u, v) \in E \mid u \in U \text{ or } v \in U\}$.

In our algorithms we will use graphs that do not only undergo edge deletions and edge weight increases, but also edge insertions. For such a graph H, we denote by $\mathscr{E}(H)$ the number of edges ever contained in H, i.e., the number of edges contained in H before any deletion or insertion plus the number of inserted edges. We denote by $\mathscr{W}(H)$ the number of edge weight increases in H. Similarly, for a set of edges F, we denote by $\mathscr{E}(F)$ the number of edges ever contained in F and by $\mathscr{W}(F)$ the number of edge weight increases in F.

Recent approaches for solving approximate decremental SSSP and APSP use special graphs called *emulators*. An (α, β)-emulator H of a graph G is a graph containing the nodes of G such that $d_G(u,v) \leq d_H(u,v) \leq \alpha d_G(u,v) + \beta$ for all nodes u and v.[3] Maintaining exact distances on H provides an (α, β)-approximation of distances in G. As good emulators are sparser than the original graph this is usually more efficient than maintaining exact distances on G. However, the edges of H also have to be maintained while G undergoes updates. For unweighted, undirected graphs undergoing edge deletions, the emulator of Thorup and Zwick (based on the second spanner construction in [126]), which provides a relatively good approximation, can be maintained quite efficiently [24]. However the definition of this emulator requires the occasional insertion of edges into the emulator. Thus, it is not possible to run a purely decremental algorithm on top of it.

There have been approaches to design algorithms that mimic the behavior of the classic ES-tree when run on an emulator that undergoes insertions. The first approach by Bernstein and Roditty [24] extends the ES-tree to a fully dynamic algorithm and analyzes the additional work incurred by the insertions. The second approach was introduced in Chapter 2 and is called *monotone ES-tree*. It basically ignores insertions of edges into H and never decreases the distance estimate it maintains. However, this algorithm does not provide an (α, β)-approximation on *any* (α, β)-approximate emulator as it needs to exploit the structure of the emulator. In Chapter 2 we gave an analysis of the monotone ES-tree when run on a specific $(1+\epsilon, 2)$-emulator and in this chapter we use a different analysis for our new algorithms. If we want to use the monotone ES-tree to maintain (α, β)-approximate distances up to depth D we will set the maximum level in the monotone ES-tree to $L = \alpha D + \beta$. The running time of the monotone ES-tree as analyzed in Chapter 2 is as follows.

Lemma 3.2.1. *For every $L \geq 1$, the total update time of a monotone ES-tree up to maximum level L on a graph H undergoing edge deletions, edge insertions, and edge weight increases is $O(\mathscr{E}(H) \cdot L + \mathscr{W}(H))$.*

3.3 Technical Overview

In the following we explain the main ideas of this chapter, which lead to an algorithm for maintaining a hop set of a graph undergoing edge deletions.

[3] For the related notion of a spanner we additionally have to require that H is a subgraph of G.

General Idea With the well-known algorithm of Even and Shiloach we can maintain a shortest paths tree from a source node up to a given depth D under edge deletions in time $O(mD)$. In unweighted graphs, all simple paths have length at most n and therefore we can set $D = n$ to maintain a full shortest paths tree. In weighted graphs with positive integer edge weights from 1 to W, all simple paths have length at most nW and therefore we can set $D = nW$ to maintain a full shortest paths tree. Using an established scaling technique [22, 23, 31, 95, 98, 130], one can use this algorithm to maintain $(1+\epsilon)$-approximate single-source shortest paths up to h hops in time $O(mh\log(nW)/\epsilon)$. With this algorithm we can set $h = n$ to maintain a full-length approximate shortest paths tree, even in weighted graphs. This algorithm would be very efficient if the graph had a small hop diameter, i.e., if for any pair of nodes there is a shortest path with a small number of edges. Our idea is to artificially construct such a graph.

To this end we will use a so-called *hop set*. An (h,ϵ)- *hop set* F of a graph $G = (V,E)$ is a set of weighted edges $F \subseteq V^2$ such that in the graph $H = (V, E \cup F)$ there exists, for every pair of nodes u and v, a path from u to v of weight at most $(1+\epsilon)d_G(u,v)$ and with at most h edges. If we run the approximate SSSP algorithm on H, we obtain a running time of $O((m+|F|)h\log(nW)/\epsilon)$. In our algorithm we will obtain an $(O(n^{o(1)}),\epsilon)$-hop set of size $O(m^{1+o(1)})$ and thus the running time will be $O(m^{1+o(1)}\log(nW)/\epsilon)$. It is however not enough to simply construct the hop set at the beginning. We also need a dynamic algorithm for maintaining the hop set under edge deletions in G. We will present an algorithm that performs this task also in almost linear time over all deletions.

Roughly speaking, we achieve the following. Given a graph $G = (V,E)$ undergoing edge deletions, we can maintain a restricted hop set F such that, for all pairs of nodes u and v if the shortest path from u to v in G has $h \geq n^{1/q}$ hops, in the shortcut graph $H = (V, E \cup F)$ there is a path from u to v of weight at most $(1+\epsilon)d_G(u,v)$ and with at most $\lceil h/n^{1/q}\rceil \log n$ hops. Our high-level idea for maintaining an (unrestricted) $(n^{o(1)},\epsilon)$ hop set is the following hierarchical approach. We start with $H_0 = G$ to maintain a hop set F_1 of G, which reduces the number of hops by a factor of $\log n/n^{1/q}$ at the cost of a multiplicative error of $1+\epsilon$. Given F_1, we use the shortcut graph $H_1 = (V, E \cup F_1)$ to maintain a hop set F_2 of G that reduces the number of hops by another factor of $\log n/n^{1/q}$ introducing another error of $1+\epsilon$. By repeating this process q times we arrive at a hop set that guarantees, for all pairs of nodes u and v, a path of weight at most $(1+\epsilon)^q d_G(u,v)$ and with at most $(\log n)^q n^{1/q}$ hops. Figure 3.1 visualizes this hierarchical approach.

The notion of hop set was first introduced by Cohen [32] in the PRAM literature and is conceptually related to the notion of emulator. It is also related to the notion of *shortest-paths diameter* used in distributed computing (e.g., [78, 98]). To the best of our knowledge, the only place that this hop set concept was used before in the dynamic algorithm literature (without the name being mentioned) is Bernstein's fully dynamic $(2+\epsilon)$-approximation algorithm for all-pairs shortest paths [22]. There, an $(n^{o(1)},\epsilon)$-hop set is essentially recomputed *from scratch* after every edge update, and a shortest-paths data structure is maintained on top of this hop set.

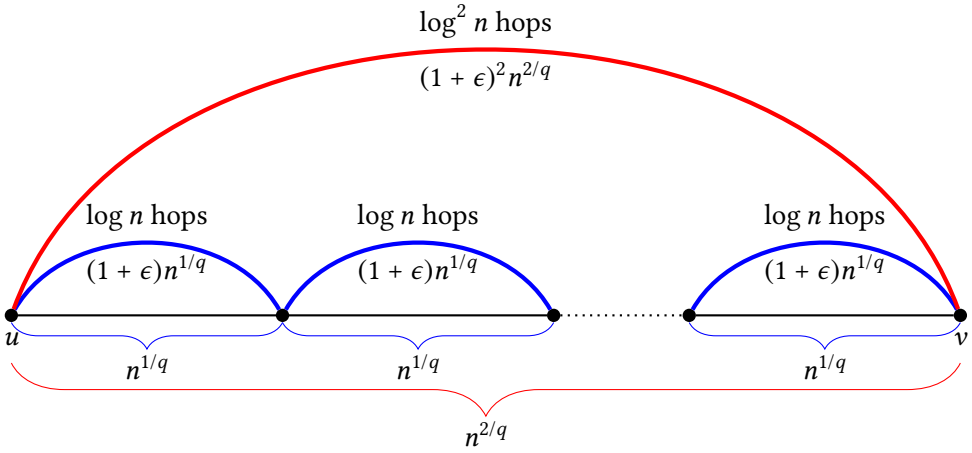


Figure 3.1: Illustration of the hierarchical approach for maintaining the hop set reduction. Here $q = \Theta(\sqrt{\log n})$ and u and v are nodes that are at distance $n^{2/q}$ from each other. In the first layer we find a hop set that shortcuts all subpaths of weight $n^{1/q}$ by paths of weight at most $(1+\epsilon)n^{1/q}$ and with at most $\log n$ hops. In the second layer, we use the shortcuts of the first layer to find a hop set that shortcuts the path from u to v of weight $n^{2/q}$ by a path of weight at most $(1+\epsilon)^2 n^{2/q}$ and with at most $\log^2 n$ hops.

Static Hop Set We first assume that $G = (V, E)$ is an unweighted undirected graph and for simplicity we also assume that ϵ is a constant. We explain how to obtain a hop set of G using a randomized construction of Thorup and Zwick [126] based on the notion of balls of nodes. We describe this construction and the hop-set analysis in the following.

Let $2 \leq p \leq \log n$ be a parameter and consider a sequence of sets of nodes $A_0, A_1, \ldots, A_p$ obtained as follows. We set $A_0 = V$ and $A_p = \emptyset$ and for $1 \leq i \leq p-1$ we obtain the set A_i by picking each node of V independently with probability $1/n^{i/p}$. The expected size of A_i is $n^{1-i/p}$. For every node u we define the priority of u as the maximum i such that $u \in A_i$. For a node u of priority i we define

$$B(u) = \{v \in V \mid d_G(u, v) < d_G(u, A_{i+1})\}$$

where $d_G(u, A_{i+1}) = \min_{v \in A_{i+1}} d_G(u, v)$. Note that $d_G(u, A_p) = \infty$ and thus if $u \in A_{p-1}$, then $B(u) = V$. For each node u of priority i the size of $B(u)$ is $n^{(i+1)/p}$ in expectation by the following argument: Order the nodes in non-decreasing distance from u. Each of these node belongs to A_{i+1} with probability $1/n^{(i+1)/p}$ and therefore, in expectation, we need to see $n^{(i+1)/p}$ nodes until one of them is contained in A_{i+1}. It follows that the expected size of all balls of priority i is at most $n^{1+1/p}$ (the expected size of A_i times the expected size of $B(u)$ for each node u of priority i) and the expected size of all balls, i.e., $\sum_{u \in V} |B(u)|$, is at most $pn^{1+1/p}$.

Let F be the set of edges $F = \{(u, v) \in V^2 \mid v \in B(u)\}$ and give each edge $(u, v) \in F$ the weight $w_F(u, v) = d_G(u, v)$. By the argument above, the expected size of F is at most $pn^{1+1/p}$. An argument of Thorup and Zwick [126] shows that the weighted graph $H = (V, F)$ has the following property for every pair of nodes u and v and any $0 < \epsilon \leq 1$ such that $1/\epsilon$ is integer:[4]

$$d_G(u, v) \leq d_H(u, v) \leq (1 + \epsilon)d_G(u, v) + 2\left(2 + \frac{2}{\epsilon}\right)^{p-2}$$

In the literature, such a graph H is known as an *emulator* of G with multiplicative error $(1 + \epsilon)$ and additive error $2(2 + 2/\epsilon)^{p-2}$.[5] Roughly speaking, the strategy in their proof is as follows. Let u' be the node following u on the shortest path from u to v in G. If the edge (u, u') is also contained in H, then we can shorten the distance to v by 1 without introducing any approximation error (recall that we assume that G is unweighted). Otherwise, one can show that there is a path with at most p edges in H from u to a node v' closer to v than u such that the ratio between the weight of this path and the weight of the shortest path from u to v' is at most $(1 + \epsilon)$, and, if $v' = v$, then the weight of this path is at most $2(2 + 2/\epsilon)^{p-2}$. The proof needs the following property of the balls: for every node u of priority i and every node v either $v \in B(u)$ or there is some node v' of priority $j > i$ such that $u \in B(v')$. We illustrate the proof strategy in Figure 3.2.

Observe that the same strategy can be used to show the following: Given any integer $\Delta \leq n$, let u' be the node that is at distance Δ from u on the shortest path from u to v in G. If the edge (u, u') is contained in H, then we can shorten the distance to v by Δ without introducing any approximation error. Otherwise, one can show that there is a path with at most p edges in H from u to a node v' closer to v than u such that the ratio between the weight of this path and the weight of the shortest path from u to v' is at most $(1 + \epsilon)$, and, if $v' = v$, then the weight of this path is at most $2(2 + 2/\epsilon)^{p-2} \cdot \Delta$. Every time we repeat this argument the distance to v is shortened by at least Δ. Therefore there is a path from u to v in H with at most $p\lceil d_G(u, v)/\Delta \rceil$ edges that has weight at most $(1 + \epsilon)d_G(u, v) + 2(2 + 2/\epsilon)^{p-2} \cdot \Delta$. One can show that this statement would also be true if we had removed all edges from F of weight more than $(1 + 2/\epsilon)(2 + 2/\epsilon)^{p-2}$, which is the maximum weight of the edge to v' in the proof strategy above. We will need this fact in the dynamic algorithm as it allows us to limit the depth of the balls.

By a suitable choice of $p = \Theta(\sqrt{\log n})$ (as a function of n and ϵ) we can guarantee that $2(2 + 2/\epsilon)^{p-2} \leq \epsilon n^{1/p}$ and $n^{1/p} = n^{o(1)}$. Now define $q = p$ and $\Delta_k = n^{k/q}$ for each

[4]The requirement that $1/\epsilon$ must be integer is not needed in the paper of Thorup and Zwick; we have added it here to simplify the exposition.

[5]In their paper, Thorup and Zwick [126] actually define a graph H' whose set of edges is the union of the shortest paths trees from every node u to all nodes in its ball. This graph has the same approximation error and the same size as H; since H' is a subgraph of G it is called a *spanner* of G.

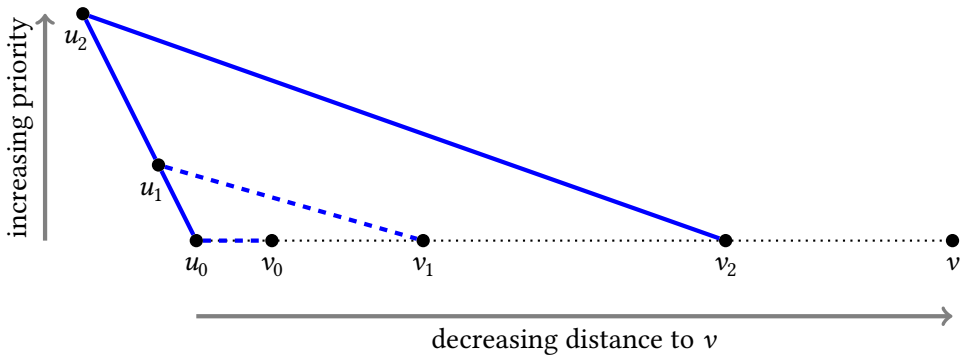


Figure 3.2: Illustration of the hop reduction for $p = 3$ priorities. The dotted line is the shortest path π from u_0 to v in G. The blue edges are the edges of F used to decrease the distance to v. The dashed blue edges are not contained in F and imply the existence of edges to nodes of increasing priority. Starting from u_0, a node of priority 0, we let v_0 be the node on π such that $d_G(u_0, v_0) = r_0 = 1$, i.e., the neighbor of u_0 on π. If the edge (u_0, v_0) is not contained in F, then F contains and edge (u_0, u_1) to a node u_1 of priority at least 1 such that $d_G(u_0, u_1) \leq r_0$. Let v_1 be the node on π such that $d_G(u_1, v_1) \leq r_1 = 1 + 2/\epsilon$. If the edge (u_1, v_1) is not contained in F, then F contains an edge (u_1, u_2) to a node u_2 of priority at least 2 such that $d_G(u_1, u_2) \leq r_1$. Let v_2 be the node on π such that $d_G(u_2, v_2) = r_2 = (1 + 2/\epsilon)(2 + 2/\epsilon)$. Since 2 is the highest priority, u_2 contains the edge (u_2, v_2). Note that the weight of these three edges from F is at most $r_0 + r_1 + r_2$ and $d_G(u_0, v_2) \geq r_2 - (r_0 + r_1)$. Since $r_2 = (1 + 2/\epsilon)(r_0 + r_1)$, the ratio between these two quantities is $(1 + \epsilon)$.

$0 \leq k \leq q - 2$. Then we have, for every $0 \leq k \leq q - 2$ and all pairs of nodes u and v

$$\begin{aligned} d_G(u, v) \leq d_H(u, v) &\leq (1 + \epsilon) d_G(u, v) + 2 \left(2 + \frac{2}{\epsilon}\right)^{p-2} \cdot \Delta_k \\ &\leq (1 + \epsilon) d_G(u, v) + \epsilon n^{1/p} \cdot \Delta_k \\ &= (1 + \epsilon) d_G(u, v) + \epsilon \Delta_{k+1} . \end{aligned}$$

Thus, if $\Delta_{k+1} \leq d_G(u, v) \leq \Delta_{k+2}$, then there is a path from u to v in H of weight at most

$$(1 + \epsilon) d_G(u, v) + \epsilon \Delta_{k+1} \leq (1 + \epsilon) d_G(u, v) + \epsilon d_G(u, v) = (1 + 2\epsilon) d_G(u, v)$$

and with at most $p\lceil d_G(u, v)/\Delta_k \rceil \leq (p + 1)\Delta_{k+2}/\Delta_k = (p + 1)n^{2/q} = n^{o(1)}$ edges. It follows that F is a $(2\epsilon, n^{o(1)})$ hop set of size $O(pn^{1+1/p}) = O(n^{1+o(1)})$. By running the algorithm with $\epsilon' = \epsilon/2$ we obtain an $(\epsilon, n^{o(1)})$ hop set of size $O(n^{1+o(1)})$.

Efficient Construction So far we have ignored the running time for computing the balls and thus constructing F, even in the static setting. Thorup and Zwick [126] have remarked that a naive algorithm for computing the balls takes time $O(mn)$. We can reduce this running time by sampling edges instead of nodes.

We modify the process for obtaining the sequence of sets $A_0, A_1, \ldots, A_p$ as follows. We set $A_0 = V$ and $A_p = \emptyset$ and for $1 \leq i \leq p-1$ we obtain the set A_i by picking each edge of E independently with probability $1/m^{i/p}$ and adding both endpoints of each sampled edge to A_i. The priority of a node u is the maximum i such that $u \in A_i$. We define, for every node u of priority i, $B(u)$ as the set of nodes

$$B(u) = \{v \in V \mid d_G(u,v) < d_G(v, A_{i+1})\}.$$

Note that the expected size of A_i is $O(m^{1-i/p})$ for every $1 \leq i \leq p-1$.

The balls can now be computed as follows. First, we compute $d_G(u, A_i) = \min_{v \in A_i} d_G(u,v)$ for every node u and every $1 \leq i \leq p-1$. Using Dijkstra's algorithm on a graph where we add an artificial source node s_i that is connected to every node in A_i by an edge of weight 0, this takes time $O(p(m + n \log n))$. Second, we compute for every node u of priority i a shortest paths tree *up to depth* $d_G(u, A_{i+1}) - 1$ to obtain all nodes contained in $B(u)$. Using an implementation of Dijkstra's algorithm that only puts nodes into its queue upon their first visit this takes time $O(|E[B(u)]| \log n)$ where $E[B(u)] = \{(u,v) \in E \mid u \in B(u) \text{ or } v \in B(u)\}$ is the set of edges incident to $B(u)$. By the sampling of edges we have, using the same ordering argument as before, that the expected size of $B(u)$ is $m^{(i+1)/p}$. For $0 \leq i \leq p-1$ the expected size of A_i is $O(m^{1-i/p})$ and thus these Dijkstra computations take time $O(m^{1+1/p} \log n)$ for all nodes of priority i. By choosing $p = \Theta(\sqrt{\log n})$ as described above we have $m^{1/p} = O(m^{o(1)})$ and thus the balls can be computed in time $O(m^{1+o(1)})$.

We define F as the set of edges $F = \{(u,v) \in V^2 \mid v \in B(u)\}$ and give each edge $(u,v) \in F$ the weight $w_F(u,v) = d_G(u,v)$. The distance-preserving and hop-reducing properties of F still hold as stated above and its expected size is $O(pm^{1+1/p})$. Note that F is not a sparsification of G anymore (as the bound on its size is even more than m). For our purposes the sparsification aspect is not relevant, we only need the hop reduction. Thus in the static setting, we can compute an $(\epsilon, m^{o(1)})$-hop set (which is also an $(\epsilon, n^{o(1)})$-hop set) of expected size $O(m^{1+o(1)})$ in expected time $O(m^{1+o(1)})$.

Maintaining Balls Under Edge Deletions As the graph G undergoes deletions the hop set has to be updated as well. We can achieve this by maintaining the balls w.r.t. a fixed sequence of randomly chosen sets $A_0, A_1, \ldots, A_p$, where $A_0 = V$ and $A_p = \emptyset$ and for $1 \leq i \leq p-1$ we obtain A_i by picking each edge of E independently with probability $c \ln n / m^{i/p}$, for some large enough constant c, and adding both endpoints of each sampled edge to A_i. Note that now, for every $1 \leq i \leq p-1$, the expected size of A_i is $O(m^{1-i/p} \log n)$ and the size of $E[B(u)]$ for every node u of priority i is at most $m^{(i+1)/p}$ *with high probability* (whp) at any time, i.e., in any version of the graph. This holds because by deleting edges there can only be a polynomial number of different versions of G. Unfortunately, we do not know how to maintain the balls efficiently. However we can maintain for all nodes u the approximate ball

$$B(u,D) = \{v \in V \mid \log d_G(u,v) < \lfloor \log d_G(u, A_{i+1}) \rfloor \text{ and } d_G(u,v) \leq D\}$$

(where u has priority i) in time $O(pm^{1+1/p}D\log D)$. Note that $B(u,D)$ differs from the definition of $B(u)$ in the following ways. First, we use the inequality $\log d_G(u,v) < \lfloor \log d_G(u,A_{i+1})\rfloor$ instead of the inequality $d_G(u,v) < d_G(u,A_{i+1})$. This alone increases the additive error of the hops set from $2(2+2/\epsilon)^{p-2}$ to $4(3+4/\epsilon)^{p-2}$, which can easily be compensated. Second, we limit the balls to a certain depth D. By using a small value of D we will only obtain a restricted hop set that provides sufficient hop reduction for nodes that are relatively close to each other. We will show later that this is enough for our purposes. Despite of these modifications we clearly have $B(u,D)\subseteq B(u)$ and therefore all size bounds still apply.

In the first part of the algorithm for maintaining the balls we maintain $d_G(u,A_i)$ every $1\le i\le p-1$ and every node u. We do this by adding an artificial source node s_i that has an edge of weight 0 to every node in A_i and maintain an ES-tree up to depth D from s_i. This step takes time $O(pmD)$.

Now, for every node u of priority i we maintain $B(u,D)$ as follows. We maintain an ES-tree up to depth

$$\min\left(2^{\lfloor \log d_G(u,A_{i+1})\rfloor}-1, D\right)$$

and every time $2^{\lfloor \log d_G(u,A_{i+1})\rfloor}$ increases, we restart the ES-tree. Naively, we incur a cost of $O(mD)$ for each instance of the ES-tree. However we can easily implement the ES-tree in a way that it never looks at edges that are not contained in $E[B(u)]$.[6] Thus, the cost of each instance of the ES-tree is $O(|B(u)|D)$. Remember that $B(u,D)\subseteq B(u)$ and that by the random sampling of edges the size of $E[B(u)]$ is at most $m^{(i+1)/p}$ whp. As $2^{\lfloor \log d_G(u,A_{i+1})\rfloor}$ can increase at most $\log D$ times until it exceeds D, we initialize at most $\log D$ ES-trees for the node u. Therefore the total time needs for maintaining $B(u,D)$ is $O(m^{(i+1)/p}D\log D)$. As there are at most $\tilde{O}(m^{1-i/p})$ nodes of priority i in expectation, the total time needed for maintaining all approximate balls is $\tilde{O}(pm^{1+1/p}D\log D)$ in expectation.

Decremental Approximate SSSP Let us first sketch an algorithm for maintaining shortest paths from a source node s with a running time of $O(m^{1+1/2+o(1)})$ for which we use $q=2$ layers and $p=\Theta(\sqrt{\log n})$ priorities. We set $\Delta=\sqrt{n}$, $D=2(3+4/\epsilon)^{p-2}\Delta$, and p such that $2(3+4/\epsilon)^{p-2}\le \epsilon n^{1/p}$ and $n^{1/p}=O(n^{o(1)})$. We maintain single-source shortest paths up to depth D from s using the ES-tree, which takes time $O(mD)=O(mn^{1/2+o(1)})$. To maintain approximate shortest paths to nodes that are at distance more than D from s we use the following approach. We maintain $B(u,D)$ for every node u, as sketched above, which takes time $\tilde{O}(pm^{1+1/p}D\log D)=O(m^{1+1/2+o(1)})$ in expectation. At any time, we set the hop set F to be the set of edges $F=\{(u,v)\in V^2 \mid \exists v\in B(u,D)\}$ and give each edge $(u,v)\in F$ the weight $w_F(u,v)=d_G(u,v)$. By our arguments above, the weighted graph $H=(V,F)$ has

[6] If we prefer to use the ES-tree as a "black box" we can, in a preprocessing step, find the initial set $B(u,D)$ and only build an ES-tree for this ball. All other nodes will never be contained in $B(u,D)$ anymore as long as the value of $2^{\lfloor \log d_G(u,A_{i+1})\rfloor}$ remains unchanged and therefore we can remove them. This can be done in time $O(|E[B(u,D)]|\log n)$ by using an implementation of Dijkstra's algorithm that only puts nodes into its queue upon their first visit.

the following property: for every pair of nodes u and v such that $d_G(u,v) \geq D \geq \Delta$ there is a path π in H of weight at most $(1+2\epsilon)d_G(u,v)$ and with at most $pd_G(u,v)/\Delta$ edges.

To maintain approximate shortest paths for nodes at distance more than D from s we will now use the hop reduction in combination with the following scaling technique. We set $\varphi = \epsilon\Delta/p$ and let H' be the graph resulting from rounding up every edge weight in H to the next multiple of φ. By using H' instead of H we incur an error of φ for every edge on the approximate shortest path path π. Thus in H', π has weight at most

$$(1+2\epsilon)d_G(u,v) + (pd_G(u,v)/\Delta)\cdot\varphi = (1+2\epsilon)d_G(u,v) + \epsilon d_G(u,v) \leq (1+3\epsilon)d_G(u,v)\,.$$

The efficiency now comes from the observation that we can run the algorithm on the graph H'' where every edge weight in H' is scaled down by a factor of $1/\varphi$. The graph H'' has integer weights and the weights of all paths in H' and H'' differ exactly by the factor $1/\varphi$. Thus, instead of maintaining a shortest paths tree up to depth n in H we only need to maintain a shortest paths tree in H'' up to depth $n/\varphi = p\sqrt{n}/\epsilon$. In this way we obtain a $(1+3\epsilon)$-approximation for all nodes such that $d_G(u,v) \geq D$.

However, we cannot simply use the ES-tree on H'' because as edges are deleted from G, nodes might join the approximate balls and therefore edges might be inserted into F and thus into H''. This means that a dynamic shortest paths algorithm running on H'' would not find itself in a purely decremental setting. However the insertions have a "nice" structure. We can deal with them by using a previously developed technique, called *monotone ES-tree*. The main idea of the monotone ES-tree is to ignore the level decreases suggested by inserting edges. The hop-set proof still goes through, even though we are not arguing about the current distance in H'' anymore, but the level of a node u in the monotone ES-tree. Maintaining the monotone ES-tree for distances up to D in H'' takes time $O(|\mathscr{E}(H)|D)$ where $\mathscr{E}(H'')$ is the set of edges ever contained in H'' (including those that are inserted over time) and $D = O(n^{1/2+1/p})$ as explained above. Each insertion of an edge into F corresponds to a node joining $B(u,D)$ for some node u. For a fixed node u of priority i there are at most $\log D$ possibilities for nodes to join $B(u,D)$ (namely each time $\lfloor \log d_G(u,A_{i+1}\rfloor$ increases) and every time at most $m^{(i+1)/p}$ nodes will join whp. It follows that $|\mathscr{E}(H)|$ is $O(m^{1+o(1)})$ whp and the running time of this step is $O(m^{1+1/2+o(1)})$ in expectation.

The almost linear-time algorithm is just slightly more complicated. Here we use $p = \Theta(\sqrt{\log n})$ priorities and $q = \sqrt{p}$ layers and set $\Delta_k = n^{k/q}$ for each $0 \leq k \leq q-2$. In the algorithm we will maintain, for each $0 \leq k \leq q-2$ a hop set F_k such that for every pair of nodes u and v with $\Delta_{k+1} \leq d_G(u,v) \leq \Delta_{k+2}$ there is a path from u to v in $H_k = (V, F_k)$ of weight at most $(1+2\epsilon)d_G(u,v)$ and with at most $pd_G(u,v)/\Delta_k \leq pn^{2/q}$ hops. To achieve this we use the following hierarchical approach. Given the hop set F_k we can maintain approximate shortest paths up to depth Δ_{k+2} in time $O(m^{1+o(1)})$ and given a data structure for maintaining approximate shortest paths up to depth Δ_k we can maintain approximate balls and thus the hop set F_{k+1} in time $O(m^{1+o(1)})$. The hierarchy "starts" with using the ES-tree as an algorithm for maintaining an (exact)

shortest paths tree up to depth $n^{2/q}$. Thus, running efficient monotone ES-trees on top of the hop sets and maintaining the hop sets (using efficient monotone ES-trees) go hand in hand.

There are two obstacles in implementing this hierarchical approach when we want to maintain the approximate balls in each layer. First, in our algorithm for maintaining the approximate balls we have used the ES-tree as an exact decremental SSSP algorithm. In the multilayer approach we have to replace the ES-tree with the monotone ES-tree which only provides approximate distance estimates. This will lead to approximation errors that increase with the number of layers. Second, by the arguments above the number of edges in F_k is $O(m^{1+1/p})$ for each $0 \leq k \leq q-2$. In the algorithm for maintaining the approximate balls for the next layer, this bound however is not good enough because we run a separate instance of the monotone ES-tree for each node u. We deal with this issue by running the monotone ES-tree in the subgraph of G induced by the nodes initially contained in $B(u)$. For a node u of priority i this subgraph contains $m_i = m^{(i+1)/p}$ edges whp and we can recursively run our algorithm on this smaller graph. By this process we incur a factor of $m^{1/p}$ in the running time each time each time we increase the depth of the recursion. This results in a total update time of $\tilde{O}(m^{1+q/p})$ which is $\tilde{O}(m^{1+1/q}) = O(m^{1+o(1)})$ since $q = \sqrt{p}$.

Extension to Weighted Graphs The hop set construction described above was only for unweighted graphs. However, the main property that we needed was $d_G(u,v) \leq n$ for any pair of nodes u and v. Using the scaling technique mentioned above, we can construct for each $0 \leq i \leq \lfloor \log nW \rfloor$ a graph G_i such that for all pairs of nodes u and v with $2^i \leq d_G(u,v) \leq 2^{i+1}$ we have $d_{G_i}(u,v) \leq 4n/\epsilon$ for some small γ and the shortest path in G_i can be turned into a $(1+\epsilon)$-approximate shortest path in G by scaling up the edge weights. We now run $O(\log(nW))$ instances of our algorithm, one for each graph G_i, and maintain the hop set and approximate SSSP for each of them.

We only need to refine the analysis of the hop-set property in the following way. Remember that in the analysis we considered the shortest path π from u to v and defined the node u' that is at distance Δ from u on π. If the hop set contained the edge (u,u') we could reduce the distance to v by Δ. In weighted graphs (even after the scaling), we cannot guarantee there is a node at distance Δ from u on π. Therefore we define u' as the furthest node that is at distance *at most* Δ from u on π. Furthermore we define u'' as the neighbor of u' on π, i.e., u'' is at distance *at least* Δ from u. Now if the hop set contains the edge (u,u') we first use the edge (u,u') from the hop set, and then the edge (u',u'') from the original graph to reduce the distance to v by at least Δ with only 2 hops. Note that for unweighted graphs it was sufficient to only use the edges of the hop set. For weighted graphs we really have to add the edges of the hop set to the original graph in our algorithm.

Outline As sketched above, our algorithm uses the following hierarchical approach: Given a decremental approximate SSSP algorithm for distances up to D_i with total update time $O(m^{1+o(1)})$, we can maintain approximate balls for distances up to D_i with total time $O(m^{1+o(1)})$ as well. And given a decremental algorithm for maintaining approximate balls for distances up to D_i with total update time $O(m^{1+o(1)})$ we can use the approximate balls to define a hop set which allows us to maintain approximate shortest paths for distances up to $D_{i+1} = n^{o(1)}D_i$ with total update time $O(m^{1+o(1)})$. This scheme is repeated until D_i is large enough to cover the full distance range.

We have formulated the two parts of this scheme as reductions. In Section 3.4 we give a decremental algorithm for maintaining approximate balls that internally uses a decremental approximate SSSP algorithm. In Section 3.5 we give a decremental approximate SSSP algorithm that internally uses a decremental algorithm for maintaining approximate balls. In Section 3.6 we explain the hierarchical approach for putting these two parts together and obtain the decremental $(1+\epsilon)$-approximate SSSP algorithm with a total update time of $O(m^{1+o(1)})$ for the full distance range. In addition to this result, the algorithm for maintaining approximate balls, together with a suitable query algorithm, gives us a decremental approximate APSP algorithm. This algorithm is also given in Section 3.6.

3.4 From Approximate SSSP to Approximate Balls

In the following we show how to maintain the approximate balls of every node if we already have an algorithm for maintaining approximate shortest paths. In our reduction we will use the algorithm for maintaining approximate shortest paths as a "black box", requiring only very few properties. Formally, we prove the following statement in this section.

Proposition 3.4.1. *Assume there is a decremental approximate SSSP algorithm* ApproxSSSP *with the following properties, using fixed values $\alpha \geq 1$, $\beta \geq 0$, and $D \geq 1$: Given a weighted graph $G = (V, E)$ undergoing edge deletions and edge weight increases and a fixed source node $s \in V$,* ApproxSSSP *maintains for every node $v \in V$ a distance estimate $\delta(s, v)$ such that:*

- ***A1*** *$\delta(s, v) \geq d_G(s, v)$*
- ***A2*** *If $d_G(s, v) \leq D$, then $\delta(s, v) \leq \alpha d_G(s, v) + \beta$. .*
- ***A3*** *After every update in G,* ApproxSSSP *returns, for every node v such that $\delta(s, v)$ has changed, v together with the new value of $\delta(s, v)$.*

We denote the total update time of ApproxSSSP *by $T(m, n)$.*

Then there is a decremental algorithm ApproxBalls *for maintaining approximate balls with the following properties: Given a weighted graph $G = (V, E)$ undergoing edge deletions and edge weight increases and parameters $2 \leq k \leq \log n$ and $0 < \epsilon \leq 1$,*

it assigns to every node $u \in V$ a number from 0 to $k-1$, called the priority *of u, and maintains for every node $u \in V$ a set of nodes $B(u)$ and a distance estimate $\delta(u,v)$ for every node $v \in B(u)$ such that:*

- ***B1*** *For every node u and every node $v \in B(u)$ we have $d_G(u,v) \le \delta(u,v) \le \alpha d_G(u,v) + \beta$.*
- ***B2*** *Let $s(\cdot,\cdot)$ be a non-decreasing function such that, for all $x \ge 1$, and $l \ge 1$,*

  $$s(x,l) \ge a(a+1)^{l-1} d_G(u,v) + ((a+1)^l - 1)b/a\,,$$

  where $a = (1+\epsilon)\alpha$ and $b = (1+\epsilon)\beta + 1$. Then for every node u of priority i and every node v such that $s(d_G(u,v), p-1-i) \le D$, either (1) $v \in B(u)$ or (2) there is some node v' of priority $j > i$ such that $u \in B(v')$ and $d_G(u,v') \le s(d_G(u,v), j-i)$.
- ***B3*** *If, for every node u, $\mathscr{B}(u)$ denotes the set of nodes ever contained in $B(u)$, then $\sum_{u\in V} |\mathscr{B}_u| = \tilde{O}(m^{1+1/k} \log D/\epsilon)$ in expectation.*
- ***B4*** *The update time of* ApproxBalls *is*

  $$t(m,n,k,\epsilon) = \tilde{O}\left(m^{1+1/k}\frac{\log D}{\epsilon} + \sum_{0\le i\le k-1} m^{1-i/k} \cdot T(m_i,n_i)\frac{\log D}{\epsilon} + T(m,n)\right),$$

  where, for each $0 \le i \le k-1$, $m_i = m^{(i+1)/k}$ and $n_i = \min(m_i, n)$.
- ***B5*** *After every update in G,* ApproxBalls *returns all pairs of nodes u and v such that v joins $B(u)$, v leaves $B(u)$, or $\hat{\delta}(u,v)$ changes.*

Our algorithm for maintaining the approximate balls $B(u)$ for every node $u \in V$ is as follows:

1. At the initialization we set $F_0 = E$ and $F_k = \emptyset$ and for $1 \le i \le k-1$, a set of edges F_i is obtained from sampling each edge of E independently with probability $(c \ln n)/m^{i/k}$ (for a large enough constant c). For every $0 \le i \le k-1$ we set $A_i = \{v \in V \mid \exists (v,w) \in F_i\}$ and for every node $v \in V$, we set the *priority of u* to be the maximum i such that $v \in A_i$.
2. For each $1 \le i \le k-1$ we run an instance of ApproxSSSP with depth D from an artificial source node s_i that has an edge of weight 0 to every node in A_i. We denote the distance estimate provided by ApproxSSSP $\delta(u, A_i)$ and set $\delta(u, A_k) = \infty$ for every node $u \in V$.
3. For every $0 \le i \le k-1$ and every node $u \in V$ of priority i, we maintain the value

   $$r(u) = \min\left(\frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha}, D\right).$$

   and at the initialization and each time $r(u)$ increases we do the following:

a) Compute the set of nodes $R(u) = \{v \in V \mid d_G(u, v) \leq r(u)\}$.

b) Run an instance of ApproxSSSP with depth D from u in $G|R(u)$, the subgraph of G induced by $R(u)$. Let $\delta(u, v)$ denote the estimate of the distance between u and v in $G|R(u)$ maintained by ApproxSSSP.

c) Maintain $B(u) = \{v \in V \mid \delta(u, v) \leq \alpha D + \beta\}$: every time $\delta(u, v)$ changes for some node v we check whether v is still contained in $B(u)$.

Note that ApproxBalls has Property **B5**, i.e., it returns changes in the approximate balls and the distance estimates, which is possible because ApproxSSSP has Property **A3**.

3.4.1 Relation to Exact Balls

In the following we compare the approximate balls maintained by our algorithm to the exact balls, as used by Thorup and Zwick [126]. We show how the main properties of exact balls translate to approximate balls. We use the following definition of the (exact) ball of a node u of priority i:

$$B(u) = \{v \in V \mid d_G(u, v) < d_G(u, A_{i+1})\}.$$

The balls have the following simple property: If $v \notin B(u)$, then there is a node v' of priority $j > i$ such that $d_G(u, v') \leq d_G(u, v)$. We show that a relaxed version of this statement also holds for the approximate balls.

Lemma 3.4.2. *Let u be a node of priority i and let v be a node such that $d_G(u, v) \leq D$. If $v \notin B(u)$, then there is a node v' of priority $j > i$ such that $d_G(u, v') \leq ad_G(u, v) + b$, where $a = (1 + \epsilon)\alpha$ and $b = (1 + \epsilon)\beta + 1$.*

Proof. We show the following: If $d_G(u, A_{i+1}) \geq ad_G(u, v)$, then $v \in B(u)$. The claim then follows from contraposition: If $v \notin B(u)$, then $d_G(u, A_{i+1}) < ad_G(u, v) + b$ and thus there exists some node $v' \in A_{i+1}$ that has priority $j \geq i + 1$ such that $d_G(u, v') < ad_G(u, v) + b$.

Assume that $d_G(u, A_{i+1}) \geq ad_G(u, v) + b$. Remember that we have set

$$r(u) = \min\left(\frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha}, D\right).$$

Since $\delta(u, A_{i+1}) \geq d_G(u, A_{i+1})$ by Property **A1** we have

$$\delta(u, A_{i+1}) \geq d_G(u, A_{i+1}) \geq ad_G(u, v) + b = (1 + \epsilon)(\alpha d_G(u, v) + \beta) + 1$$

which is equivalent to

$$d_G(u, v) \leq \frac{\frac{\delta(u,A_{i+1})-1}{1+\epsilon} - \beta}{\alpha}.$$

Since $(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} \geq (1+\epsilon)^{\log_{1+\epsilon}(\delta(u,A_{i+1})-1)-1} = (\delta(u,A_{i+1})-1)/(1+\epsilon)$ it follows that

$$d_G(u,v) \leq \frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha}$$

Since we have assumed that $d_G(u,v) \leq D$ we get $d_G(u,v) \leq r(u)$ which implies that $d_{G|R(u)}(u,v) \leq r(u) \leq D$ as well. Thus, by Property **A2**, it follows that $\delta(u,v) \leq \alpha d_{G|R(u)} + \beta \leq \alpha D + \beta$ and $v \in B(u)$ as desired. □

We now show that the approximate balls are contained in the exact balls. The exact balls are useful in our analysis because we can easily bound their size.

Lemma 3.4.3. *At any time $B(u) \subseteq B(u)$ for every node u.*

Proof. Let $R(u) = \{v \in V \mid d_G(u,v) \leq r(u)\}$ denote the set of nodes at distance at most $r(u)$ from u at the last time $r(u)$ has increased. Note that $B(u)$ is a set of nodes of the graph $G|R(u)$ and therefore $B(u) \subseteq R(u)$. It remains to show that $R(u) \subseteq B(u)$.

If $i = k-1$, then the claim is trivially true because $B(u)$ contains all nodes that are connected to u in G. In the case $0 \leq i < k-1$ remember that

$$r(u) = \min\left(\frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha}, D\right).$$

If $d_G(u,A_{i+1}) > r(u)$, we trivially have $d_G(u,v) \leq r(u) < d_G(u,A_{i+1})$. If $d_G(u,A_{i+1}) \leq r(u)$, then in particular $d_G(u,A_{i+1}) \leq D$ and by Property **A2** we have $\delta(u,A_{i+1}) \leq \alpha d_G(u,A_{i+1}) + \beta$. It follows that

$$\begin{aligned} d_G(u,v) \leq r(u) &\leq \frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha} \\ &\leq \frac{(1+\epsilon)^{\log_{1+\epsilon}(\delta(u,A_{i+1})-1)} - \beta}{\alpha} \leq \frac{\delta(u,A_{i+1}) - 1 - \beta}{\alpha} \leq d_G(u,A_{i+1}) - \frac{1}{\alpha}. \end{aligned}$$

This is implies that $d_G(u,v) < d_G(u,A_{i+1})$ because the distances are integer and $1/\alpha > 0$. In both cases we get $d_G(u,v) < d_G(u,A_{i+1})$ and as this is the defining property of $B(u)$ we have $v \in B(u)$. □

Lemma 3.4.4. *At any time the size of $B(u)$ is $O(m^{(i+1)/k})$ whp for every node u of priority i.*

Proof. Note that the claim is trivially true for $i = k-1$. Therefore assume $i < k-1$ in the following. For every edge $e = (v,w) \in E$ we define $d_G(u,e) = \min(d_G(u,v), d_G(v,w))$. Let $F \subseteq E$ be the set consisting of the $m^{(i+1)/k}$ edges that are closest to u according to this definition of $d_G(u,e)$ for each edge e, where ties are broken in an arbitrary but fixed order. (Note that if less than $m^{(i+1)/k}$ edges are connected to u, then the claim is true anyway).

Let U be the set of nodes $U = \{v \in V \mid \exists (v,w) \in F\}$. By the random sampling (Lemma 1.3.2), F contains an edge (v,w) of F_{i+1} with high probability. Assume

without loss of generality that $d_G(u,v) \leq d_G(v,w)$, i.e., $d_G(u,e) = d_G(u,v)$. Let $v' \in B(u)$ and let $e' = (v', w')$ be some edge incident to v'. Since the node v is contained in A_{i+1} we have

$$d_G(u,e') \leq d_G(u,v') < d_G(u,A_{i+1}) \leq d_G(u,v) = d_G(u,e)\,.$$

Therefore we have $e' \in F$ and thus $v' \in U$. It follows that $B(u) \subseteq U$. Observe that $|U| \leq 2|F|$ and thus $|B(u)| \leq |U| \leq 2|F| = 2m^{(i+1)/k}$ whp. □

3.4.2 Properties of Approximate Balls

We now show that the approximate balls and the corresponding distance estimate have the Properties **B1**–**B4**. We first show that the distance estimates for nodes in the approximate balls have the desired approximation guarantee, although they have been computed in subgraphs of G.

Lemma 3.4.5 (Property **B1**). *For every pair of nodes u and v such that $v \in B(u)$ we have $d_G(u,v) \leq \delta(u,v) \leq \alpha d_G(u,v) + \beta$.*

Proof. By Property **A1** we have $\delta(u,v) \geq d_{G|R(u)}(u,v)$ and since $G|R(u)$ is a subgraph of G we have $d_{G|R(u)}(u,v) \geq d_G(u,v)$. Therefore the inequality $\delta(u,v) \geq d_G(u,v)$ follows.

Since $v \in B(u)$ we have $\delta(u,v) \leq \alpha D + \beta$. If $d_G(u,v) \geq D$, then trivially $\delta(u,v) \leq \alpha D + \beta \leq \alpha d_G(u,v) + \beta$. If $d_G(u,v) \geq D$, then there is a path k from u to v in G of weight at most D. This path was also contained in previous versions of G, possibly with smaller weight, and in particular at the time the algorithm has computed $R(u)$, the set of nodes that are at distance at most D from u in G, for the last time. It follows that k is also contained in $G|R(u)$ and thus $d_{G|R(u)}(u,v) = d_G(u,v) \leq D$. By Property **A2** we then have $\delta(u,v) \leq \alpha d_{G|R(u)} + \beta = \alpha d_G + \beta$. □

We show now that the approximate balls have a certain structural property that either allows us shortcut the path between two nodes or helps us in finding a nearby node of higher priority.

Lemma 3.4.6 (Property **B2**). *Let $s(\cdot,\cdot)$ be a non-decreasing function such that, for all $x \geq 1$, and $l \geq 1$,*

$$s(x,l) \geq a(a+1)^{l-1} d_G(u,v) + ((a+1)^l - 1)b/a\,,$$

where $a = (1+\epsilon)\alpha$ and $b = (1+\epsilon)\beta + 1$. Then for every node u of priority i and every node v such that $s(d_G(u,v), p-1-i) \leq D$, either (1) $v \in B(u)$ or (2) there is some node v' of priority $j > i$ such that $u \in B(v')$ and $d_G(u,v') \leq s(d_G(u,v), j-i)$.

Proof. We first define the following series: let $f(1) = a d_G(u,v) + b$ and for all $l \geq 1$ let $f(l+1) = f(l) + a f(l) + b$. It is easy to verify that for all $l \geq 1$ we have

$$f(l) = \frac{a^2(a+1)^{l-1} d_G(u,v) + ((a+1)^l - 1)b}{a} \leq s(d_G(u,v), l)\,.$$

Note that since $s(\cdot,\cdot)$ is non-decreasing we have, for all $1 \le l \le k-1-i$, $f(l) \le s(d_G(u,v),l) \le s(d_G(u,v),p-1-i) \le D$.

If $v \in B(u)$, then we are done. Otherwise, by Lemma 3.4.2, there is some node v_1 of priority $p_1 \ge i+1$ such that

$$d_G(v_1,u) \le a d_G(u,v) + b = f(1) \le D\,.$$

Thus, if $u \in B(v_1)$, then we are done. Otherwise, by Lemma 3.4.2, there is some node v_2 of priority $p_2 \ge p_1 + 1 \ge i+2$ such that

$$d_G(v_2,v_1) \le a d_G(v_1,u) + b \le a f(1) + b\,.$$

By the triangle inequality we have

$$d_G(v_2,u) \le d_G(v_2,v_1) + d_G(v_1,u) \le a f(1) + b + f(1) = f(2) \le D\,.$$

We now repeat this argument to obtain nodes $v_1, v_2, \ldots v_l$ of priorities $p_1, p_2, \ldots, p_l$ such that $d_G(v_l,u) \le f(l) \le D$ and $p_l \ge i+l$ until the inequality $d_G(v_l, A_{p_l+1}) \ge a d_G(v_l,u)+b$ is fulfilled. This happens eventually since $A_k = \emptyset$ and thus $d_G(u,A_k) = \infty$. □

Next, we bound the size of the system of approximate balls we maintain. Here we use the fact that we can easily bound the size of the exact ball $B(u)$ for every node u and that by our definitions we ensure that the approximate balls are subsets of the exact balls.

Lemma 3.4.7 (Size of Approximate Balls (Property **B3**))**.**

If, for every node u, $\mathscr{B}(u)$ denotes the set of nodes ever contained in $B(u)$, then $\sum_{u\in V} |\mathscr{B}_u| = \tilde{O}(m^{1+1/k} \log_D/\epsilon)$ in expectation.

Proof. We first bound $\mathscr{B}_u$, the number of nodes ever contained in the approximate ball $B(u)$, of some node u of priority i. Remember that nodes are joining $B(u)$ only when $r(u)$ increases and that

$$r(u) = \min\left(\frac{(1+\epsilon)^{\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor} - \beta}{\alpha}, D\right).$$

Thus, $r(u)$ can only increase if $\lfloor \log_{1+\epsilon}(\delta(u,A_{i+1})-1)\rfloor$ increases and the left term in the minimum is at most D. Since $1+\epsilon \ge 1$ and $\beta \ge 0$ it follows that $r(u)$ increases only $O(\log_{1+\epsilon} D) = O(\log D/\epsilon)$ times. As $B(u) \subseteq B(u)$ by Lemma 3.4.3, after every increase of $r(u)$ only nodes contained in $B(u)$ can join $B(u)$. By Lemma 3.4.4 the size of $B(u)$ is $O(m^{(i+1)/k})$ whp. Thus, the number of nodes ever contained in $B(u)$ is $|\mathscr{B}(u)| = O(m^{(i+1)/k} \log D/\epsilon)$ whp.

As the number of nodes of priority i is $\tilde{O}(m/m^{i/k})$ in expectation, the number of nodes ever contained in the approximate balls is $\sum_{u\in V} |\mathscr{B}(u)| \tilde{O}(m^{1+1/k} \log D/\epsilon)$ in expectation. □

Finally, we analyze the running time of our algorithm for maintaining the approximate balls. Since we use the data structure ApproxSSSP as a black box, the running time of our algorithm depends on the running time of ApproxSSSP.

Lemma 3.4.8 (Running Time (Property **B4**)). *The total time needed for maintaining the sets $B(u)$ for all nodes $u \in V$ is*

$$\tilde{O}\left(m^{1+1/k} \log D/\epsilon + \sum_{0 \le i \le k-1} T(m_i, n_i) \log D/\epsilon + T(m, n) \right),$$

where, for each $0 \le i \le k-1$, $m_i = m^{(i+1)/k}$ and $n_i = \min(m_i, n)$.

Proof. The initialization in Step 1 of the algorithm, where we determine the sets $A_0, \ldots, A_p$ takes time $O(pm)$. In Step 2, we run for each $1 \le i \le k-1$ an instance of ApproxSSSP with depth D. This takes time $kT(m, n)$. Step 3, where we maintain for every node u of priority i the approximate ball and corresponding distance estimates can be analyzed as follows. Remember that every time $r(u)$ increases we first compute $R(u)$, the set of nodes that are at distance at most $r(u)$ from u. Using a suitable implementation of Dijkstra's algorithm, this takes time $O(|E[R(u)]| \log n)$, where $E[R(u)]$ is the set of edges incident to $R(u)$. By Lemma 3.4.3 we have $R(u) \subseteq B(u)$ and by Lemma 3.4.4 we have $|B(u)| = O(m^{(i+1)/k})$ whp. Thus, computing $R(u)$ takes time $\tilde{O}(m^{(i+1)/k})$ whp. We then maintain an instance of ApproxSSSP up to depth D on $G|R(u)$, the subgraph of G induced by $R(u)$. Note that $G|R(u)$ has $m_i = m^{(i+1)/k}$ edges and $n_i = \min(m_i, n)$ nodes and therefore this takes time $T(m_i, n_i)$. As $r(u)$ increases $O(\log D/\epsilon)$ times and the number of nodes of priority i is $\tilde{O}(m/m^{i/k})$ in expectation, maintaining $B(u)$ for all nodes u of priority i (and the corresponding distance estimates) takes time

$$\sum_{0 \le i \le k-1} \tilde{O}\left(\frac{m}{m^{i/k}} \log D/\epsilon \cdot (m^{(i+1)/k} + T(m_i, n_i) \right) = \tilde{O}\left(m^{1+1/k} \log D/\epsilon + \sum_{0 \le i \le k-1} m^{1-i/k} \cdot T(m_i, n_i) \log D/\epsilon \right).$$

Since $k \le \log n$, the claimed running time follows. □

3.5 From Approximate Balls to Approximate SSSP

In the following we show how to maintain an approximate shortest paths tree if we already have an algorithm for maintaining approximate balls. Our main tool in this reduction is a hop set that we define from the approximate balls. We will add the "shortcut" edges of the hop set to the graph and scale down the edge weights, maintaining the approximate shortest paths with a monotone ES-tree, as introduced in Chapter 2. Formally, we prove the following statement in this section.

Proposition 3.5.1. *Assume there is a decremental algorithm* ApproxBalls *for maintaining approximate balls with the following properties, using fixed values $a \geq \alpha \geq 1$, $b \geq \beta \geq 0$, and $\hat{D} \geq 1$. Given a weighted graph $G = (V, E)$ undergoing edge deletions and edge weight increases and a parameters $2 \leq k \leq \log n$, it assigns to every node $u \in V$ a number from 0 to $k-1$, called the* priority *of u, and maintains for every node $u \in V$ a set of nodes $B(u)$ and, for every node $v \in B(u)$, a distance estimate $\hat{\delta}(u, v)$ such that:*

- ***B1*** *For every node u and every node $v \in B(u)$ we have $d_G(u, v) \leq \hat{\delta}(u, v) \leq \alpha d_G(u, v) + \beta$.*
- ***B2*** *There is a function $s(\cdot, \cdot)$ such that, for all $x \geq 1$, $s(x, 1) = ax + b$ for some $a \geq \alpha$ and $b \geq \beta$ and, for all $l \geq 1$,*

$$s(x, l+1) \leq \frac{a(\alpha + 1 + \epsilon)(\alpha s(x, l) + \beta) + \beta}{\epsilon} + b\,.$$

  guaranteeing the following: For every node u of priority i and every node v such that $s(d_G(u, v), p - 1 - i) \leq \hat{D}$, either (1) $v \in B(u)$ or (2) there exist some node $v' \in V$ of priority $j > i$ such that $u \in B(v')$ and $d_G(u, v') \leq s(d_G(u, v'), j - i)$.
- ***B3*** *After every update in G,* ApproxBalls *returns all pairs of nodes u and v such that v joins $B(u)$, v leaves $B(u)$, or $\hat{\delta}(u, v)$ changes.*

For every node u, let $\mathscr{B}(u)$ denote the set of nodes ever contained in $B(u)$ and let $t(m, n, k)$ denote the total update time of ApproxBalls.

Then there is an approximate SSSP data structure ApproxSSSP *with the following properties: Given a weighted graph G undergoing edge deletions and edge weight increases, a fixed source node s, and parameters p, Δ, D, and ϵ such that*

$$2 \leq p \leq \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}}\,,$$

$\Delta \geq b$, $n^{1/p}\Delta \leq \hat{D}$, $D \geq \Delta$ and $0 < \epsilon \leq 1$, it maintains a distance estimate $\delta(s, v)$ for every node $v \in V$ such that:

- ***A1*** *$\delta(s, v) \geq d_G(s, v)$*
- ***A2*** *If $d_G(s, v) \leq D$, then $\delta(s, v) \leq (\alpha + \epsilon) d_G(s, v) + \epsilon n^{1/p}\Delta$*
- ***A3*** *The total update time of* ApproxSSSP *is*

$$T(m, n, \Delta, D, \epsilon) = \tilde{O}((\alpha D/\Delta + n^{1/p}) \sum_{u \in V} (m + |\mathscr{B}(u)|)/\epsilon + t(m, n, p))\,.$$

- ***A4*** *After every update in G,* ApproxSSSP *returns, for every node v such that $\delta(s, v)$ has changed, v together with the new value of $\delta(s, v)$.*

We assume without loss of generality that the distance estimate maintained by ApproxBalls is non-decreasing. If ApproxBalls ever reports a decrease we can ignore it because Property **B1** will still hold as distances in G are non-decreasing under edge deletions and edge weight increases.

3.5.1 Algorithm Description

The algorithm ApproxSSSP maintains the set of edges $F = \{(u, v) \in V^2 \mid v \in B(u)\}$ such that each edge $(u, v) \in F$ has weight $w_F(u, v) = \delta(u, v)$. We update F every time in ApproxBalls a node joins or leaves an approximate ball or if the distance estimate $\delta(u, v)$ increases for some pair of nodes u and v. By Property **B3** this information is returned by ApproxBalls after every update in G. Thus the set of edges F undergoes insertions, deletions, and weight increases.

In the following we will define a shortcut graph H'' with scaled-down edge weights and our algorithm ApproxSSSP will simply run a monotone ES-tree from s in H''. Note that the monotone ES-tree trivially has properties **A1** and **A4**. We denote the weight of an edge (u, v) in G by $w_G(u, v)$ and define H as a graph that has the same nodes as G and contains all edges of G and F that have weight at most $D+n^{1/p}\Delta$. We set the weight of every edge (u, v) in G to $w_H = \min(w_G(u, v), w_F(u, v))$. We set

$$\varphi = \frac{\epsilon\Delta}{p+1}$$

and define H' as the graph that has the same nodes and edges as H and in which every edge (u, v) has weight

$$w_{H'}(u, v) = \left\lceil \frac{w_H(u, v)}{\varphi} \right\rceil \varphi,$$

i.e., we round every edge weight to the next multiple of φ. Furthermore, we define H'' as the graph that has the same nodes and edges as H' and in which every edge (u, v) has weight

$$w_{H''}(u, v) = \frac{w_{H'}(u, v)}{\varphi} = \left\lceil \frac{w_H(u, v)}{\varphi} \right\rceil,$$

i.e., we scale down every edge weight by a factor of $1/\varphi$. We maintain a monotone ES-tree with maximum level

$$L = (\alpha + 2\epsilon)D/\varphi + (p+1)n^{1/p}$$

from s and denote the level of a node v in this tree by $\ell(v)$. For every node v our algorithm returns the distance estimate $\delta(s, v) = \ell(v) \cdot \varphi$. Note that the graph H'' has integer edge weights and, as F might undergo insertions, deletions, and edge weight increases, the same type of updates might occur in H''. Furthermore, observe that the rounding guarantees that

$$w_H(u, v) \leq w_{H'}(u, v) \leq w_H(u, v) + \varphi$$

for every edge (u, v) of H'.

3.5.2 Running Time Analysis

We first provide the running time analysis. We run the algorithm in a graph in which we scale down the edge weights by a factor of φ. This makes the algorithm efficient.

Lemma 3.5.2 (Running Time (Property **A3**)). *The expected total update time of a monotone ES-tree with maximum level $L = (\alpha + 2\epsilon)D/\varphi + (p+1)n^{1/p}$ on H'' is*

$$\tilde{O}\left(\left(\alpha D/\Delta + n^{1/p}\right)\left(m + \sum_{u \in V} |\mathscr{B}(u)|\right)/\epsilon\right).$$

Proof. By Lemma 3.2.1 the total time needed for maintaining the monotone ES-tree with maximum level L on H'' is

$$O(\mathscr{E}(H'') \cdot L + \mathscr{W}(H''))$$

where $\mathscr{E}(H'')$ is the number of edges ever contained in H'' and $\mathscr{W}(H'')$ is the number of updates (i.e., edge deletions, edge weight increases, and edge insertions) on H''.

Remember that $\varphi = \epsilon\Delta/(p+1)$. Since $\epsilon \leq 1$ and $p \leq \log n$ we have $L = \tilde{O}(\alpha D/(\epsilon\Delta) + n^{1/p})$. We now bound $\mathscr{E}(H'')$ and $\mathscr{W}(H'')$. Note that at any time H'' has the same edges as H and each edge of H either is also an edge in G, which contains m edges, or is an edge from F. As F is defined via the approximate balls (i.e., $(u,v) \in F$ if and only $v \in B(u)$), the number of edges ever contained in F is at most $\sum_{u \in v} |\mathscr{B}(u)|$, the total number of nodes ever contained in the approximate balls. It follows that $\mathscr{E}(H'') = \tilde{O}(m + \sum_{u \in v} |\mathscr{B}(u)|)$ in expectation. Note that every edge contained in H'' can be inserted and deleted at most once and its weight can increase at most $(D + n^{1/p}\Delta)/\varphi$ times as we have limited the maximum edge weight in H to $D + n^{1/p}\Delta$. Note that $(D + n^{1/p}\Delta)/\varphi = (D + n^{1/p}\Delta)(p+1)/(\epsilon\Delta) = \tilde{O}((D/\Delta + n^{1/p})/\epsilon)$. Therefore we have

$$\mathscr{W}(H'') \leq 2\mathscr{E}(H'') + \mathscr{E}(H'') \cdot (D + n^{1/p}\Delta)/\varphi = \tilde{O}((m + \sum_{u \in v} |\mathscr{B}(u)|) \cdot (D/\Delta + n^{1/p})/\epsilon).$$

We conclude that

$$\mathscr{E}(H'') \cdot L + \mathscr{W}(H'') = \tilde{O}((\alpha D/\Delta + n^{1/p})(m + \sum_{u \in v} |\mathscr{B}(u)|)/\epsilon)$$

and thus the claimed running time follows. □

3.5.3 Definitions of Values for Approximation Guarantee

Before we analyze the approximation guarantee we define the following values. We set

$$r_0 = \Delta$$

and for every $0 \le i \le p-1$ we set

$$\begin{aligned} s_i &= ar_i + b\,, \\ w_i &= \alpha s_i + \beta\,, \text{ and} \\ r_i &= \frac{(\alpha+1+\epsilon)\sum_{0\le j\le i-1} w_j + \beta}{\epsilon} \text{ (if } i \ge 1)\,. \end{aligned}$$

Finally, we set

$$\gamma_{p-1} = \beta$$

and, for every $0 \le i \le p-2$,

$$\gamma_i = \gamma_{i+1} + (\alpha+1+\epsilon)w_i = (\alpha+1+\epsilon)\sum_{i\le j\le p-2} w_j + \beta\,.$$

We also set

$$\gamma = \gamma_0 + 2\epsilon\Delta\,.$$

Lemma 3.5.3. *For all* $0 \le i \le p-1$, $\epsilon r_i = \gamma_0 - \gamma_i + \beta$

Proof. First, observe that for all $0 \le i \le p-1$ we have

$$\gamma_i = (\alpha+1+\epsilon)\sum_{i\le j\le p-2} w_j + \beta\,.$$

Thus, for all $0 \le i \le p-1$, we get

$$\begin{aligned} \gamma_0 - \gamma_i + \beta &= (\alpha+1+\epsilon)\sum_{0\le j\le p-2} w_j - (\alpha+1+\epsilon)\sum_{i\le j\le p-2} w_j + \beta \\ &= (\alpha+1+\epsilon)\sum_{0\le j\le i-1} w_j + \beta = \epsilon r_i\,. \end{aligned}$$ □

Lemma 3.5.4. $(4a^3/\epsilon)^p = n^{1/p}$

Proof. Remember that we have

$$p \le \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}}\,.$$

We only need to simplify both expressions as follows:

$$\begin{aligned} n^{1/p} &= 2^{1/p\cdot\log n} \ge 2^{\frac{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}}{\sqrt{\log n}}\cdot\log n} = 2^{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}\cdot\sqrt{\log n}} \\ \left(\frac{4a^3}{\epsilon}\right)^p &= 2^{p\cdot\log\left(\frac{4a^3}{\epsilon}\right)} \le 2^{\frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}}\cdot\log\left(\frac{4a^3}{\epsilon}\right)} = 2^{\sqrt{\log n}\cdot\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}}\,. \end{aligned}$$ □

Lemma 3.5.5. *For all* $0 \le i \le p-1$ *we have*

$$r_i \le \frac{3 \cdot 4^{i-1} a^{3i} \Delta + (9 \cdot 4^{i-1} - 2) a^{3i-1} b}{\epsilon^i}$$

and

$$\sum_{0 \le j \le i} w_j \le \frac{4^i a^{3i+2} \Delta + (3 \cdot 4^i - 1) a^{3i+1} b}{\epsilon^i}.$$

Proof. Remember that $\epsilon \le 1 \le \alpha \le a$. Now observe that for all $1 \le i \le p-1$ we have

$$r_i = \frac{(\alpha + 1 + \epsilon) \sum_{0 \le j \le i-1} w_j + \beta}{\epsilon} \le \frac{(a + 1 + \epsilon) \sum_{0 \le j \le i-1} w_j + b}{\epsilon} \le \frac{3a \sum_{0 \le j \le i-1} w_j + b}{\epsilon}$$

and for all $0 \le i \le p-1$ we have

$$w_i = \alpha s_i + \beta \le a s_i + b = a(a r_i + b) + b = a^2 r_i + ab + b \le a^2 r_i + 2ab.$$

We now prove the inequalities induction on i. We begin with the base case $i = 0$ where $r_0 = \Delta$ and

$$\sum_{0 \le j \le 0} w_j = w_0 \le a^2 r_0 + 2ab = a^2 \Delta + 2ab = \frac{4^0 a^{3 \cdot 0 + 2} \Delta + (3 \cdot 4^0 - 1) a^{3 \cdot 0 + 1} b}{\epsilon^0}.$$

In the induction step we assume that $i \ge 1$:

$$\begin{aligned} r_i &\le \frac{3a \sum_{0 \le j \le i-1} w_j + b}{\epsilon} \\ &\le \frac{3a(4^{i-1} a^{3(i-1)+2} \Delta + (3 \cdot 4^{i-1} - 1) a^{3(i-1)+1} b) + b}{\epsilon^i} \\ &= \frac{3 \cdot 4^{i-1} a^{3i} \Delta + (9 \cdot 4^{i-1} - 2) a^{3i-1} b}{\epsilon^i} \end{aligned}$$

$$\begin{aligned} \sum_{0 \le j \le i} w_j &= \sum_{0 \le j \le i-1} w_j + w_i \\ &\le \sum_{0 \le j \le i-1} w_j + a^2 r_i + 2ab \\ &\le \frac{4^{i-1} a^{3(i-1)+2} \Delta + (3 \cdot 4^{i-1} - 1) a^{3(i-1)+1} b}{\epsilon^{i-1}} + a^2 r_i + 2ab \\ &\le \frac{4^{i-1} a^{3(i-1)+2} \Delta + (3 \cdot 4^{i-1} - 1) a^{3(i-1)+1} b}{\epsilon^{i-1}} \\ &\quad + \frac{3 \cdot 4^{i-1} a^{3i+2} \Delta + (9 \cdot 4^{i-1} - 2) a^{3i+1} b}{\epsilon^i} + 2ab \\ &\le \frac{4^{i-1} a^{3(i-1)+2} \Delta + (3 \cdot 4^{i-1} - 1) a^{3(i-1)+1} b}{\epsilon^i} \end{aligned}$$

$$
\begin{aligned}
&\quad + \frac{3 \cdot 4^{i-1} a^{3i+2} \Delta + (9 \cdot 4^{i-1} - 2) a^{3i+1} b + 2ab}{\epsilon^i} \\
&\leq \frac{4^{i-1} a^{3i+2} \Delta + (3 \cdot 4^{i-1} - 1) a^{3i+1} b}{\epsilon^i} \\
&\quad + \frac{3 \cdot 4^{i-1} a^{3i+2} \Delta + (9 \cdot 4^{i-1} - 2) a^{3i+1} b + 2a^{3i+1} b}{\epsilon^i} \\
&= \frac{(1+3) \cdot 4^{i-1} a^{3i+2} \Delta + (3 \cdot 4^{i-1} - 1 + 9 \cdot 4^{i-1} - 2 + 2) a^{3i+1} b}{\epsilon^i} \\
&= \frac{4^i a^{3i+2} \Delta + (3 \cdot 4^i - 1) a^{3i+1} b}{\epsilon^i} .
\end{aligned}
$$

□

Lemma 3.5.6. $a\gamma + b \leq \epsilon n^{1/p} \Delta$.

Proof. Remember that we have $\epsilon \leq 1 \leq \alpha \leq a$ and $\beta \leq b \leq \Delta$. By Lemma 3.5.5 we have

$$
\sum_{0 \leq j \leq p-2} w_j \leq \frac{4^{p-2} a^{3(p-2)+2} \Delta + (3 \cdot 4^{p-2} - 1) a^{3(p-2)+1} b}{\epsilon^{p-2}} \leq \frac{4^{p-2} a^{3p-2} \Delta + 3 \cdot 4^{p-2} a^{3p-2} \Delta}{\epsilon^{p-1}}
$$

We now get:

$$
\begin{aligned}
\frac{a\gamma + b}{\epsilon} &= \frac{a\gamma_0 + 2\epsilon a \Delta + \beta}{\epsilon} \\
&= \frac{a(\alpha + 1 + \epsilon) \sum_{0 \leq j \leq p-2} w_j + \alpha\beta + 2\epsilon a \Delta + \beta}{\epsilon} \\
&\leq \frac{a(a + 1 + \epsilon) \sum_{0 \leq j \leq p-2} w_j + a\Delta + 2a\Delta + \Delta}{\epsilon} \\
&\leq \frac{3a^2 \sum_{0 \leq j \leq p-2} w_j + 4a\Delta}{\epsilon} \\
&\leq \frac{3 \cdot 4^{p-2} a^{3p} \Delta + 9 \cdot 4^{p-2} a^{3p} \Delta + 4a\Delta}{\epsilon^p} \\
&\leq \frac{3 \cdot 4^{p-2} a^{3p} \Delta + 9 \cdot 4^{p-2} a^{3p} \Delta + 4^{p-2} a^{3p} \Delta}{\epsilon^p} \\
&= \frac{(3 + 9 + 1) \cdot 4^{p-2} a^{3p} \Delta}{\epsilon^p} \\
&\leq \frac{4^p a^{3p} \Delta}{\epsilon^p} \\
&= (4a^3/\epsilon)^p \Delta \leq n^{1/p} \Delta .
\end{aligned}
$$

The last inequality follows from Lemma 3.5.4. □

Lemma 3.5.7. $a r_{p-1} + b \leq n^{1/p} \Delta$.

Proof. By the definitions of r_{p-1} and γ_0 we have $r_{p-1} = \gamma_0/\epsilon$. Since $\gamma_0 \leq \gamma$ and $a\gamma + b \leq \epsilon n^{1/p}\Delta$ by Lemma 3.5.6, we have

$$ar_{p-1} + b = a\frac{\gamma_0}{\epsilon} + b \leq \frac{a\gamma_0 + b}{\epsilon} \leq \frac{a\gamma + b}{\epsilon} \leq n^{1/p}\Delta\,. \qquad \square$$

Lemma 3.5.8. *For all* $0 \leq i < j \leq p-1$, $s(r_i, j-i) \leq s(r_{j-1}, 1)$

Proof. Fix some $0 \leq i \leq p-2$. The proof is by induction on j. In the base case $j = i+1$ the claim holds trivially. Consider now the induction step where we assume that the inequality holds for $j \geq i+1$ and have to show that it also holds for $j+1$. First, observe that

$$r_j = \frac{(\alpha+1+\epsilon)\sum_{0\leq j'\leq j-1} w_{j'} + \beta}{\epsilon} \geq \frac{(\alpha+1+\epsilon)w_{j-1} + \beta}{\epsilon} = \frac{(\alpha+1+\epsilon)(\alpha s_{j-1} + \beta) + \beta}{\epsilon}$$

and thus

$$s(r_j, 1) = ar_j + b \geq \frac{a(\alpha+1+\epsilon)(\alpha s_{j-1} + \beta) + \beta}{\epsilon} + b = \frac{a(\alpha+1+\epsilon)(\alpha s(r_{j-1}, 1) + \beta) + \beta}{\epsilon} + b$$

By the Property **B2** we have

$$s(r_i, j-1) \leq \frac{a(\alpha+1+\epsilon)(\alpha s(r_i, j-i-1) + \beta) + \beta}{\epsilon} + b$$

and by the induction hypothesis we have $s(r_i, j-i-1) \leq s(r_j, j-1)$. Therefore it follows that

$$s(r_i, j-1) \leq \frac{a(\alpha+1+\epsilon)(\alpha s(r_j, j-1) + \beta) + \beta}{\epsilon} + b \leq s(r_j, 1)\,. \qquad \square$$

3.5.4 Analysis of Approximation Guarantee

We now analyze the approximation error of a monotone ES-tree maintained on H''. This approximation error consists of two parts. The first part is an approximation error that comes from the fact that the monotone ES-tree only considers paths from s with a relatively small number of edges and therefore has to use edges from the hop set F. The second part is the approximation error we get from rounding the edge weights. We first give a formula for the approximation error that depends on the priority of the nodes and their distance to the root of the monotone ES-tree.

Before we give the proof we review a few properties of the monotone ES-tree (see Chapter 2 for the full algorithm). Similar to the classic ES-tree, the monotone ES-tree with root s maintains a level $\ell(v)$ for every node v. The monotone ES-tree

is initialized by computing a shortest paths tree up to depth L from s in H'' and thus, initially, $\ell(v) = d_{H''}(s, v)$. A single deletion or edge weight increase in G might result in a sequence of deletions, weight increases and insertions in F, and thus H''. The monotone ES-tree first processes the insertions and then the deletions and edge weight increases. It handles deletions and edge weights increases in the same way as the classic ES-tree. The procedure for handling the insertion of an edge (u, v) is trivial: it only stores the new edge and in particular does *not* change $\ell(u)$ or $\ell(v)$. Once the level $\ell(u)$ of a node u exceeds the maximum level L, we set $\ell(u) = \infty$. The pseudocode of the monotone ES-tree can be found in Algorithm 2.2 in Chapter 2.

For the analysis of the monotone ES-tree we will use the following notions, just like in Chapter 2. We say that an edge (u, v) is *stretched* if $\ell(u) > \ell(v) + w_{H''}(u, v)$. We say that a node u is *stretched* if it is incident to an edge (u, v) that is stretched. Note that for a node u that is not stretched we have $\ell(u) \leq \ell(v) + w_{H''}(u, v)$ for every edge (u, v) contained in H''. In our proof we will use the following properties of the monotone ES-tree from Chapter 2.

Observation 3.5.9. *The following holds for the monotone ES-tree:*

(1) The level of a node never decreases.

(2) An edge can only become stretched when it is inserted.

(3) As long as a node x is stretched, its level does not change.

(4) For every tree edge (u, v) (where v is the parent of u), $\ell(u) \geq \ell(v) + w(u, v)$.

A second prerequisite from Chapter 2 tells us when we may apply a variant of the triangle inequality to argue about the levels of nodes.

Lemma 3.5.10. *Let (u, v) be an edge of H'' such that $\ell(v) + w_{H''}(u, v) \leq L$. If (u, v) is not stretched and after the previous update in G the level of u was less than ∞, then for the current level of u we have $\ell(u) \leq \ell(v) + w_{H''}(u, v)$.*

Note that condition (2) simply captures the property of the monotone ES-tree that once the level of a node exceeds L it is set to ∞ and will never be decreased anymore. At the initialization (i.e., before the first update in H''), conditions (1) and (2) are fulfilled automatically.

To count the additive error from rounding the edge weights, we define, for every node u and every $0 \leq i \leq p - 1$, the function $h(u, i)$ as follows:

$$h(u, i) = \begin{cases} 0 & \text{if } u = s \\ (p + 1)\left\lceil \frac{\max(d_G(u,s) - r_i, 0)}{\Delta} \right\rceil + p + 1 - i & \text{otherwise} \end{cases}.$$

The intuition is that $h(u, i)$ bounds the number of hops from u to s, i.e., the number of edges required to go from u to s while at the same time providing the desired approximation guarantee. The approximation guarantee can now formally be stated as follows

Lemma 3.5.11 (Approximation Guarantee)**.** *For every node u of priority i with $d_G(u,s) \le D + \sum_{0\le i'\le i-1} s_{i'}$ we have*

$$\delta(s,u) \le (\alpha+\epsilon)d_G(u,s) + \gamma_i + h(u,i)\cdot\varphi\,.$$

Once we have proved this lemma, the desired bound on the approximation error (Property **A2**) follows easily because $h(u,i)\cdot\varphi \le \epsilon d_G(u,v) + 2\epsilon\Delta$ (as we show below) and thus

$$\begin{aligned}\delta(s,u) &\le (\alpha+\epsilon)d_G(u,s) + \gamma_i + h(u,i)\cdot\varphi\\ &\le (\alpha+\epsilon)d_G(u,s) + \gamma_0 + h(u,i)\cdot\varphi\\ &\le (\alpha+\epsilon)d_G(u,s) + \gamma_0 + \epsilon d_G(u,v) + 2\epsilon\Delta\\ &= (\alpha+2\epsilon)d_G(u,s) + \gamma\,.\end{aligned}$$

Lemma 3.5.12. *For every node u and every $0 \le i \le p-1$,*

$$h(u,i)\cdot\varphi \le \epsilon d_G(u,s) + 2\epsilon\Delta$$

Proof.

$$\begin{aligned}h(u,i)\cdot\varphi &= \left((p+1)\left\lceil\frac{\max(d_G(u,s)-r_i,0)}{\Delta}\right\rceil + p + 1 - i\right)\cdot\varphi\\ &\le \left((p+1)\left\lceil\frac{d_G(u,s)}{\Delta}\right\rceil + p + 1\right)\cdot\varphi\\ &\le \left((p+1)\left(\frac{d_G(u,s)}{\Delta}+1\right) + p + 1\right)\cdot\varphi\\ &= \left(\frac{(p+1)d_G(u,s)}{\Delta} + 2(p+1)\right)\cdot\varphi\\ &= \left(\frac{(p+1)d_G(u,s)}{\Delta} + 2(p+1)\right)\cdot\frac{\epsilon\Delta}{p+1}\\ &= \epsilon d_G(u,s) + 2\epsilon\Delta\,.\end{aligned}$$

□

Proof of Lemma 3.5.11. The proof is by double induction first on the number of updates in G and second on $h(u,i)$. Let u be a node of priority i such that $d_G(u,s) \le D + \sum_{0\le i'\le i-1} s_{i'}$. Remember that $\delta(u,s) = \ell(u)\cdot\varphi$, where $\ell(u)$ is the level of u in the monotone ES-tree of s. We know that after the last previous in G the distance estimate gave an approximation of the true distance in G. Since distances in G are non-decreasing it must have been the case that the level of u was less than ∞ after the previous.

If $u = s$, the claim is trivially true because $\ell(s) = 0$. Assume that $u \ne s$. If u is stretched in the monotone ES-tree, then the level of u has not changed since the previous deletion in G and thus the claim is true by induction. If u is not stretched, then $\ell(u) \le \ell(v) + w_{H''}(u,v)$ for every edge (u,v) in H''. Define the nodes v and x as follows. If $d_G(u,s) \le r_i$, then $v = s$. If $d_G(u,s) > r_i$, then consider a shortest path π

from u to s in G and let v be the furthest node from u on π such that $d_G(u,v) \leq r_i$ (which implies $d_G(v,s) \geq d_G(u,s) - r_i$). Furthermore let x be the neighbor of v on the shortest path π that is closer to s than v. Note that $d_G(u,x) \geq r_i$ (and thus $d_G(x,s) \leq d_G(u,s) - r_i$) and in particular G contains the edge (v,x). Note that (v,x) is also contained in H (and thus in H' and H'') because for $d_G(u,s) \leq D + \sum_{0 \leq i' \leq i-1} s_{i'}$ to hold it has to be the case that $w_G(v,x) \leq D + \sum_{0 \leq i' \leq i-1} s_{i'}$. Note that $\sum_{0 \leq i' \leq i-1} s_{i'} \leq \sum_{0 \leq i' \leq i-1} w_{i'} \leq r_{p-1} \leq n^{1/p}\Delta$ by Lemma 3.5.7. Thus, $w_G(v,x) \leq D + n^{1/p}\Delta$, which by the definition of H means that the edge (v,x) is contained in H.

Note that $s(d_G(u,v), p-1-i) \leq s(r_i, p-1-i)$ since the function $s(\cdot,\cdot)$ is non-decreasing. By Lemma 3.5.8 we have $s(r_i, p-1-i) \leq s(r_{p-2}, 1)$ and by the definition of $s(\cdot,1)$ and Lemma 3.5.7 we have $s(r_{p-2},1) = ar_{p-2} + b \leq ar_{p-1} + b \leq n^{1/p}\Delta \leq \hat{D}$. Thus, by Property **B2** we know that either $v \in B(u)$ or there is a node v' of priority $j' > i$ such that $d_G(u,v') \leq s(d_G(u,v), j-i)$. Note that in the first case the set of edges F contains the edge (u,v) and in the second case it contains the edge (u,v').

Case 1: $v \in B(u)$

If $v \in B(u)$, then F contains an edge (u,v) such that

$$w_F(u,v) = \hat{\delta}(u,v) \leq \alpha d_G(u,v) + \beta \tag{3.1}$$

Since $d_G(u,v) \leq r_i$ we have $w_F(u,v) \leq \alpha r_i + \beta \leq \alpha r_{p-1} + \beta \leq n^{1/p}\Delta$, where the last inequality holds by Lemma 3.5.7. Thus, (u,v) is contained in H and thus also in H' and H''.

If $d_G(u,s) \leq r_i$, then we have $v = s$. First observe that by the definition of H'' we have $w_{H''}(u,s) = w_{H'}(u,s)/\varphi$. Furthermore the rounding of the edge weights in H' guarantees that $w_{H'}(u,s) \leq w_H(u,s) + \varphi$. We therefore get

$$\begin{aligned}
w_{H''}(u,s) = &\leq \frac{w_F(u,s) + \varphi}{\varphi} \\
&\leq \frac{\alpha d_G(u,s) + \beta + \varphi}{\varphi} \\
&\leq \frac{\alpha\left(D + \sum_{0 \leq i' \leq i-1} s_{i'}\right) + \beta + \varphi}{\varphi} \\
&\leq \frac{\alpha D + (\alpha + 1 + \epsilon)\sum_{0 \leq i' \leq p-2} w_{i'} + \beta + \varphi}{\varphi} \\
&= \frac{\alpha D + \gamma_0 + \varphi}{\varphi} \\
&= \frac{\alpha D + \gamma_0 + \frac{\epsilon\Delta}{p+1}}{\varphi} \\
&\leq \frac{\alpha D + \gamma_0 + 2\epsilon\Delta}{\varphi} \\
&= \frac{\alpha D + \gamma}{\varphi} \leq \frac{\alpha D + \epsilon n^{1/p}\Delta}{\varphi} \leq \frac{(\alpha + 2\epsilon)D}{\varphi} + (p+1)n^{1/p} = L\,.
\end{aligned}$$

Here we have used the inequality $\gamma \le \epsilon n^{1/p}\Delta$ from Lemma 3.5.6. Since the maximum level in the monotone ES-tree is L and u is not stretched, it follows from Lemma 3.5.10 that $\ell(u) \le \ell(s) + w_{H''}(u,s) = w_{H''}(u,s)$. Together with the observation that $h(u,i) \ge 1$ since $u \ne s$ and $\beta \le \gamma_0$ we therefore get

$$\begin{aligned}\delta(s,u) = \ell(u)\cdot\varphi &\le w_{H''}(u,s)\cdot\varphi \le \alpha d_G(u,s) + \beta + \varphi \\ &\le \alpha d_G(u,s) + \beta + h(u,i)\cdot\varphi \le (\alpha+\epsilon)d_G(u,s) + \gamma_0 + h(u,i)\cdot\varphi\,.\end{aligned}$$

Consider now the case $d_G(u,s) > r_i$. Let j denote the priority of x. We first prove the following inequality, which will allow us among other things to use the induction hypothesis on x.

Claim 3.5.13. *If $d_G(u,s) > r_i$, then $h(x,j)+2 \le h(u,i)$.*

Proof. Remember that $i \le p-1$. The assumption $d_G(u,s) > r_i$ implies that $d_G(x,s) \le d_G(u,s) - r_i$. If $d_G(x,s) < r_j$, we have

$$\begin{aligned}h(x,j)+2 \le p+1-j+2 \le p+1+2 &\le p+1+p+1-i \\ &\le (p+1)\left\lceil\frac{d_G(u,s)-r_i}{\Delta}\right\rceil + p + 1 - i = h(u,i)\,.\end{aligned}$$

Here we use the inequality $\lceil (d_G(u,s)-r_j)/\Delta\rceil \ge 1$ which follows from the assumption $d_G(u,s) > r_i$.

If $d_G(x,s) \ge r_j$, then, using $r_j \ge r_0 \ge \Delta$, we get

$$\begin{aligned}h(x,j)+2 &= (p+1)\left\lceil\frac{d_G(x,s)-r_j}{\Delta}\right\rceil + p + 1 - j + 2 \\ &\le (p+1)\left\lceil\frac{d_G(x,s)-\Delta}{\Delta}\right\rceil + p + 1 + 2 \\ &= (p+1)\left\lceil\frac{d_G(x,s)}{\Delta} - 1\right\rceil + p + 1 + 2 \\ &= (p+1)\left(\left\lceil\frac{d_G(x,s)}{\Delta}\right\rceil - 1\right) + p + 1 + 2 \\ &= (p+1)\left\lceil\frac{d_G(x,s)}{\Delta}\right\rceil + 2 \\ &\le (p+1)\left\lceil\frac{d_G(x,s)}{\Delta}\right\rceil + p + 1 - i \\ &\le (p+1)\left\lceil\frac{d_G(u,s)-r_i}{\Delta}\right\rceil + p + 1 - i \\ &\le (p+1)\left\lceil\frac{\max(d_G(u,s)-r_i,0)}{\Delta}\right\rceil + p + 1 - i = h(u,i)\,.\end{aligned}$$

Here the last inequality follows from $d_G(u,s) - r_i \le \max(d_G(u,s)-r_i, 0)$, a trivial observation. □

Having proved this claim, we go on with the proof of the lemma. We will now show that

$$\ell(x) + w_{H''}(v,x) + w_{H''}(u,v) \leq \frac{(\alpha+\epsilon)d_G(u,s) + \gamma_i + h(u,i)\cdot\varphi}{\varphi} \tag{3.2}$$

as follows. If $d_G(u,s) > r_i$, then we have $d_G(u,x) \geq r_i$ by the choice of x. Remember that the edge (v,x) lies on a shortest path from u to s in G. It is therefore contained in G since before the first deletion and thus will never be stretched. We also may apply the induction hypothesis on x since

$$d_G(x,s) = d_G(u,s) - d_G(u,x) \leq d_G(u,s) - r_i \leq D + \sum_{0\leq i'\leq i-1} s_{i'} - r_i \leq D$$

due to $\sum_{0\leq i'\leq i-1} s_{i'} \leq r_i$ by the definition of r_i. Therefore we get

$$\begin{aligned}
&(\ell(x) + w_{H''}(v,x) + w_{H''}(u,v))\cdot\varphi \\
&\quad\leq \delta(s,x) + w_{H''}(v,x)\cdot\varphi + w_{H''}(u,v)\cdot\varphi && \text{(definition of } \delta(s,x)\text{)} \\
&\quad= \delta(s,x) + w_{H'}(v,x) + w_{H'}(u,v) && \text{(definition of } H'\text{)} \\
&\quad\leq \delta(s,x) + w_H(v,x) + \varphi + w_H(u,v) + \varphi && \text{(property of } w_{H'}\text{)} \\
&\quad\leq \delta(s,x) + w_G(v,x) + \varphi + w_F(u,v) + \varphi && ((v,x)\in E \text{ and } (u,v)\in F) \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \gamma_j + h(x,j)\cdot\varphi + w_G(v,x) + \varphi + w_F(u,v) + \varphi && \text{(induction hypothesis)} \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \gamma_j + w_F(u,v) + w_G(v,x) + (h(x,j)+2)\cdot\varphi && \text{(rearranging terms)} \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \gamma_j + w_F(u,v) + w_G(v,x) + h(u,i)\cdot\varphi && \text{(Claim 3.5.13)} \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \gamma_0 + w_F(u,v) + w_G(v,x) + h(u,i)\cdot\varphi && (\gamma_j \leq \gamma_0) \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \gamma_0 + \alpha d_G(u,v) + \beta + w_G(v,x) + h(u,i)\cdot\varphi && \text{(by Inequality (3.1))} \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \gamma_0 + \alpha d_G(u,v) + \beta + d_G(v,x) + h(u,i)\cdot\varphi && ((v,x) \text{ on shortest path}) \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \gamma_0 + \alpha d_G(u,v) + \beta + \alpha d_G(v,x) + h(u,i)\cdot\varphi && (\alpha \geq 1) \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \alpha(d_G(u,v) + d_G(v,x)) + \beta + \gamma_0 + h(u,i)\cdot\varphi && \text{(rearranging terms)} \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \alpha d_G(u,x) + \beta + \gamma_0 + h(u,i)\cdot\varphi && (v \text{ on shortest path}) \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \alpha d_G(u,x) + \beta + \gamma_0 - \gamma_i + \gamma_i + h(u,i)\cdot\varphi && \text{(rearranging terms)} \\
&\quad= (\alpha+\epsilon)d_G(x,s) + \alpha d_G(u,x) + \epsilon r_i + \gamma_i + h(u,i)\cdot\varphi && \text{(by Lemma 3.5.3)} \\
&\quad\leq (\alpha+\epsilon)d_G(x,s) + \alpha d_G(u,x) + \epsilon d_G(u,x) + \gamma_i + h(u,i)\cdot\varphi && (d_G(u,x) \geq r_i) \\
&\quad= (\alpha+\epsilon)(d_G(u,x) + d_G(x,s)) + \gamma_i + h(u,i)\cdot\varphi && \text{(rearranging terms)} \\
&\quad= (\alpha+\epsilon)d_G(u,s) + \gamma_i + h(u,i)\cdot\varphi && (x \text{ on shortest path})\,.
\end{aligned}$$

By Lemma 3.5.12 we have $h(u,i)\cdot\varphi \leq \epsilon d_G(u,s) + 2\epsilon\Delta$ and thus Inequality (3.2)

implies that

$$\begin{aligned}
\ell(x) + w_{H''}(v,x) + w_{H''}(u,v) &\leq \frac{(\alpha+2\epsilon)d_G(u,s) + \gamma_i + 2\epsilon\Delta}{\varphi} \\
&\leq \frac{(\alpha+2\epsilon)\left(D + \sum_{0\leq i'\leq i-1} s_{i'}\right) + \gamma_i + 2\epsilon\Delta}{\varphi} \\
&\leq \frac{(\alpha+2\epsilon)D + (\alpha+1+\epsilon)\left(\sum_{0\leq i'\leq i-1} w_{i'}\right) + \gamma_i + 2\epsilon\Delta}{\varphi} \\
&\leq \frac{(\alpha+2\epsilon)D + \gamma_i + 2\epsilon\Delta}{\varphi} \\
&= \frac{(\alpha+2\epsilon)D + \gamma}{\varphi} \\
&\leq \frac{(\alpha+2\epsilon)D}{\varphi} + (p+1)n^{1/p} = L\,.
\end{aligned}$$

As the maximum level in the monotone ES-tree is L and the edge (v,x) is not stretched, it follows from Lemma 3.5.10 that $\ell(v) \leq \ell(x) + w_{H''}(v,x)$ and since u is not stretched, we have

$$\ell(u) \leq \ell(v) + w_{H''}(u,v) \leq \ell(x) + w_{H''}(v,x) + w_{H''}(u,v)\,.$$

and thus

$$\delta(s,u) = \ell(u)\cdot\varphi \leq (\ell(x) + w_{H''}(v,x) + w_{H''}(u,v))\cdot\varphi \leq (\alpha+\epsilon)d_G(u,s) + \gamma_i + h(u,i)\cdot\varphi$$

Case 2: $v \notin B(u)$

By Property **B2** we know that there is some node v' of priority $j' > i$ such that $u \in B(v')$ and $d_G(u,v') \leq s(d_G(u,v), j'-i)$. By Lemma 3.5.8 we therefore have

$$d_G(u,v') \leq s(r_i, j'-i) \leq s(r_{j'-1}, j'-1) = s_{j'-1}\,.$$

From the definition of F and Property **B1** it now follows that F contains the edge (u,v') of weight

$$d_G(u,v') \leq w_F(u,v') = \hat{\delta}(u,v') \leq \alpha d_G(u,v') + \beta \leq \alpha s_{j'-1} + \beta = w_{j'-1}$$

Since $j' \leq p-1$ we have $w_{j'-1} \leq w_{p-2} \leq r_{p-1}$. As $r_{p-1} \leq n^{1/p}\Delta$, by Lemma 3.5.7, we conclude that the edge (u,v') is contained H and thus also in H' and H''.

We first prove the following inequality, which will allow us among other things to use the induction hypothesis on x.

Claim 3.5.14. $h(v',j') + 1 \leq h(u,i)$

Proof. Remember that $j' \geq i+1$. If $d_G(v',s) < r_{j'}$, we get

$$h(v',j') + 1 \leq p + 1 - j' + 1 \leq p + 1 - i \leq h(u,i)\,.$$

If $d_G(v', s) \geq r_{j'}$, then we use the inequality $r_i + s_{j'-1} \leq r_{j'}$ (which easily follows from the definition of $r_{j'}$) and get

$$\begin{aligned}
h(v', j') + 1 &= (p+1)\left\lceil \frac{d_G(v', s) - r_{j'}}{\Delta} \right\rceil + p + 1 - j' + 1 \\
&\leq (p+1)\left\lceil \frac{d_G(v', s) - r_{j'}}{\Delta} \right\rceil + p + 1 - i - 1 + 1 \\
&\leq (p+1)\left\lceil \frac{d_G(u, s) + d_G(v', u) - r_{j'}}{\Delta} \right\rceil + p + 1 - i \\
&\leq (p+1)\left\lceil \frac{d_G(u, s) + s_{j'-1} - r_{j'}}{\Delta} \right\rceil + p + 1 - i \\
&\leq (p+1)\left\lceil \frac{d_G(u, s) - r_i}{\Delta} \right\rceil + p - i \\
&\leq (p+1)\left\lceil \frac{\max(d_G(u, s) - r_i, 0)}{\Delta} \right\rceil + p + 1 - i = h(u, i)\,.
\end{aligned}$$

□

Having proved this claim, we go on with the proof of the lemma. Note that we may apply the induction hypothesis on v' because by the triangle inequality we have

$$\begin{aligned}
d_G(v', s) \leq d_G(u, s) + d_G(v', u) &\leq D + \sum_{0 \leq i' \leq i-1} s_{i'} + d_G(v', u) \\
&\leq D + \sum_{0 \leq i' \leq i-1} s_{i'} + s_{j'-1} \leq D + \sum_{0 \leq i' \leq j'-1} s_{i'}\,.
\end{aligned}$$

We will now show that

$$\ell(v') + w_{H''}(u, v') \leq \frac{(\alpha + \epsilon) d_G(u, s) + \gamma_i + h(u, i) \cdot \varphi}{\varphi} \tag{3.3}$$

as follows:

$$\begin{aligned}
&(\ell(v') + w_{H''}(u, v')) \cdot \varphi && (u \text{ not stretched}) \\
&\quad = \delta(v', s) + w_{H''}(u, v') \cdot \varphi && (\text{definition of } \delta(v', s)) \\
&\quad = \delta(v', s) + w_{H'}(u, v') && (\text{definition of } H'') \\
&\quad \leq \delta(v', s) + w_H(u, v') + \varphi && (\text{property of } w_{H'}(u, v')) \\
&\quad \leq \delta(v', s) + w_F(u, v') + \varphi && (\text{definition of } H) \\
&\quad \leq (\alpha + \epsilon) d_G(v', s) + \gamma_{j'} + h(v', j') \cdot \varphi + w_F(u, v') + \varphi && (\text{induction hypothesis}) \\
&\quad = (\alpha + \epsilon) d_G(v', s) + \gamma_{j'} + w_F(u, v') + (h(v', j') + 1) \cdot \varphi && (\text{rearranging terms}) \\
&\quad \leq (\alpha + \epsilon) d_G(v', s) + \gamma_{j'} + w_F(u, v') + h(u, i) \cdot \varphi && (\text{Claim 3.5.14}) \\
&\quad \leq (\alpha + \epsilon)(d_G(v', u) + d_G(u, s)) + \gamma_{j'} + w_F(u, v') + h(u, i) \cdot \varphi && (\text{triangle inequality})
\end{aligned}$$

$$
\begin{aligned}
&\leq (\alpha+\epsilon)(w_F(u,v')+d_G(u,s))+\gamma_{j'}+w_F(u,v')+h(u,i)\cdot\varphi && \text{(by Inequality 3.5.4)}\\
&= (\alpha+\epsilon)d_G(u,s)+\gamma_{j'}+(\alpha+\epsilon+1)w_F(u,v')+h(u,i)\cdot\varphi && \text{(rearranging terms)}\\
&\leq (\alpha+\epsilon)d_G(u,s)+\gamma_{j'}+(\alpha+\epsilon+1)w_{j'-1}+h(u,i)\cdot\varphi && \text{(by Inequality 3.5.4)}\\
&= (\alpha+\epsilon)d_G(u,s)+\gamma_{j'-1}+h(u,i)\cdot\varphi && \text{(definition of } \gamma_{j'-1})\\
&\leq (\alpha+\epsilon)d_G(u,s)+\gamma_i+h(u,i)\cdot\varphi && (\gamma_i \geq \gamma_{j'-1} \text{ as } j' \geq i+1)\,.
\end{aligned}
$$

By Lemma 3.5.12 we have $h(u,i)\cdot\varphi \leq \epsilon d_G(u,s)+2\epsilon\Delta$ and thus Inequality (3.3) implies that

$$\ell(v')+w_{H''}(u,v') \leq \frac{(\alpha+2\epsilon)d_G(u,s)+\gamma_i+2\epsilon\Delta}{\varphi} \leq \frac{(\alpha+2\epsilon)D}{\varphi}+(p+1)n^{1/p}=L\,.$$

As the maximum level in the monotone ES-tree is L and u is not stretched, it follows from Lemma 3.5.10 that $\ell(u) \leq \ell(v')+w_{H''}(u,v')$ and thus

$$\delta(s,u)=\ell(u)\cdot\varphi \leq (\ell(v')+w_{H''}(u,v'))\cdot\varphi \leq (\alpha+\epsilon)d_G(u,s)+\gamma_i+h(u,i)\cdot\varphi\,. \quad \square$$

3.6 Putting Everything Together

In the following we combine our results of Section 3.4 and Section 3.5 to obtain decremental algorithms for approximate SSSP and approximate APSP.

3.6.1 Approximate SSSP

We first show how to obtain an algorithm for approximate SSSP. First, we obtain an algorithm that provides approximate distance for all nodes that are at distance at most R from the source, where R is some range parameter. We use a hierarchical approach to obtain this algorithm: Given an algorithm for maintaining approximate shortest paths, we obtain an algorithm for maintaining approximate balls, which in turn gives us an algorithm for maintaining approximate shortest paths for a larger range of distances than the initial algorithm. This scheme is repeated several times and can be "started" with the (exact) ES-tree.

Lemma 3.6.1. *For every $R \geq n$ and every $0 < \epsilon \leq 1$, there is a decremental approximate SSSP algorithm that, given a fixed source node s, maintains, for every node v, a distance estimate $\delta(s,v)$ such that $\delta(s,v) \geq d_G(s,v)$ and if $d_G(s,v) \leq R$, then $\delta(s,v) \leq (1+\epsilon)d_G(s,v)$. It has a total update time of $\tilde{O}(m^{1+3(\log\log R)/q}R^{2/q})$, where*

$$q=\left\lfloor\sqrt{\left\lfloor\frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8\cdot 4^3\log n}{\epsilon}\right)}}\right\rfloor}\right\rfloor$$

and, after every update in G, returns, for every node v such that $\delta(s,v)$ has changed, v together with the new value of $\delta(s,v)$.

Proof. In the proof we will use the following values. We set $a = 4$,

$$p = \left\lfloor \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8a^3 \log n}{\epsilon}\right)}} \right\rfloor$$

and $q = \lfloor \sqrt{p} \rfloor$. Furthermore we set $\epsilon' = \epsilon/2(q-2)$ and for every $0 \le k \le q-2$ we set $\alpha_k = 1 + 2k\epsilon' \le 1 + \epsilon$, $\Delta_k = R^{k/p}$, and $D_k = R^{(k+2)/p}$.

The heart of our proof is the following claim which gives us decremental approximate SSSP algorithms for larger and larger depths, until finally the full range R is covered.

Claim 3.6.2. *For every $0 \le k \le q-2$, there is a decremental approximate SSSP algorithm ApproxSSSP$_k$ with the following properties:*

A1 *$\delta(s,v) \ge d_G(s,v)$*

A2 *If $d_G(s,v) \le D_k$, then $\delta(s,v) \le \alpha_k d_G(s,v)$.*

A3 *The total update time of ApproxSSSP$_k$ is*

$$T_k(m) = \tilde{O}(2^k m^{1+k/p} R^{2/q} (\log R)^k/\epsilon') .$$

A4 *After every update in G, ApproxSSSP$_k$ returns, for every node v such that $\delta(s,v)$ has changed, v together with the new value of $\delta(s,v)$.*

Proof. We prove the claim by induction on k. In the base case $k = 0$ we use the (exact) ES-tree, which for distances up to $D \le D_0$ has a total update time of $O(mD_0) = O(mR^{2/q})$ and thus has all claimed properties

We now consider the induction step. We apply Proposition 3.4.1 to obtain a decremental algorithm ApproxBalls$_k$ (with parameters $\hat{k} = p$ and $\hat{\epsilon} = 1$) that maintains for every node $u \in V$ a set of nodes $B_k(u)$ and a distance estimate $\hat{\delta}_k(u,v)$ for every node $v \in B(u)$ such that:

B1 For every node u and every node $v \in B_k(u)$ we have $d_G(u,v) \le \hat{\delta}_k(u,v) \le \alpha_{k-1} d_G(u,v)$.

B2 Let $s(x,l) = a(a+1)^l x$. For every node u of priority $i \le k-1$ and every node v such that $s(d_G(u,v), p-1-i) \le D_k$ either $v \in B(u)$ or there is some node v' of priority $j > i$ such that $d_G(u,v') \le s(d_G(u,v), j-i)$.

B3 In expectation, $\sum_{u \in V} |\mathscr{B}_k(u)| = \tilde{O}(m^{1+1/k} \log D_k)$, where $\mathscr{B}_k(u)$ denotes the set of nodes ever contained in $B_k(u)$.

B4 The update time of ApproxBalls$_k$ is

$$t_k(m) = \tilde{O}\left(m^{1+1/p} \log D_k + \sum_{0 \le i \le p-1} m^{1-i/p} \cdot T_{k-1}(m^{(i+1)/p}) \log D_k + T(m) \right) .$$

Note that $D_k \leq R$ and thus $\log D_k \leq \log R$ and remember that by the induction hypothesis we have

$$T_{k-1}(m) = \tilde{O}(2^{k-1} m^{1+(k-1)/p} R^{2/q} (\log R)^{k-1}/\epsilon') .$$

We now analyze $m^{1-i/p} \cdot T_{k-1}(m^{(i+1)/p})$ for each $0 \leq i \leq p-1$. The algorithm ApproxSSSP$_{k-1}$ is run on a graph with $m_i = m^{(i+1)/p}$ edges and $n_i = n$ nodes. Using the parameter $p_i = p$, it has a total update time of $\tilde{O}(m_i^{1+1/p_i}) = \tilde{O}(m_i^{1+1/p})$. Furthermore, we have

$$1-i/p+((i+1)/p)\cdot(1+(k-1)/p) = 1+1/p+((i+1)/p)((k-1)/p) \leq 1+1/p+(k-1)/p = 1+k/p$$

Thus, $m^{1-i/p} \cdot T_{k-1}(m^{(i+1)/p}) = \tilde{O}(2^{k-1} m^{1+k/p} R^{2/q} (\log R)^{k-1}/\epsilon^k)$ and since $k \geq 1$ it follows that

$$t_k(m) = \tilde{O}(2^k m^{1+k/p} R^{2/q} (\log R)^k/\epsilon') .$$

We now want to argue that we may apply Proposition 3.5.1 to obtain an approximate decremental SSSP algorithm ApproxSSSP$'_k$ (with parameters p, Δ_k D_k, and ϵ'). We first show that

$$p \leq \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}} ,$$

First note that $q \leq \log n$ and thus $\epsilon' = \epsilon(2(q-2) \geq \epsilon/(2q) \geq \epsilon/(2\log n)$. It follows that

$$\frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{4a^3}{\epsilon}\right)}} \geq \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8a^3 \log n}{\epsilon}\right)}} \geq \left\lfloor \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8\cdot 4^3 \log n}{\epsilon}\right)}} \right\rfloor = p .$$

Note also that for all $x \geq 1$ and $l \geq 1$ we have

$$s_k(x, l+1) = (a+1)s_k(x,l) \geq 2as_k(x,l) \geq a(\alpha_{k-1} + 1 + \epsilon')\alpha_{k-1} s(x,l)/\epsilon' .$$

We therefore may apply Proposition 3.5.1 to obtain an approximate decremental SSSP algorithm ApproxSSSP$'_k$ (with parameters p, D_k, and ϵ') that maintains, for every node $v \in V$, a distance estimate $\delta'(s,v)$ such that:

A1' $\delta'(s,v) \geq d_G(s,v)$

A2' If $d_G(s,v) \leq D_k$, then $\delta'(s,v) \leq (\alpha_k + \epsilon')d_G(s,v) + \epsilon' n^{1/p}\Delta_k$

A3' The total update time of ApproxSSSP$'_k$ is

$$T'_k(m) = \tilde{O}((\alpha_k D_k/\Delta_k + n^{1/p}) \sum_{u\in V} (m + |\mathscr{B}(u)|)/\epsilon' + t_k(m,n,p,n^{1/p}\Delta)) .$$

A4' After every update in G, ApproxSSSP$'_k$ returns, for every node v such that $\delta(s,v)$ has changed, v together with the new value of $\delta(s,v)$.

Its total update time is

$$T_k(m) = \tilde{O}((\alpha_k D_k/\Delta_k + n^{1/p}) \sum_{u \in V} (m + |\mathscr{B}_k(u)|)/\epsilon' + t_k(m, n, n^{1/p}\Delta_k))$$

Note that $\alpha_k \leq 1 + \epsilon \leq 2$ and $D_k/\Delta_k = R^{2/q}$. Since $q \leq p$ and $R \geq n$ we have $n^{1/p} \leq R^{2/q}$. We also have $\sum_{u \in V} |\mathscr{B}_k(u)| = \tilde{O}(m^{1+1/p} \log R)$. Therefore the total update time of ApproxSSSP$'_k$ is

$$T'_k(m) = \tilde{O}(m^{1+1/p} R^{2/q} \log R/\epsilon' + 2^k m^{1+k/p} R^{2/q} (\log R)^k/\epsilon')$$

Since $k \geq 1$ it follows that

$$T_k(m) = \tilde{O}(2^k m^{1+k/p} R^{2/q} (\log R)^k/\epsilon') .$$

Let ApproxSSSP$_k$ denote the algorithm that internally runs ApproxSSSP$'_k$ and ApproxSSSP$_{k-1}$ and additionally maintains, for every node v, the value $\delta_k(s, v) = \min(\delta'_k(s, v), \delta_{k-1}(s, v))$. Since both ApproxSSSP$'_k$ and ApproxSSSP$_{k-1}$ return, after each update in G, every node v for which $\delta(s, v)$ has changed, and the minimum can be computed in constant time, ApproxSSSP$_k$ has the same asymptotic total update time as ApproxSSSP$'_k$. It remains to show that $\delta_k(s, v)$ fulfills the desired approximation guarantee for every node v. Since both $\delta'_k(s, v) \geq d_G(s, v)$ and $\delta_{k-1}(s, v) \geq d_G(s, v)$ also $\delta_k(s, v) \geq d_G(s, v)$. Furthermore, we know that if $d_G(s, v) \leq D_k$, then $\delta'_k(s, v) \leq \epsilon n^{1/p} \Delta_k$. Let v be a node such that $d_G(s, v) \leq D_k$. If $d_G(s, v) \leq D_{k-1}$, then $\delta_k(s, v) \leq \delta_{k-1}(s, v) \leq \alpha_{k-1} d_G(s, v) \leq \alpha_k d_G(u, v)$. If $d_G(s, v) \geq D_{k-1}$, then

$$\begin{aligned} \delta_k(s, v) \leq \delta'_k(s, v) &\leq (\alpha_{k-1} + \epsilon') d_G(s, v) + \epsilon' n^{1/p} \Delta_k \\ &\leq (\alpha_{k-1} + \epsilon') d_G(s, v) + \epsilon' D_{k-1} \\ &\leq (\alpha_{k-1} + 2\epsilon') d_G(s, v) = \alpha_k d_G(s, v) . \end{aligned}$$

This finishes the proof of the claim. □

The lemma now follows from the claim by observing that ApproxSSSP$_{q-2}$ is the desired decremental approximate SSSP algorithm. The correctness simply follows from $D_{q-2} = R$. The total update time is

$$T_{q-2}(m) = \tilde{O}(2^{q-2} m^{1+(q-2)/p} R^{2/q} (\log R)^{q-2}/\epsilon') .$$

Remember that $q = \lfloor \sqrt{p} \rfloor$ and thus $(q-2)/p \leq q/p \leq 1/\sqrt{p} \leq 1/q$. By the definition of p we have $(2/\epsilon')^p \leq n^{1/p}$ and thus $(2/\epsilon')^q \leq (2/\epsilon')^p \leq n^{1/p} \leq n^{1/q}$ and furthermore $(\log R)^q \leq (\log R)^p = (2^p)^{\log\log R} \leq (n^{1/p})^{\log\log R} = n^{(\log\log R)/p} \leq n^{(\log\log R)/q}$. It follows that the total update time is

$$T_{q-2}(m) = \tilde{O}(m^{1+3(\log\log R)/q} R^{2/q}) .$$ □

We can turn the algorithm above into an algorithm for the full distance range by using the scaling technique once more.

Theorem 3.6.3. *For every $0 < \epsilon \leq 1$, there is a decremental approximate SSSP algorithm that, given a fixed source node s, maintains, for every node v, a distance estimate $\delta(s, v)$ such that $d_G(s, v) \leq \delta(s, v) \leq (1 + \epsilon)d_G(s, v)$. It has constant query time and a total update time of*

$$O(m^{1+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log W)\,.$$

If $1/\epsilon = O(\text{polylog}\, n)$, then the total update time is $O(m^{1+o(1)} \log W)$.

Proof. For every $0 \leq i \leq \lfloor \log(nW) \rfloor$ we define

$$\varphi_i = \frac{\epsilon 2^i}{n}\,.$$

Let G'_i be the graph that has the same nodes and edges as G and in which every edge weight is rounded to the next multiple of φ_i, i.e., every edge (u, v) in G'_i has weight

$$w_{G'_i}(u, v) = \left\lceil \frac{w_G(u, v)}{\varphi_i} \right\rceil \cdot \varphi_i$$

where $w_G(u, v)$ is the weight of (u, v) in G. This rounding guarantees that

$$w_G(u, v) \leq w_{G_i}(u, v) \leq w_G(u, v) + \varphi_i$$

for every edge (u, v) of G. Furthermore we define G''_i to be the graph that has the same nodes and edges as G'_i and in which every edge weight is scaled down by a factor of $1/\varphi_i$, i.e., every edge (u, v) in G''_i has weight

$$w_{G''_i}(u, v) = \frac{w_{G'_i}(u, v)}{\varphi_i} = \left\lceil \frac{w(u, v)}{\varphi_i} \right\rceil\,.$$

The algorithm is as follows: For every $0 \leq i \leq \lfloor \log(nW) \rfloor$ we use the algorithm of Lemma 3.6.1 on the graph G''_i with $R = 4n/\epsilon$ to maintain a distance estimate $\delta_i(s, v)$ for every node v that satisfies

- $\delta_i(s, v) \geq d_{G''_i}(s, v)$ and
- if $d_{G''_i}(s, v) \leq R$, then $\delta_i(s, v) \leq (1 + \epsilon)d_{G''_i}(s, v)$.

We let our algorithm return the distance estimate

$$\delta(s, v) = \min_{0 \leq i \leq \lfloor \log nW \rfloor} \varphi_i \delta_i(s, v)\,.$$

We now show that there is some $0 \leq i \leq \lfloor \log(nW) \rfloor$ such that $\varphi_i \delta_i(s, v) \leq (1 + 3\epsilon)d_G(s, v)$. As $\delta(s, v)$ is the minimum of all the distance estimates, this implies that $\delta(s, v) \leq (1+3\epsilon)d_G(s, v)$. In particular, we know that there is some $0 \leq i \leq \lfloor \log(nW) \rfloor$ such that $2^i \leq d_G(s, v) \leq 2^{i+1}$ since W is the maximum edge weight and all paths consist of at most n edges. Consider a shortest path π from v to s in G whose weight is equal to $d_G(s, v)$. Let $w_G(\pi)$ and $w_{G'_i}(\pi)$ denote the weight of the path π in G

and G_i', respectively. Since π consists of at most n edges we have $w_{G_i'}(\pi) \leq w(\pi) + n\varphi_i$. Therefore we get

$$d_{G_i'}(s,v) \leq w_{G_i'}(\pi) \leq w(\pi) + n\varphi_i = d_G(s,v) + \epsilon 2^i \leq d_G(s,v) + \epsilon d_G(s,v) = (1+\epsilon) d_G(s,v)\,.$$

Now observe the following:

$$d_{G_i''}(s,v) = \frac{d_{G_i'}(s,v)}{\varphi_i} \leq \frac{(1+\epsilon) d_G(s,v)}{\varphi_i} \leq \frac{2 d_G(s,v)}{\varphi_i} = \frac{2 d_G(s,v) n}{\epsilon 2^i}$$
$$\leq \frac{2 \cdot 2^{i+1} n}{\epsilon 2^i} = \frac{4n}{\epsilon} = R\,.$$

Since $d_{G_i''}(s,v) \leq R$ we get $\delta_i(s,v) \leq (1+\epsilon) d_{G_i''}(s,v)$ by Lemma 3.6.1. Thus, we get

$$\varphi_i \delta_i(s,v) \leq \varphi_i((1+\epsilon) d_{G_i''}(s,v)) = (1+\epsilon) d_{G_i'}(s,v) \leq (1+\epsilon)^2 d_G(s,v)$$
$$\leq (1+3\epsilon) d_G(s,v)$$

as desired.

We now analyze the running time of this algorithm. By Lemma 3.6.1, for every $0 \leq i \leq \lfloor \log(nW) \rfloor$, maintaining $\delta_i(s,v)$ on G_i'' for every node s takes time $\tilde{O}(m^{1+3(\log\log R)/q} R^{2/q})$, where

$$q = \left\lfloor \sqrt{\left\lfloor \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8 \cdot 4^3 \log n}{\epsilon}\right)}} \right\rfloor} \right\rfloor$$

By our choice of $R = 4n/\epsilon$, the total update time for maintaining all these $\lfloor \log(nW) \rfloor$ distance estimates is $\tilde{O}(m^{1+5(\log\log(4n/\epsilon))/q} \log W/\epsilon)$, where

$$q = \left\lfloor \sqrt{\left\lfloor \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{8 \cdot 4^3 \log n}{\epsilon}\right)}} \right\rfloor} \right\rfloor$$

To obtain a $(1+\epsilon)$ approximation (instead of a $(1+3\epsilon)$-approximation, we simply run the whole algorithm with $\epsilon' = \epsilon/3$. This results in a total update time of $\tilde{O}(m^{1+5(\log\log(12n/\epsilon))/q} \log W/\epsilon)$, where

$$q = \left\lfloor \sqrt{\left\lfloor \frac{\sqrt{\log n}}{\sqrt{\log\left(\frac{24 \cdot 4^3 \log n}{\epsilon}\right)}} \right\rfloor} \right\rfloor$$

Now observe that $1/\epsilon \leq n^{1/q}$ and that

$$\frac{5\left(\log\log\left(\frac{12n}{\epsilon}\right)\right)}{q} = O\left(\frac{\left(\log\log\left(\frac{n}{\epsilon}\right)\right)\left(\log\left(\frac{\log n}{\epsilon}\right)\right)^{1/4}}{(\log n)^{1/4}}\right) = O\left(\frac{\left(\log\left(\frac{\log n}{\epsilon}\right)\right)^{5/4}}{(\log n)^{1/4}}\right).$$

Since $\tilde{O}(1) = O(\text{polylog}\, n) = O(n^{O(\log^{5/4}((\log n)}))$ the total update time therefore is

$$O(m^{1+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log W).$$

If $1/\epsilon = O(\text{polylog}\, n)$, then the total update time is $O(m^{1+\log^{5/4}\log n/\log^{1/4} n} \log W)$, which is $O(m^{1+o(1)} \log W)$ since $\lim_{x\to\infty}(\log^{5/4}\log n/\log^{1/4} n) = 0$.

The query time of the algorithm described above is $O(\log(nW))$ as it has to compute $\delta(s,v) = \min_{0\leq i\leq\lfloor\log nW\rfloor} \delta_i(s,v)\cdot\varphi_i$ when asked for the approximate distance from v to s. We can reduce the query time to $O(1)$ by using a min-heap for every node v that stores $\delta_i(s,v)$ for all $0 \leq i \leq \lfloor\log(nW)\rfloor$. This allows us to query for $\delta(s,v)$ in constant time. □

3.6.2 Approximate APSP

We now show how to use our techniques to obtain a decremental approximate APSP algorithm. This is conceptually simple now. We simply use the approximate SSSP algorithm from Theorem 3.6.3 and plug it into the algorithm for maintaining approximate balls from Proposition 3.4.1. By using an adequate query procedure we can use the distance estimates maintained for the approximate balls to return the approximate distances between any two nodes.

Theorem 3.6.4. *There is a decremental approximate APSP algorithm that upon a query for the approximate between any pair of nodes u and v returns a distance estimate $\delta(u,v)$ such that $d_G(u,v) \leq \delta(u,v) \leq ((2+\epsilon)^k - 1)d_G(u,v)$. It has a query time of $O(k^k)$ and a total update time of*

$$O(m^{1+1/k+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log^2 W)$$

If $1/\epsilon = O(\text{polylog}\, n)$, then the total update time is $O(m^{1+o(1)}\log^2 W)$.

Proof. We use the approximate SSSP algorithm of Theorem 3.6.3 that provides a $(1+\epsilon)$-approximation and has a total update time of

$$T(m,n) = O(m^{1+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log W)$$

and if $1/\epsilon = O(\text{polylog}\, n)$, then the total update time is $T(m,n) = O(m^{1+o(1)}\log W)$. By Proposition 3.4.1 we can maintain approximate balls with a total update time of

$$t(m,n,k,\epsilon) = \tilde{O}\left(m^{1+1/k}\log D/\epsilon + \sum_{0\leq i\leq k-1} m^{1-i/k}\cdot T(m_i,n_i)\log D/\epsilon + T(m,n)\right),$$

where, for each $0 \le i \le k-1$, $m_i = m^{(i+1)/k}$ and $n_i = \min(m_i, n)$. Using similar arguments as above we get that $t(m, n, k, \epsilon) = O(m^{1+1/k+O(\log^{5/4}((\log n)/\epsilon)/\log^{1/4} n)} \log^2 W)$ and $O(m^{1+o(1)} \log^2 W)$ if $1/\epsilon = O(\text{polylog } n)$.

Additionally we maintain, for every node $v \in V$, the node $c_i(v)$ which is a node with minimum $\delta(u, v)$ among all nodes u of priority j such that $v \in B(u)$. This can be done as follows. For every node v we maintain a heap containing all nodes u of priority i such that $v \in B(u)$ using the key $\delta(u, v)$. Every time v joins or leaves $B(u)$ we insert or remove u from the heap of v. Every time $\delta(u, v)$ changes, we update the key of u in the heap of v. After each insert, remove, or update in the heap of some node v, we find the minimal element $c_i(v)$ of the heap. As each heap operation takes logarithmic time, the total update time of the algorithm of Proposition 3.4.1 only increases by a logarithmic factor.

Procedure 3.1: QUERY(u, v)

```
1  if v ∈ B(u) then
2  |  δ'(u, v) ← δ(u, v)
3  else
4  |  Set i to the priority of u
5  |  foreach j = i + 1 to k − 1 do
6  |  |  if c_j(u) exists then
7  |  |  |  v'' ← c_j(u)
8  |  |  |  δ'(v'', v) ← QUERY(v'', v)
9  |  |  |  δ'_j(u, v) ← δ(v, v'') + δ'(v'', v)
10 |  |  else
11 |  |  |  δ'_j(u, v) ← ∞
12 |  |  δ'(u, v) ← min_{i+1≤j≤k−1} δ'_j(u, v)
13 return δ'(u, v)
```

To answer a query for the approximate distance between a pair of nodes u and v we use Procedure 3.1. This procedure first tests whether $v \in B(u)$ and if yes returns $\delta(u, v)$. Otherwise it does the following for every $j \ge i + 1$, where i is the priority of u: It first computes the node $c_j(u)$, which among the nodes v' of priority j with $u \in B(v')$ is the one with the minimum value of $\delta(v', u)$. Then it recursively queries for the approximate distance $\delta'(c_j(u), v)$ from $c_j(u)$ to v and sets the distance estimate via $c_j(u)$ to $\delta'_j(u, v) = \delta(v, c_j(u)) + \delta'(c_j(u), v)$. Finally, it returns the minimum of all distance estimates $\delta'_j(u, v)$.

Note that in each instance there are $O(k)$ recursive calls and with each recursive call the priority of u increases by at least one. Thus the running time of the query procedure is $O(k^k)$.

Claim 3.6.5. *For every pair of nodes u and v the distance estimate $\delta'(u, v)$ computed by Procedure 3.1 satisfies $\delta'(u, v) \le (((1+\epsilon)^2 + 1)^{k-i} - 1) d_G(u, v)$, where i is the priority of u.*

Proof. The proof is by induction on the priority i of u. Let $\delta'(u,v)$ denote the distance estimate returned by Procedure 3.1. If $i = k-1$, then we know that $v \in B(u)$ and thus $\delta'(u,v) = \delta(u,v) \leq (1+\epsilon)d_G(u,v)$. If $i < k-1$ we distinguish between the two cases $v \in B(u)$ and $v \notin B(u)$. If $v \in B(u)$, then $\delta'(u,v) = \delta(u,v) \leq (1+\epsilon)d_G(u,v)$. If $v \notin B(u)$, then by Proposition 3.4.1 there is a node v' of priority $j > i$ such that $u \in B(v')$ and $d_G(u,v') \leq (1+\epsilon)^2((1+\epsilon)^2+1)^{j-i-1}d_G(u,v)$.

We will now argue that $\delta'_j(u,v) \leq 2((1+\epsilon)^3+1)^{k-1-i}-1)d_G(u,v)$, which implies the same upper bound for $\delta'(u,v)$. Set $v'' \leftarrow c_j(u)$. As both v'' and v' have priority j and $u \in B(v')$ as well as $v \in B(v'')$ we have $\delta(u,v'') \leq \delta(u,v')$ by the definition of v''. Since $\delta(u,v') \leq (1+\epsilon)d_G(u,v')$, we have

$$\begin{aligned}\delta(u,v'') &\leq (1+\epsilon)d_G(u,v') \leq (1+\epsilon)^3((1+\epsilon)^2+1)^{j-i-1}d_G(u,v)\\ &\leq (1+\epsilon)^3((1+\epsilon)^3+1)^{j-i-1}d_G(u,v)\,.\end{aligned}$$

To simplify the presentation in the following we set $a = (1+\epsilon)^3$ and thus have $\delta(u,v'') \leq a(a+1)^{j-i-1}d_G(u,v)$. By the triangle inequality we have

$$\begin{aligned}d_G(v'',v) &\leq d_G(v'',u) + d_G(u,v) \leq \delta(v'',u) + d_G(u,v)\\ &\leq (a(a+1)^{j-i-1}+1)d_G(u,v)\end{aligned}$$

and by the induction hypothesis we have

$$\begin{aligned}\delta'(v'',v) &\leq (2(a+1)^{k-1-j}-1)d_G(v'',v)\\ &\leq (2(a+1)^{k-1-j}-1)(a(a+1)^{j-i-1}+1)d_G(u,v)\,.\end{aligned}$$

Since $j \geq i+1$ we get

$$\begin{aligned}\delta'_j(u,v) &= \delta(u,v'') + \delta'(v'',v)\\ &\leq \left(a(a+1)^{j-i-1} + (2(a+1)^{k-1-j}-1)(a(a+1)^{j-i-1}+1)\right)d_G(u,v)\\ &= \left(2(a+1)^{k-1-j}(a(a+1)^{j-i-1}+1)-1\right)d_G(u,v)\\ &= \left(2a(a+1)^{k-1-(i+1)} + 2(a+1)^{k-1-j})-1\right)d_G(u,v)\\ &\leq \left(2a(a+1)^{k-1-(i+1)} + 2(a+1)^{k-1-(i+1)})-1\right)d_G(u,v)\\ &= \left(2(a+1)^{k-1-(i+1)}(a+1)-1\right)d_G(u,v)\\ &= (2(a+1)^{k-1-i}-1)d_G(u,v)\end{aligned}$$

□

Note that $2 \leq ((1+\epsilon)^3+1)$ and therefore we have $\delta'(u,v) \leq (((1+\epsilon)^3+1)^{k-i}-1)d_G(u,v)$. Furthermore, $(1+\epsilon)^3 \leq 1+7\epsilon$ and in the worst case $i=0$. Thus, by running the whole algorithm with $\epsilon' = \epsilon/7$, we can guarantee that $\delta'(u,v) \leq ((2+\epsilon)^k-1)d_G(u,v)$. □

3.7 Conclusion

In this chapter, we showed that single-source shortest paths in undirected graphs can be maintained under edge deletions with near-linear total update time and constant

query time. The main approach is to maintain an $(n^{o(1)}, \epsilon)$-hop set of near-linear size in near-linear time. We leave two major open problems. The first problem is whether the same total update time can be achieved for directed graphs. This problem is very challenging because such a hop set is not known even in the static setting. Moreover, improving the current $\tilde{O}(mn^{9/10+o(1)})$ total update time of Chapter 4 for the decremental reachability problem is already very interesting. The second major open problem is to derandomize our algorithm. The major task here is to deterministically maintain the priorities and corresponding balls of the nodes, which is the key to maintaining the hop set. A related question is whether the algorithm of Roditty and Zwick [113] for decrementally maintaining the original distance oracle of Thorup and Zwick (and the corresponding spanners and emulators) can be derandomized. (Note however that the distance oracle of Thorup and Zwick can be constructed deterministically in the static setting [110].)

CHAPTER 4

Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs

We consider dynamic algorithms for maintaining Single-Source Reachability (SSR) and approximate Single-Source Shortest Paths (SSSP) on n-node m-edge *directed* graphs under edge deletions (*decremental algorithms*). The previous fastest algorithm for SSR and SSSP goes back three decades to Even and Shiloach [49]; it has $O(1)$ query time and $O(mn)$ total update time (i.e., linear amortized update time if all edges are deleted). This algorithm serves as a building block for several other dynamic algorithms. The question whether its total update time can be improved is a major, long standing, open problem.

In this chapter, we answer this question affirmatively. We obtain a randomized algorithm with an expected total update time of $O(\min(m^{7/6}n^{2/3+o(1)}, m^{3/4}n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$ for SSR and $(1+\epsilon)$-approximate SSSP if the edge weights are integers from 1 to $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c. We also extend our algorithm to achieve roughly the same running time for Strongly Connected Components (SCC), improving the algorithm of Roditty and Zwick [114]. Our algorithm is most efficient for sparse and dense graphs. When $m = \Theta(n)$ its running time is $O(n^{1+5/6+o(1)})$ and when $m = \Theta(n^2)$ its running time is $O(n^{2+3/4+o(1)})$. For SSR we also obtain an algorithm that is faster for dense graphs and has a total update time of $O(m^{2/3}n^{4/3+o(1)} + m^{3/7}n^{12/7+o(1)})$ which is $O(n^{2+2/3})$ when $m = \Theta(n^2)$. All our algorithms have constant query time in the worst case and are correct with high probability against an oblivious adversary.

4.1 Introduction

Dynamic graph algorithms are data structures that maintain a property of a dynamically changing graph, supporting both update and query operations on the graph. In *undirected* graphs fundamental properties such as the connected, 2-edge connected, and 2-vertex connected components as well as a minimum spanning forest can be maintained *very quickly*, i.e., in polylogarithmic time per operation ([58, 71, 77, 122]), where an operation is either an edge insertion, an edge deletion, or a query. Some of these properties, such as connectivity, can even be maintained in polylogarithmic *worst-case* time. More general problems, such as maintaining distances, also admit sublinear amortized time per operation as long as only edge deletions are allowed [63, 64].

These problems when considered on *directed* graphs, however, become much harder. Consider, for example, a counterpart of the connectivity problem where we want to know whether there is a directed path from a node u to a node v, i.e., whether u can *reach* v. In fact, consider a very special case where we want to maintain whether a *fixed* node s can reach any node v under *edge deletions only*. This problem is called *single-source reachability* (SSR) in the *decremental* setting. It is one of the simplest, oldest, yet most useful dynamic graph problems. It is a special case of and was used as a subroutine for solving many dynamic graph problems, such as single-source shortest paths (SSSP), all-pairs shortest paths, and strongly connected components (SCC). Yet, no algorithm with sublinear update time was known for this problem.

Related Work The previously fastest decremental SSR algorithm was published in 1981 and takes $O(mn)$ total update time [49][1], i.e., linear time ($O(n)$ time) per update if we delete all m edges; here, n and m are the number of nodes and edges, respectively. For directed *acyclic* graphs, Italiano [74] gave a decremental algorithm with a total update time of $O(m)$. For the *incremental* version of the problem, where we only allow insertions of edges, a total update time of $O(m)$ is sufficient in general directed graphs [72]. For the *fully dynamic* version of the problem, where both insertions and deletions of edges are allowed, Sankowski obtained an algorithm with a worst-case running time of $O(n^{1.575})$ *per update*, resulting in a total update time of $O(\Delta n^{1.575})$, where Δ is the number of updates.[2]

King [79] showed how to extend the algorithm of Even and Shiloach to *weighted* graphs, giving the first decremental single-source shortest path algorithm with total update time $O(mnW)$, where W is the maximum edge weight (and all edge weights are positive integers)[3]. Using a scaling technique [22, 23, 95] this can be turned

[1] It was actually published for undirected graphs and it was observed by Henzinger and King [61] that it can be easily adapted to work for directed graphs.

[2] Sankowski's worst-case update time for the fully dynamic single-source single-sink reachability (stR) problem is $O(n^{1.495})$.

[3] The total update time is actually $O(md)$, where d is the maximum distance, which could be $\Theta(nW)$.

into a $(1+\epsilon)$-approximate single-source shortest paths algorithm with total update time $\tilde{O}(mn \log W)$, where the $\tilde{O}(\cdot)$ notation hides factors polylogarithmic in n. The situation is similar for decremental strongly connected components: The fastest decremental SCC algorithms take total update time $O(mn)$ ([90, 109, 114]). Thus many researchers in the field have asked whether the $O(mn)$ total update time for the decremental setting can be improved upon for these problems while keeping the query time constant or polylogarithmic [80, 90, 114].

Our Results We improve the previous $O(mn)$-time algorithms for decremental SSR, approximate SSSP, and SCC in directed graphs. We also give algorithms for *s-t reachability* (stR), where we want to maintain whether the node s can reach the node t, and approximate *s-t* shortest path (stSP) where we want to maintain the distance from s to t. We summarize our results in the following theorem. In Figure 4.1 we compare the running times of our new algorithms with the previous solution using Even and Shiloach for different densities of the graph.

Theorem 4.1.1. *There exist decremental algorithms for reachability and shortest path problems in directed graphs with the following expected total update times.*

- *SSR and SCC:* $\tilde{O}(\min(m^{7/6}n^{2/3}, m^{3/4}n^{5/4+o(1)}, m^{2/3}n^{4/3+o(1)} + m^{3/7}n^{12/7+o(1)})) = O(mn^{9/10+o(1)})$.
- *stR:* $\tilde{O}(\min(m^{5/4}n^{1/2}, m^{2/3}n^{4/3+o(1)}) = O(mn^{6/7+o(1)})$.
- $(1+\epsilon)$*-approximate SSSP:* $O(\min(m^{7/6}n^{2/3+o(1)}, m^{3/4}n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$.
- $(1+\epsilon)$*-approximate stSP:* $O(\min(m^{5/4}n^{1/2+o(1)}, m^{2/3}n^{4/3+o(1)}) = O(mn^{6/7+o(1)})$.

The algorithms are correct with high probability against an oblivious adversary and have constant query time. Our algorithms can maintain $(1+\epsilon)$*-approximate shortest paths in integer-weighted graphs with largest edge weight* W *if* $W \leq 2^{\log^c n}$ *and* $\epsilon \geq 1/\log^c n$ *for some constant* c.

Discussion Our main result are dynamic algorithms with constant query time and sublinear update time for single source reachability and $(1+\epsilon)$-approximate single source shortest paths in directed graphs undergoing edge deletions. There is some evidence that it is hard to generalize our results in the following ways.

- *(All pairs vs. single source)* The naive algorithm for computing all-pairs reachability (also called transitive closure) in a directed graph takes time $O(mn)$ even in the static setting. To date no combinatorial algorithm (not relying on fast matrix multiplication) is known that gives a polynomial improvement over this running time. Thus, unless there is a major breakthrough for static transitive closure, we cannot hope for a combinatorial algorithm for decremental all-pairs reachability with a total update time of $o(mn)$ and small query time.

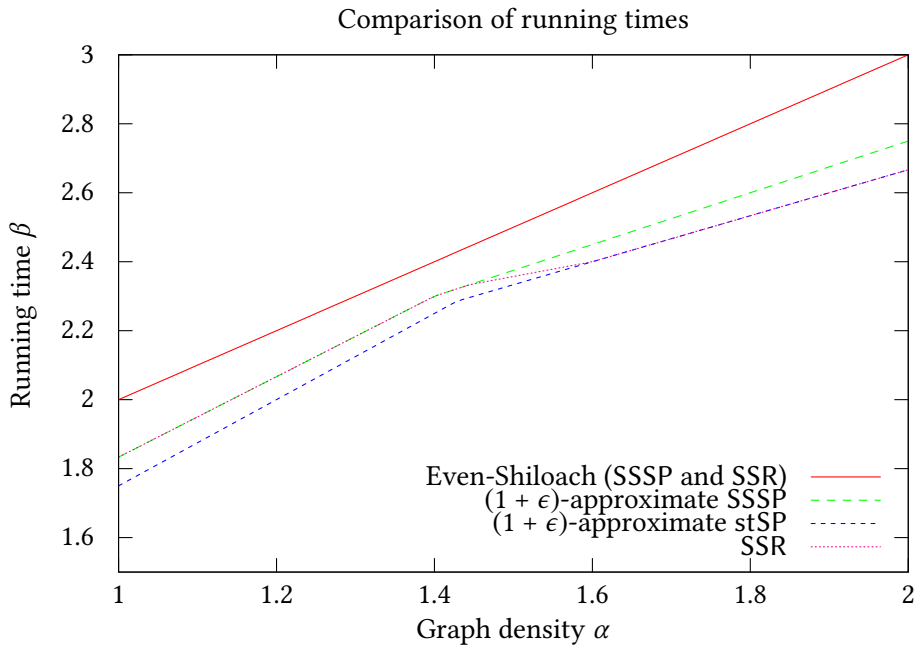


Figure 4.1: Running times of our decremental reachability and approximate shortest paths algorithms as a function of the density of the initial graph in comparison to the Even-Shiloach algorithm with total update time $O(mn)$. A point (α, β) in this diagram means that for a graph with $m = \Theta(n^\alpha)$ the algorithm has a running time of $O(n^{\beta+o(1)})$.

- *(Amortized update time)* The total update time of our single-source reachability algorithms is $o(mn)$ which gives an amortized update time of $o(n)$ over a sequence of $\Omega(m)$ deletions. It recently has been shown by Abboud and Vassilevska Williams [1] that a combinatorial algorithm with *worst-case* update time and query time of $o(n^2)$ per deletion implies a faster combinatorial algorithm for Boolean matrix multiplication and, as has been shown by Vassilevska Williams and Williams [128], for other problems as well. Furthermore, for the more general problem of maintaining the number of reachable nodes from a source under deletions (which our algorithms can do) worst-case update and query times of both $o(m)$ falsifies the strong exponential time hypothesis. It might therefore not be possible to deamortize our algorithms.

- *(Approximate vs. exact)* Our SSSP algorithms only provide approximate solutions. It has been observed by Roditty and Zwick [115] that any exact combinatorial SSSP algorithm handling edge deletions that has a total update time of $o(mn)$ and small query time implies a faster combinatorial for Boolean matrix multiplication. After the preliminary version [68] of this work appeared, Henzinger et al. [70] showed that $O(mn)$ is essentially the best pos-

sible total update time for maintaining exact distances under the assumption that there is no "truly subcubic" algorithm for a problem called online Boolean matrix-vector multiplication. Both of these hardness results apply even for unweighted undirected graphs. Thus, approximation might be necessary to break the $O(mn)$ barrier.

Organization We first introduce the notation and basic concepts shared by all our algorithms in Section 4.2. To provide some intuition for our approach we sketch two basic s-t reachability algorithms in Section 4.2.3. We give the reductions for extending our results to SSSP (and SSR) and SCC in Sections 4.2.4 and 4.2.5, respectively. In Section 4.3 we give a hierarchical s-t reachability algorithm, which in turn gives algorithms for SSR and SCC. In Section 4.4 we extend this idea to $(1+\epsilon)$-approximate stSP and SSSP. Finally, in Section 4.5 we give an SSR algorithm with improved total update time for dense graphs. We conclude this chapter by discussing open problems.

4.2 Preliminaries

4.2.1 Problem Description

We are given a directed graph G that might be weighted or unweighted. The graph undergoes a sequence of *updates* in the form of edge deletions, i.e., its set of edges shrinks over time (whereas its set of nodes remains the same). This is called the *decremental setting*. We say that a node v is *reachable* from u (or u can reach v) if there is a path from u to v in G. The *distance* $d_G(x, y)$ of a node x to a node y in G is the weight of the shortest path, i.e., the minimum-weight path, from x to y in G. If there is no path from x to y in G we set $d_G(x, y) = \infty$.

Our goal is to design efficient dynamic algorithms for the following problems.

Definition 4.2.1 (Single-source single-sink reachability)**.** *A* decremental single-source reachability (stR) algorithm *for a directed graph G undergoing edge deletions, a source node s, and a sink node t maintains the information whether s can reach t. It supports the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G.*
- Query()*: Return 'yes' if s can reach t and 'no' otherwise.*

Definition 4.2.2 (Single-source reachability)**.** *A* decremental single-source reachability (SSR) algorithm *for a directed graph G undergoing edge deletions and a source node s maintains the set of nodes reachable from s in G. It supports the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G.*
- Query(v)*: Return 'yes' if s can reach v and 'no' otherwise.*

Definition 4.2.3 (Strongly connected components)**.** *A* decremental strongly connected components (SCC) algorithm *for a directed graph G undergoing edge deletions maintains a set of IDs of the strongly connected components of G and, for every node v, the ID of the strongly connected component that contains v. It supports the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G.*
- Query(v)*: Return the ID of the strongly connected component that contains v.*

Definition 4.2.4 (α-approximate stSP)**.** *A* decremental α-approximate single-source single-sink shortest path (stSP) algorithm *for a weighted directed graph G undergoing edge deletions, a source node s, and a sink node t maintains a distance estimate $\delta(s, t)$ such that $d_G(s, t) \leq \delta(s, t) \leq \alpha d_G(s, t)$. It supports the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G.*
- Query()*: Return the α-approximate distance estimate $\delta(s, t)$.*

Definition 4.2.5 (α-approximate SSSP)**.** *A* decremental α-approximate single-source shortest paths (SSSP) algorithm *for a weighted directed graph G undergoing edge deletions and edge weight increases and a source node s maintains, for every node v, a distance estimate $\delta(s, v)$ such that $d_G(s, v) \leq \delta(s, v) \leq \alpha d_G(s, v)$. It supports the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G.*
- Query(v)*: Return the α-approximate distance estimate $\delta(s, v)$.*

The algorithms we design will have constant query time and we will compare them by their *total update time* over all deletions. They are randomized and will be correct with high probability (whp) against an oblivious adversary. We assume that arithmetic operations on integers can be performed in constant time. The total update times we state are in expectation. We use $\tilde{O}$-notation to hide logarithmic factors, i.e., we write $\tilde{O}(t(m, n, W))$ as an abbreviation for $\tilde{O}(t(m, n, W) \log^c n)$ when c is a constant. Similarly, we use the notation $\hat{O}(t(m, n))$ as an abbreviation for $O(t(m, n) \cdot n^{o(1)})$. The approximation factors we obtain will be of the form $\alpha = 1 + \epsilon$ such that $0 < \epsilon \leq 1$. In this chapter we assume that $\epsilon \geq 1/\log^c n$ for some constant c.

4.2.2 Definitions and Basic Properties

In the following introduce the notation and the basic concepts shared by all our algorithms.

Let $G = (V, E)$ be a weighted directed graph, where V is the set of nodes of G and E is the set of edges of G. We denote by n the number of nodes of G and by m the number of edges of G *before the first edge deletion.* We denote the *weight* of an edge (u, v) in G by $w_G(u, v)$. In the weighted case we consider positive integer edge

weights and denote the maximum edge weight by W. We assume that $W \leq 2^{\log^c n}$ for some constant c. In unweighted graphs we think of every edge weight as equal to 1. For every path $\pi = \langle v_0, v_2, \dots, v_k \rangle$ we denote its weight in G by $w(\pi, G) = \sum_{0 \leq i \leq k-1} w_G(v_i, v_{i+1})$ and its number of edges (also called *hops*) by $|\pi| = k$. We say that the *length* of a path is its weight, but, to avoid ambiguity, we reserve this notion for unweighted graphs where the length of a path is equal to its number of edges. For every integer $h \geq 1$ and all nodes x and y in a directed graph G, the *h-hops distance* $d^h_G(x, y)$ is the minimum weight of all paths from x to y in G consisting of at most h edges. Note that $d_G(u, v) = d^n_G(u, v)$. For every graph $G = (V, E)$ and every subset of nodes $U \subseteq V$, $G|U = (U, E \cap U^2)$ is the *subgraph of G induced by U*. We set the weight of every edge in $G|U$ equal to its weight in G. We denote by $E(U) = E \cap U^2$ the set of edges of $G|U$. For sets of nodes $S \subseteq V$ and $T \subseteq V$ we let $E(S, T)$ denote the set of edges $(u, v) \in E$ such that $u \in S$ and $v \in T$, i.e., $E(S, T) = E \cap (S \times T)$.

We now introduce the main new concept used by our algorithms, the path union of a pair of nodes.

Definition 4.2.6. *For every directed graph G, every $D \geq 1$, and all pairs of nodes x and y of G, the path union $\mathscr{P}(x, y, D, G) \subseteq V$ is the set containing all nodes that lie on some path π from x to y in G of weigh $w_G(\pi) \leq D$.*

Note that if the shortest path from x to y in G has weight at most D, then the subgraph of G induced by the path union $\mathscr{P}(x, y, D, G)$ contains this shortest path. Thus, instead of finding this shortest path in G directly we can also find it in this potentially smaller subgraph of G. In our algorithms we will be able to bound the number of the path union subgraphs, which makes them very useful. Observe that for a fixed value of D, the path union can be computed in nearly linear time.

Lemma 4.2.7. *For every directed graph G, every $D \geq 1$ and all pairs of nodes x and y of G, we have $\mathscr{P}(x, y, D, G) = \{v \in V \mid d_G(x, v) + d_G(v, y) \leq D\}$. We can compute this set in time $\tilde{O}(m)$ in weighted graphs and $O(m)$ in unweighted graphs, respectively.*

Proof. Clearly, if $d_G(x, v) + d_G(v, y) \leq D$, then the concatenation of the shortest path from x to v and the shortest path from v to y is a path from x to y of weight at most D and thus $v \in \mathscr{P}(x, y, D, G)$. Conversely, if $v \in \mathscr{P}(x, y, D, G)$, then there is a shortest path π from x to y containing v of weight at most D. Let π_1 and π_2 be the subpaths of π from x to v and from v to y, respectively. Then $d_G(x, v) + d_G(v, y) \leq w_G(\pi_1) + w_G(\pi_2) = w_G(\pi) \leq D$.

Using Dijkstras algorithm we compute $d_G(x, v)$ and $d_G(v, y)$ for every node $v \in V$ in time $\tilde{O}(m)$. Afterwards, we iterate over all nodes and check for every node v whether $d_G(x, v) + d_G(v, y) \leq D$ in total time $O(n)$. In unweighted graphs we can compute $d_G(x, v)$ and $d_G(v, y)$ for every node $v \in V$ in time $O(m)$ by performing breadth-first search (BFS). □

Observe that path unions are monotone in the decremental setting, i.e., the path union can only shrink under deletions in the graph for a fixed value of D. This means

that once we have computed the path union we can also use it for future versions of the graph, as long as we do not want to consider larger weights of the paths.

Lemma 4.2.8. *Let G be a directed graph and let G' be the result of deleting some edges from G. Then for every $D \geq 1$ and all pairs of nodes x and y of G, $\mathscr{P}(x, y, h, G') \subseteq G|\mathscr{P}(x, y, h, G)$.*

Proof. Let $v \in \mathscr{P}(x, y, h, G')$, i.e., there is a path π from x to y in G' of weight at most D. All edges of π are also contained in G and thus there is a path from x to y in G' of weight $w_G(\pi) = w_{G'}(\pi) \leq D$. Thus, every node of π is contained in $\mathscr{P}(x, y, h, G)$ which implies that all edges of π are contained in $G|\mathscr{P}(x, y, h, G)$. Therefore $w_{G|\mathscr{P}(x,y,h,G)}(\pi) = w_G(\pi) \leq D$ and thus every node on π, and in particular v, is contained in $G|\mathscr{P}(x, y, h, G)$. □

The last property of path unions we will use repeatedly is that we can update them by "computing the path union of the path union". This rebuilding of path unions will be useful later to restrict the overlap of path unions of different pairs of nodes.

Lemma 4.2.9. *For every directed graph G, every $D \geq 1$, every pair of nodes x and y, and every set of nodes Q such that $\mathscr{P}(x, y, D, G) \subseteq Q \subseteq V$ we have $\mathscr{P}(x, y, D, G|Q) = \mathscr{P}(x, y, D, G)$.*

Proof. Let $v \in \mathscr{P}(x, y, D, G|Q)$, which means that v lies on a path π from x to y of weight at most D. As $G|Q$ is a subgraph of G, this path is also contained in G (with the same weight) and thus $v \in \mathscr{P}(x, y, D, G)$.

Now let $v \in \mathscr{P}(x, y, D, G)$, which means that v lies on a path π from x to y of weight at most D. By the assumption $\mathscr{P}(x, y, D, G) \subseteq Q$ every node v' of π is contained in Q and thus π is contained in $G|Q$ (and has the same weight as in G). As v lies on a path from x to y of weight at most D in G, we have $v \in \mathscr{P}(x, y, D, G|Q)$. □

4.2.3 Algorithm Overview for s-t Reachability

In this section, we illustrate our main ideas by giving simple algorithms for the s-t reachability problem (stR). At the heart of all our algorithms is a new way of maintaining reachability or distances between some *nearby* pairs of nodes using small path unions.

Decremental Bounded-Hop Multi-Pair Reachability

We first give an algorithm for solving the following "restricted" reachability problem: We are given k pairs of sources and sinks (s_1, t_1), ..., (s_k, t_k) and a parameter h. We want to maintain, for each $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$.

In this algorithm we use the following set $B \subseteq V$ of *hubs* of size $\tilde{O}(b)$ (for some parameter b). Let $U \subseteq V$ be a set of nodes obtained by sampling each node independently with probability $ab \ln n/n$ and let $F \subseteq E$ be a set of edges obtained by

sampling each edge independently with probability $ab \ln n/m$ from the initial graph (for a large enough constant a). We let B be the set containing U and the endpoints of every edge in F. For every pair of nodes u and v we define the *hub-distance from u to v* as $d_B(u, v) = \min_{x \in B} d_G(u, x) + d_G(x, v)$.

The hubs are useful in combination with the path unions in the following way. Intuitively, instead of maintaining whether $d_G(s_i, t_i) \leq h$, we can maintain whether $d_{G|\mathscr{P}(s_i,t_i,h,G)}(s_i, t_i) \leq h$ since $G|\mathscr{P}(s_i, t_i, h, G)$ (the subgraph of G induced by $\mathscr{P}(s_i, t_i, h, G)$) contains all s_i-t_i paths of length at most h in G. This could be helpful when $\mathscr{P}(s_i, t_i, h, G)$ is much smaller than V. Our first key idea is the observation that all (s_i, t_i) pairs with large $\mathscr{P}(s_i, t_i, h, G)$ can use their paths through a small number of hubs to check whether $d_{G|Q(s_i,t_i)}(s_i, t_i) \leq h$ (thus the name "hub"). In particular, consider the following algorithm. We maintain the distance from each s_i to each t_i in two ways.

1. Maintain $d_B(s_i, t_i)$, for each $1 \leq i \leq k$, as long as $d_B(s_i, t_i) \leq h$.

2. Once $d_B(s_i, t_i) > h$, construct $Q(s_i, t_i) = \mathscr{P}(s_i, t_i, h, G)$ and maintain $d_{G|Q(s_i,t_i)}(s_i, t_i)$ up to value h.

Our outputs $d_G(s_i, t_i) \leq h$ if and only if either $d_B(s_i, t_i) \leq h$ or $d_{G|Q(s_i,t_i)}(s_i, t_i) \leq h$. The correctness is obvious: either $d_B(s_i, t_i) \leq h$ which already implies that $d_G(s_i, t_i) \leq h$ or otherwise we maintain $d_{G|Q(s_i,t_i)}(s_i, t_i)$ which captures all h-hop s_i-t_i paths. The more important point is the efficiency of maintaining both distances. Our analysis mainly uses the following lemma.

Lemma 4.2.10 (Either hub-distance or path-union graph is small)**.** *With high probability, for each i, either $d_B(s_i, t_i) \leq h$, or $G|\mathscr{P}(s_i, t_i, h, G)$ has at most $\min(m/b, n^2/b^2)$ edges.*

Proof Sketch. By the random sampling of the hubs, if $G|\mathscr{P}(s_i, t_i, h, G)$ has more than n/b nodes, then one of these nodes, say x, will have been sampled by the algorithm and thus contained in B whp (Lemma 1.3.2). By definition, x lies on some s_i-t_i path of length at most h. Thus, $d_B(s_i, t_i) \leq d_G(s_i, x) + d_G(x, t_i) \leq h$. Similarly, if $G|\mathscr{P}(s_i, t_i, h, G)$ has more than m/b edges, then one of these edges, say (x, y), will have been sampled by the algorithm whp. This means that x will be contained in B, and thus $d_B(s_i, t_i) \leq d_G(s_i, x)+d_G(x, t_i) \leq h$. As the number of edges of $G|\mathscr{P}(s_i, t_i, h, G)$ can be at most $|\mathscr{P}(s_i, t_i, h, G)|^2$, the stated bound follows. □

Lemma 4.2.10 guarantees that when we construct $Q(s_i, t_i)$, its induced subgraph is much smaller than G whp, and so it is beneficial to maintain the distance in $G|Q(s_i, t_i)$ instead of G. For each $1 \leq i \leq k$, we can maintain $d_{G|Q(s_i,t_i)}(s_i, t_i)$ by running an ES-tree rooted at s_i up to distance h in $G|Q(s_i, t_i)$, which takes time $\tilde{O}(|E(Q(s_i, t_i))|h)$ (where $E(Q(s_i, t_i))$ denotes the set of edges of $G|Q(s_i, t_i)$). As $|E(Q(s_i, t_i))| \leq \min(m/b, n^2/b^2)$ this takes time $O(kh \min(m/b, n^2/b^2))$. By Lemma 4.2.7 we can construct each $Q(s_i, t_i)$ in $O(m)$ time for each $1 \leq i \leq k$, resulting in a total cost of $O(km)$ for computing the

path unions. Finally, the time needed for maintain the hub distances can be analyzed as follows.

Lemma 4.2.11 (Maintaining all $d_B(s_i, t_i)$)**.** *We can maintain whether $d_B(s_i, t_i) \leq h$, for all $1 \leq i \leq k$, in total time $\tilde{O}(bmh + kbh)$.*

Proof Sketch. For each hub $v \in B$, we maintain the values of $d_G(s_i, v)$ and $d_G(v, t_i)$ up to h using ES-trees up to distance h rooted at each hub. This takes $\tilde{O}(bmh)$ total update time. Every time $d_G(s_i, v)$ or $d_G(v, t_i)$ changes, for some hub $v \in B$ and some $1 \leq i \leq k$, we update the value of $d_G(s_i, t_i) = \min_{v \in B} d_G(s_i, v) + d_G(v, t_i)$, incurring $\tilde{O}(b)$ time. Since the value of each of $d_G(s_i, v)$ and $d_G(v, t_i)$ can change at most h times, we need $\tilde{O}(kbh)$ time in total. □

It follows that we can maintain, for all $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$ with a total update time of

$$\tilde{O}\Big(\underbrace{bmh + kbh}_{\substack{\text{maintain } d_B(s_i,t_i) \\ \text{(Lemma 4.2.11)}}} + \underbrace{km}_{\text{construct } Q(s_i,t_i)} + \underbrace{kh \cdot \min(m/b, n^2/b^2)}_{\text{maintain } d_{G|Q(s_i,t_i)}(s_i,t_i)} \Big). \tag{4.1}$$

Note that, previously, the fastest way of maintaining, for all $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$ was to maintain an ES-tree separately for each pair. This takes $O(mhk)$ time.

Decremental s-t reachability in Dense Graphs

In the following we are given a source node s and a sink node t and want to maintain whether s can reach t. In our algorithm we use a set of *centers* $C \subseteq V$ of size $\tilde{O}(c)$ (for some parameter c) obtained by sampling each node independently with probability $ac \ln n/n$ (for some large enough constant a). Using these centers, we define the *center graph*, denoted by $\mathscr{C}$, as follows. The nodes of $\mathscr{C}$ are the centers in C and for every pair of centers u and v in C, there is a directed edge (u, v) in $\mathscr{C}$ if and only if $d_G(u, v) \leq n/c$.

Lemma 4.2.12 ($\mathscr{C}$ preserves s-t reachability)**.** *Whp, s can reach t in G if and only if s can reach t in $\mathscr{C}$.*

Proof Sketch. Since we can convert any path in $\mathscr{C}$ to a path in G, the "if" part is clear. To prove the "only if" part, let π be an s-t path in G. By the random sampling of centers, there is a set of centers $c_1, c_2, \ldots, c_k$ on π such that $c_1 = s$, $c_k = t$, and $d_G(c_i, c_{i+1}) \leq n/c$ for all i whp (Lemma 1.3.2). The last property implies that the edge (c_i, c_{i+1}) is contained in $\mathscr{C}$ for all i. Thus s can reach t in $\mathscr{C}$. □

Lemma 4.2.12 implies that to maintain s-t reachability in $\mathscr{C}$ it is sufficient to maintain the edges of $\mathscr{C}$ and to maintain s-t reachability in $\mathscr{C}$. To maintain the edges of $\mathscr{C}$, we simply have to maintain the distances between all pairs of centers up to n/c. This can be done using the bounded-hop multi-pair reachability algorithm from

before with $k = c^2$ and $h = n/c$, where we make each pair of centers a source-sink pair. By plugging in these values in Equation (4.1), we obtain a total update time of $\tilde{O}(bmn/c + bcn + c^2m + cn^3/b^2)$ for this this bounded-hop multi-pair reachability. To maintain s-t reachability in $\mathscr{C}$, note that $\mathscr{C}$ is a dynamic graph undergoing only deletions. Thus, we can simply maintain an ES-tree rooted at s in $\mathscr{C}$ As $\mathscr{C}$ has $\tilde{O}(c)$ nodes and $\tilde{O}(c^2)$ edges this takes time $\tilde{O}(c^3)$. It follows that the total update time of this s-t reachability algorithm is

$$\tilde{O}\Big(\underbrace{bmn/c + bnc}_{\substack{\text{maintain } d_B(\cdot,\cdot) \\ \text{(Lemma 4.2.11)}}} + \underbrace{c^2 m}_{\text{construct } Q(\cdot,\cdot)} + \underbrace{cn^3/b^2}_{\text{maintain } d_{G|Q(s_i,t_i)}(\cdot,\cdot)} + \underbrace{c^3}_{\text{maintain } d_{\mathscr{C}}(s,t)} \Big). \tag{4.2}$$

By setting[4] $b = n^{8/7}/m^{3/7}$ and $c = (bn)^{1/3} = n^{5/7}/m^{1/7}$, we get a total update time of[5] $\tilde{O}\left(m^{5/7}n^{10/7} + n^{20/7}/m^{4/7}\right)$. This is $o(mn)$ if $m = \omega(n^{3/2})$.

Decremental s-t reachability in Sparse Graphs

Maintaining s-t reachability in sparse graphs, especially when $m = \Theta(n)$, needs a slightly different approach. Carefully examining the running time of the previous algorithm in Equation (4.2) reveals that we *cannot* maintain $d_{G|Q(u,v)}(u,v)$ for *all* pairs of centers u and v at all times: this would cost $\tilde{O}(\min(cmn/b, cn^3/b^2))$—the third term of Equation (4.1)—but we always need $c = \omega(b)$ to keep the first term of Equation (4.1) to $bmn/c = o(mn)$ and, since $b \le n$, $cmn/b = \omega(mn)$ and $cn^3/b^2 = \omega(n^2) = \omega(mn)$. The new strategy is to maintain $d_{G|Q(u,v)}(u,v)$ only for *some* pairs of centers at each time step.

Algorithm As before, we sample $\tilde{O}(b)$ hubs $\tilde{O}(c)$ centers, and maintain $d_B(\cdot,\cdot)$ between all centers which takes $\tilde{O}(mbn/c + bnc)$ time (Lemma 4.2.11). The algorithm runs in phases. At the beginning of each phase i, the algorithm does the following. Compute a BFS-tree T on the outgoing edges of every node rooted at the source s in G. If T does not contain t, then we know that s cannot reach t anymore and there is nothing to do. Otherwise, let $L = (c_1, c_2, \ldots, c_k)$ be the list of centers on the shortest path from s to t in T ordered increasingly by their distances to s. For simplicity, we let $c_0 = s$ and $c_{k+1} = t$. Note that we can assume that

$$\forall i,\ d_G(c_i, c_{i+1}) \le n/c \ \text{ and } d_G(c_i, c_{i+2}) > n/c\,. \tag{4.3}$$

The first inequality holds because the centers are obtained from random sampling (Lemma 1.3.2) and the second one holds because, otherwise, we can remove c_{i+1} from

[4] Note that we have to make sure that $1 \le b \le n$ and $1 \le c \le n$. It is easy to check that this is the case using the fact that $m \le n^2$.

[5] *Detailed calculation:* First note that $c \le n^{5/7} \le m$. So, the term c^3 is dominated by the term mc^2. Observe that $b = (n^3/mc)^{1/2}$, thus $n^3c/b^2 = mc^2$. So, the fourth term is the same as the third term (mc^2). Using $c = (bn)^{1/3}$, we have that the first and third terms are the same. Now, the third term is $mc^2 = m(n^{5/7}/m^{1/7})^2 = m^{1-2/7}n^{10/7} = m^{5/7}n^{10/7}$. For the second term, using $bn = c^3$, we have $bnc = c^4 = n^{20/7}/m^{4/7}$.

the list L without breaking the first inequality. Observe that L would induce an s-t path in the "center graph". Our intention in this phase is to maintain whether s can still reach t using this path; in other words, whether $d_G c_i, c_{i+1}) \leq n/c$ for all i. (We start a new phase if this is not the case.) We do this using the framework of Section 4.2.3: For each pair (c_i, c_{i+1}), we know that $d_G(c_i, c_{i+1}) \leq n/c$ when $d_B(c_i, c_{i+1}) \leq n/c$. Once $d_B(c_i, c_{i+1}) > n/c$, we construct $Q(c_i, c_{i+1}) = \mathscr{P}(c_i, c_{i+1}, n/c, G)$. After each deletion of an edge (u, v) in this phase, we find every index i such that u and v are contained in $Q(c_i, c_{i+1})$ and, using the static BFS algorithm, check whether $d_{G|Q(c_i,c_{i+1})}(c_i, c_{i+1}) \leq n/c$. We start a new phase when $d_G(c_i, c_{i+1})$ for some pair of centers (c_i, c_{i+1}) changes from $d_G(c_i, c_{i+1}) \leq n/c$ to $d_G(c_i, c_{i+1}) > n/c$.

Running Time Analysis First, let us bound the number of phases. As for each pair (c_i, c_{i+1}) the distance $d_G(c_i, c_{i+1})$ can only become larger than n/c at most once, there are at most $\tilde{O}(c^2)$ phases. At the beginning of each phase, we have to construct a BFS-tree in G, taking $O(m)$ time and contributing $\tilde{O}(c^2 m)$ to the total time over all phases. During the phase, we have to construct $Q(c_i, c_{i+1})$ for at most c pairs of center, taking $\tilde{O}(cm)$ time (by Lemma 4.2.7) and contributing $\tilde{O}(c^3 m)$ total time. Moreover, after deleting an edge (u, v), we have to update $d_{G|Q(c_i,c_{i+1})}(c_i, c_{i+1})$ for every $Q(c_i, c_{i+1})$ containing both u and v, by running a BFS algorithm. This takes $O(|E(Q(c_i, c_{i+1})|)$ time for each $Q(c_i, c_{i+1})$, which is $\tilde{O}(m/b)$ whp, by Lemma 4.2.10. The following lemma implies that every node will be contained in only a constant number of such sets $Q(c_i, c_{i+1})$; so, we need $\tilde{O}(m^2/b)$ time to update $d_{G|Q(c_i,c_{i+1})}(c_i, c_{i+1})$ over all m deletions.

Lemma 4.2.13. *For any i and $j \geq i + 3$, $Q(c_i, c_{i+1})$ and $Q(c_j, c_{j+1})$ are disjoint.*

Proof. First, we claim that $d_G(c_i, c_{j+1}) > 2n/c$. To see this, let G' be the version of the graph at the beginning of the current phase. We know that $d_{G'}(c_i, c_{i+4}) = d_{G'}(c_i, c_{i+2}) + d_{G'}(c_{i+2}, c_{i+4}) > 2n/c$, where the equality holds because at the beginning of the current phase (i.e., in G'), every c_i lies on the shortest s-t path, and the inequality holds because of Equation (4.3). Since $j + 1 \geq i + 4$ and both c_{i+4} and c_{j+1} lie on the shortest s-t path in G', $d_{G'}(c_i, c_{j+1}) \geq d_{G'}(c_i, c_{i+4}) > 2n/c$. The claim follows since the distance between two nodes never decreases after edge deletions.

Now, suppose for the sake of contradiction that there is some node v that is contained in both $Q(c_i, c_{i+1}) = \mathscr{P}(c_i, c_{i+1}, n/c, G)$ and $Q(c_j, c_{j+1}) = \mathscr{P}(c_j, c_{j+1}, n/c, G)$. This means that v lies in some c_i-c_{i+1} path and some c_j-c_{j+1} path, each of length at most n/c. This means that $d_G(c_i, v) \leq n/c$ and $d_G(v, c_{j+1}) \leq n/c$ and therefore we get $d_G(c_i, c_{j+1}) \leq d_G(c_i, v) + d_G(v, c_{j+1}) \leq 2n/c$. This contradicts the lower bound above. □

Thus, the total update time is

$$\tilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } d_B(\cdot,\cdot) \\ \text{(Lemma 4.2.11)}}} + \underbrace{mc^2}_{\substack{\text{construct BFS} \\ \text{tree in every phase}}} + \underbrace{mc^3}_{\substack{\text{construct } Q(c_i, c_{i+1}) \\ \text{in every phase}}} + \underbrace{m^2/b}_{\text{update } d_{G|Q(c_i,c_{i+1})}(c_i, c_{i+1})} \Big).$$

By setting $c = (mn)^{1/7}$ and $b = c^4/n = m^{4/7}/n^{3/7}$, we get a running time of $\tilde{O}\left(m^{10/7}n^{3/7}\right)$.[6] This is $o(mn)$ if $m = o(n^{4/3})$.

4.2.4 Single-Source Shortest Paths

In the following we show a reduction of decremental approximate single-source single-sink shortest path to decremental approximate single-source shortest paths. The naive way of doing this would be to use n instances of the approximate single-source single-sink shortest path algorithm, one for every node. We can use much fewer instances by randomly sampling the nodes at which we maintain single-source single-sink shortest path and by using Bernstein's shortcut edges technique [23].

Theorem 4.2.14. *Assume we already have the following decremental algorithm that, given a weighted directed graph G undergoing edge deletions, a source node s, and a set of sinks T of size k, maintains, for every sink $t \in T$, a distance estimate $\delta(s,t)$ such that $d_G(s,t) \leq \delta(s,t) \leq \alpha d_G(s,t))$ for some $\alpha \geq 1$ with constant query time and a total update time of $T(k,m,n)$. Then, for any $k \leq n$, there exists a decremental $(1+\epsilon)\alpha$-approximate SSSP algorithm with constant query time and expected total update time $O(T(O(k \log n), m, n) + mn \log(nW)/(\epsilon k))$ that is correct with high probability against an oblivious adversary.*

Proof. At the initialization we randomly sample each node of G with probability $ak \ln n/n$ (for a large enough constant a). We call the sampled nodes sinks. We use the decremental algorithm for a set of sinks from the assumption to maintain a distance estimate $\delta(s,t)$ for every sink t such that $d_G(s,t) \leq \delta(s,t) \leq \alpha d_G(s,t)$. Additionally, we maintain a graph G' which consists of the graph G augmented by the following edges: for every sink t we add an edge (s,t) of weight $w'(s,t) = (1+\epsilon)^{\lceil \log_{1+\epsilon} \delta(s,t) \rceil}$ (these edges are called *shortcut edges*). We maintain G' by updating the weights of these edges every time the distance estimate $\delta(s,t)$ of some sink t changes its value. On G' we use Bernstein's decremental $(1+\epsilon)$-approximate SSSP algorithm [23] with source s and hop count $h = n/k$. This algorithm maintains a distance estimate $\delta'(s,v)$ for every node v such that $d_{G'}(s,v) \leq \delta'(s,v) \leq (1+\epsilon) d^h_{G'}(s,v)$.

First, observe that $d_G(s,v) = d_{G'}(s,v)$ as the new shortcut edges never underestimate the true distances in G. We now claim that $d^h_{G'}(s,v) \leq (1+\epsilon)\alpha d_G(s,v)$. If $d_G(s,v) = \infty$, then the claim is trivially true. Otherwise let π be the shortest path from s to v in G. All edges of this path are also contained in G'. Thus, if π has at most h edges, then $d^h_{G'}(s,v) = d_G(s,v)$. If π has more than h edges, then we know that the set of nodes consisting of the last h nodes of π contains a sink t whp by Lemma 1.3.2 as the sinks are obtained by sampling from the nodes with probability $ak \ln n/n = a \ln n/h$. Thus, the graph G' contains a shortcut edge (s,t) of weight $w'(s,t) \leq (1+\epsilon)\delta(s,t) \leq (1+\epsilon)\alpha d_G(s,t)$. Now let π' be the path from s to v that starts

[6] *Detailed calculation:* First note that the third term (mc^2) is dominated by the fourth term (mc^3). Using $b = c^4 n$, the first term is the same as the fourth term (mc^3). Now, the fourth term is $mc^3 = m(mn)^{3/7} = m^{10/7}n^{3/7}$. For the second term, using $bn = c^4$, we have $bnc = c^5 = (mn)^{5/7}$ which is at most $m^{10/7}n^{3/7}$ (using $n \leq m$). For the last term, $m^2/b = m^2/(m^{4/7}/n^{3/7}) = m^{(14-4)/7}n^{3/7}$.

with this edge (s, t) and then follows the path π from t to v. Clearly, the path π' has at most h edges and is contained in G'. As the weight of π' is $w'(s, t) + d_G(t, v)$ we get

$$d^h_{G'}(s, v) = w'(s, v) + d_G(t, v) \leq (1 + \epsilon)\alpha d_G(s, t) + d_G(t, v) \leq (1 + \epsilon)\alpha d_G(s, v)\,.$$

Putting everything together, we get that the distance estimate $\delta'(s, v)$ fulfills

$$\begin{aligned} d_G(s, v) = d_{G'}(s, v) \leq \delta'(s, v) \leq (1 + \epsilon) d^h_{G'}(s, v) &\leq (1 + \epsilon)^2 \alpha d_G(s, v) \\ &\leq (1 + 3\epsilon)\alpha d^h_{G'}(s, v) \end{aligned}$$

By running the whole algorithm with $\epsilon' = \epsilon/3$ we obtain a $(1 + \epsilon)\alpha$-approximation instead of a $(1 + \epsilon)\alpha$-approximation.

Finally, we argue about the running time. Our running time has two parts. (1) Bernstein's decremental $(1 + \epsilon)$-approximate SSSP algorithm [23] has constant query time and a total update time of $O(mh \log(nW)/\epsilon)$ (here we also use the fact that $w'(u, t)$ increases $O \log(nW)/\epsilon$ times for every sink t). Thus, our decremental algorithm also has constant query time and by our choice of $h = n/k$, our total update time contains the term $mn \log(nW)/(\epsilon k)$. (2) Furthermore, we have $O(k \log n)$ sinks in expectation. Thus, the expected total update time of the decremental algorithm for a set of sinks from the assumption is $T(O(k \log n), m, n)$. □

4.2.5 Strongly Connected Components

In the following we reduce decremental strongly connected components to decremental single-source reachability. Our reduction is almost identical to the one of Roditty and Zwick [114], but in order to work in our setting we have to generalize their running time analysis. They show that an $O(mn)$ algorithm for single-source reachability implies an $O(mn)$ algorithm for strongly connected components. We show that in fact $o(mn)$ time for single-source reachability implies $o(mn)$ time for strongly connected components. In the following we will often just write "component" instead of "strongly connected component".

In contrast to the rest of this chapter, we will here impose the following technical condition on the decremental single-source reachability algorithm: when we update the algorithm after the deletion of an edge, the update procedure will return all nodes that were reachable before the deletion, but are not reachable anymore after this deletion. Note that all the reachability algorithms we present in this chapter fulfill this condition.

Algorithm The algorithm works as follows. For every component we, uniformly at random, choose among its nodes one *representative*. In an array, we store for every node a pointer to the representative of its component. Queries that ask for the component of a node v are answered in constant time by returning (the ID of) the representative of v's component. Using the decremental SSR algorithm, we maintain,

for every representative w of a component C, the sets $I(w)$ and $O(w)$ containing all nodes that reach w and that can be reached by w, respectively. Note that, for every node v, we have $v \in C$ if and only if $v \in I(w)$ and $v \in O(w)$. After the deletion of an edge (u, v) such that u and v are contained in the same component C we check whether C decomposes. This is the case only when, after the deletion, $u \notin I(w)$ or $v \notin O(w)$ (which can be checked with the SSR algorithm of w).

We now explain the behavior of the algorithm when a component C decomposes into the new components $C_1, \dots, C_k$. The algorithm chooses a new random representative w_i for every component C_i and starts maintaining the sets $I(w_j)$ and $O(w_j)$ using two new decremental SSR algorithms. There is one notable exception: If the representative w of C is still contained in one of the components C_j. For this component we do *not* choose a new representative. Instead, C_j reuses w and its SSR algorithms without any re-initialization. The key to the efficiency of the algorithm is that a large component C_i has a high probability of inheriting the representative from C.

Note that before choosing the new representatives we actually have to determine the new components $C_1, \dots, C_k$. We slightly deviate from the original algorithm of Roditty and Zwick to make this step more efficient. If $w \in C_j$, then it is not necessary to explicitly compute C_j as all nodes in C_j keep their representative w. We only have to explicitly compute $C_1, \dots, C_{j-1}, C_{j+1}, \dots C_k$. This can be done as follows: Let A denote the set of nodes that were contained in $I(w)$ before the deletion of (u, v) and are not contained in $I(w)$ anymore after this deletion. Similarly, let B denote the set of nodes that were contained in $O(w)$ before the deletion and are not contained in $O(w)$ anymore afterwards. The nodes in $A \cup B$ are exactly those nodes of C that are not contained in C_j. Let G' denote the subgraph of G induced by $A \cup B$. Then the components of G' are exactly the desired components $C_1, \dots, C_{j-1}, C_{j+1}, \dots C_k$. Note that the sets A and B are returned by the update-procedure of the SSR algorithms of w, which allows us to compute $A \cup B$. The graph G' can be constructed by iterating over all outgoing edges of $A \cup B$ and the components of G' can be found using a static SCC algorithm.

Analysis The correctness of the algorithm explained above is immediate. For the running time we will argue that, up to a $\log n$-factor, it is the same as the running time of the SSR algorithm. When the running time of SSR is of the form $\tilde{O}(m^\alpha n^\beta)$, our argument works when $\alpha \geq 1$ or $\beta \geq 1$. To understand the basic idea (for $\beta \geq 1$), consider the case that the graph decomposes into only two components C_1 and C_2 (with $n_1, n_2 \leq n$ and $m_1, m_2 \leq m$ being the corresponding number of nodes and edges). We know that one of the two components still contains the representative w. For this component we do not have to spawn a new decremental SSR algorithm. This is an advantage for large components as they have a high probability of containing the representative. The probability of w being contained in C_1 is n_1/n and it is n_2/n for being contained in C_2. Thus, the expected cost of the decomposition is $O(m_2^\alpha n_2^\beta n_1/n + m_1^\alpha n_1^\beta n_2/n)$. We charge this cost to the smaller

component, say C_1. As C_1 has n_1 nodes, the average cost we charge to every node in C_1 is $O(m_2^\alpha n_2^\beta/n + m_1^\alpha n_1^{\beta-1} n_2/n)$. This amounts to an average cost of $O(m^\alpha n^{\beta-1})$ per node. As we will charge each node only when the size of its component has halved, the total update time is $O(m^\alpha n^\beta \log n)$.

Theorem 4.2.15 (From SSR to SCC)**.** *If there is a decremental SSR algorithm with constant query time and a total update time of $O(nt_1(m,n) + mt_2(m,n))$ such that $t_1(m,n) \geq 1$, $t_2(m,n) \geq 1$, and $t_1(m,n)$ and $t_2(m,n)$ are non-decreasing*[7] *in m and n, then there exists a decremental SCC algorithm with constant query time and an expected total update time of $O((nt_1(m,n) + mt_2(m,n)) \log n)$.*

Proof. Let us first analyze the costs related to the decomposition of a component. Assume that the component C_0 decomposes into $C_1, \ldots, C_k$. Let n_i and m_i denote the number of nodes and edges in component i, respectively. Let m_i' denote the sum of the out-degrees of the nodes in C_i in the initial graph (i.e., before the first deletion). Note that m_i' is an upper bound on m_i. Furthermore we have $\sum_{i=1}^k n_i = n_0$, $\sum_{i=1}^k m_i \leq m_0$, and $\sum_{i=1}^k m_i' = m_0'$.

Assume that the representative w of C_0 is contained in C_i after the decomposition. First of all, for every $j \neq i$ we have to pay a cost of $O(n_j t_1(m_j, n_j) + m_j t_2(m_j, n_j))$ for initializing and updating the decremental SSR algorithm of the new representative of C_j. Second, we have to pay for computing the new components. This consists of three steps: (a) computing $A \cup B$, (b) computing G', and (c) computing the components of G'. Remember that $A \cup B$ is the union of A, the set of nodes that cannot reach w anymore after deleting (u, v), and B, the set of nodes that w cannot reach anymore after deleting (u, v). After deleting (u, v) the incoming SSR algorithm of w outputs A and the outgoing SSR algorithm of w outputs B. Thus, the cost of computing $A \cup B$ can be charged to the reachability algorithms of w (which have to output A and B anyway). The graph G' is the subgraph of G induced by the nodes in $A \cup B$. We construct G' by checking, for every node in $A \cup B$, which of its outgoing edges stay in $A \cup B$. This takes time $O(\sum_{j \neq i} m_j')$. Using Tarjan's linear time algorithm [120], we can compute the strongly connected components of G' in the same running time. As $m_j' \geq m_j$ the total cost of C_j is $O(n_j t_1(m_j, n_j) + m_j' t_2(m_j, n_j)) = O(n_j t_1(m, n) + m_j' t_2(m, n))$

By the random choice of the representatives, the probability that w is contained in C_i is n_i/n_0. Thus, the expected cost of the decomposition of C_0 is proportional to

$$\sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} (n_j t_1(m,n) + m_j' t_2(m,n)) = \sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} n_j t_1(m,n) + \sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} m_j' t_2(m,n)\,. \quad (4.4)$$

We analyze each of these terms individually.

[7] The technical assumption that $t_1(m,n)$ and $t_2(m,n)$ are non-decreasing in m and n is natural as usually the running time of an algorithm does not improve with increasing problem size.

Consider first the cost of $O(\sum_{i=1}^{k} n_i/n_0 \sum_{j\neq i} n_j t_1(m,n))$. For every pair i,j such that $i \neq j$ we have to pay a cost of $O(n_i n_j(t_1(m,n)+n)/n_0)$. If $n_i \leq n_j$ we charge this cost to the component C_i, otherwise we charge it to C_j (i.e., we always charge the cost to the smaller component). Note that the component to which we charge the cost has at most $n_0/2$ nodes (otherwise it would not be the smaller one). For a fixed component i, the total charge is proportional to

$$\sum_{j\neq i} \frac{n_i n_j(t_1(m,n)+n)}{n_0} = n_i(t_1(m,n)+n)\cdot\frac{\sum_{j\neq i} n_j}{n_0} \leq n_i(t_1(m,n)+n)\,.$$

We share this cost equally among the nodes in C_i and thus charge $O(t(m,n)+n)$ to every node in C_i. Every time we charge a node, the size of its component halves. Thus, every node is charged at most $\log n$ times and the total update time for the first term in Equation (4.4) is $O(nt(m,n)\log n)$.

Consider now the cost of $O(\sum_{i=1}^{k} n_i/n_0 \sum_{j\neq i} m'_j t_2(m,n))$. Note that

$$\sum_{i=1}^{k}\frac{n_i}{n_0}\sum_{j\neq i} m'_j = \sum_{i=1}^{k} m'_i \sum_{j\neq i}\frac{n_j}{n_0} = \sum_{i=1}^{k} m'_i\frac{n_0-n_i}{n_0}\,.$$

We now charge $m'_i(n_0-n_i)/n_0$ to every component C_i $(1\leq i\leq k)$. In particular we charge $(n_0-n_i)/n_0$ to every edge (u,v) of the initial graph such that $u\in C_i$. We now argue that in this way every edge is charged only $O(\log n)$ times, which will imply that the total update time for the second term in Equation (4.4) is $O(mt_2(m,n)\log n)$. Consider an edge (u,v) and the component containing u. We only charge the edge (u,v) when the component containing u decomposes. Let a_0 denote the initial number of nodes of this component and let a_p its number of nodes after the p-th decomposition. As argued above, we charge $(a_{p-1}-a_p)/a_{p-1}$ to (u,v) for the p-th decomposition. Thus, for q decompositions we charge $\sum_{1\leq p\leq q}(a_{p-1}-a_p)/a_{p-1}$. Now observe that

$$\sum_{1\leq p\leq q}\frac{a_{p-1}-a_p}{a_{p-1}} \leq \sum_{1\leq p\leq q}\sum_{i=0}^{a_{p-1}-a_p-1}\frac{1}{a_{p-1}} \leq \sum_{1\leq p\leq q}\sum_{i=0}^{a_{p-1}-a_p-1}\frac{1}{a_{p-1}-i}$$
$$= \sum_{1\leq p\leq q}\sum_{i=a_p+1}^{a_{p-1}}\frac{1}{i} = \sum_{i=a_q+1}^{a_0}\frac{1}{i}\,.$$

Since $a_0\leq n$, this harmonic series is bounded by $O(\log n)$.

Finally, we bound the initialization cost. Let $C_1,\ldots,C_k$ denote the initial components and let n_i and m_i denote the number of nodes and edges of component C_i, respectively. The initial components can be computed in time $O(m)$ with Tarjan's algorithm [120]. Furthermore, each component starts two decremental SSR algorithms and we have to pay for the total update time of these algorithms. This time is proportional to

$$\sum_{i=1}^{k}(n_i t_1(m_i,n_i)+t_2(m_i,n_i)) \leq t_1(m,n)\sum_{i=1}^{k} n_i + t_2(m,n)\sum_{i=1}^{k} m_i \leq nt_1(m,n)+mt_2(m,n)\,.$$

□

4.3 Single-Source Single-Sink Reachability

In this section we give an algorithm for maintaining a path from a source node s to a sink node t in a directed graph undergoing edge deletions, i.e., we solve the decremental stR problem. Using the reduction of Section 4.2.4, this implies an algorithm for the decremental single-source reachability (SSR) problem.

4.3.1 Algorithm Description

Our s-t reachability algorithm has a parameter $k \geq 1$ and for each $1 \leq i \leq k$ parameters $b_i \leq n$ and $c_i \leq n$. We determine suitable choices of these parameters in Section 4.3.3. For each $1 \leq i \leq k-1$, our choice will satisfy $b_i \geq b_{i+1}$ and $c_i \geq 2c_{i+1}$. We also set $b_{k+1} = 1$, $c_0 = n$, $c_{k+1} = 1$, and $h_i = n/c_i$ for all $0 \leq i \leq k+1$. Note that this implies $h_{i+1} \geq 2h_i$ for all $1 \leq i \leq k$. Intuitively, b_i and c_i are roughly the number of i-hubs and i-centers used by our algorithm and h_i is the hop range of the i-centers. In the algorithm we will often consider ordered pairs of centers of the form (x, y).

Initialization At the initialization (i.e., before the first deletion), our algorithm determines sets of nodes $B_1 \supseteq B_2 \supseteq \ldots \supseteq B_k$ and $C_0 \supseteq C_1 \supseteq \ldots \supseteq C_{k+1}$ as follows. For each $1 \leq i \leq k$, we sample each node of the graph with probability $ab_i \ln n/n$ and each edge with probability $ab_i \ln n/m$ (for a large enough constant a). The set B_i then consists of the sampled nodes and the endpoints of the sampled edges. We set $C_0 = V$ and $C_{k+1} = \{s, t\}$. For each $1 \leq i \leq k$, we sample each node of the graph with probability $ac_i \ln n/n$ (for a large enough constant a). The set C_i then consists of the sampled nodes together with the nodes in C_{i+1}. For every $1 \leq i \leq k$ we call the nodes in B_i i-hubs and for every $0 \leq i \leq k+1$ we call the nodes in C_i i-centers. Note that the number of i-hubs is $\tilde{O}(b_i)$ in expectation and the number of i-centers is $\tilde{O}(c_i)$ in expectation.

Data Structures Our algorithm uses the following data structures:

- For every i-hub z (with $1 \leq i \leq k$) an ES-tree up to depth $2h_i$ in G (*outgoing* tree of z) and an ES-tree up to depth $2h_i$ in the reverse graph of G (*incoming* tree of z)
- For every pair of i-centers (x, y) (with $1 \leq i \leq k$) the set of i-hubs from B_i linking x to y.
- For every pair of i-centers (x, y) (with $1 \leq i \leq k$) a set of nodes $Q(x, y, i)$, which is initially empty.
- For every pair of i-centers (x, y) (with $0 \leq i \leq k$) a list of pairs of $(i+1)$-centers called $(i+1)$-parents of (x, y).

- For every pair of i-centers (x, y) (with $1 \leq i \leq k+1$) a list of pairs of $(i-1)$-centers called $(i-1)$-children of (x, y).

Maintaining Hub Links For every $1 \leq i \leq k$, we say that an i-hub z *links* an i-center x to an i-center y if $d_G(x, z) \leq 2h_i$ and $d_G(z, y) \leq 2h_i$. The hub links of every pair of i-centers (x, y), for all $1 \leq i \leq k$, can be maintained as follows. After every edge deletion in the graph, we report the deletion to the ES-trees maintained by the hubs. Initially, and after each deletion, the level of a node v in the incoming (outgoing) tree of an i-hub z is at most $2h_i$ if and only if $d_G(v, z) \leq 2h_i$ ($d_G(z, v) \geq 2h_i$). Thus, we can check whether an i-hub z links an i-center x to an i-center y by summing up the levels of x and y in the incoming and outgoing ES-tree of z, respectively. Thus, for every priority i and every pair of i-centers (x, y) we can initialize the set of i-hubs linking x to y by iterating over all hubs. We can maintain these sets under the edge deletions in G as follows: Every time the level of some i-center x in the incoming ES-tree of some i-hub z exceeds $2h_i$, we iterate over all i-centers y and remove z from the set of hubs linking x to y. We proceed similarly if the level of some i-center y in the outgoing ES-tree of some i hub z exceeds $2h_i$. In this way, we can also generate, after every edge deletion, a list of pairs of i-centers (x, y) such that x is not linked to y by an i-hub anymore (but was linked to an i-hub before the deletion).

Main Algorithm Initially, the algorithm computes the shortest path π from s to t in G. It then determines a sequence of k-centers $v_1, \ldots, v_l$ on π. For each $1 \leq j \leq l-1$, the algorithm now tries to maintain a path from v_j to v_{j+1} under the edge deletions in G. If it fails to do so, it recomputes the shortest path from s to t in G. We call this a *refresh* operation.

For each $1 \leq j \leq l-1$, the path from v_j to v_{j+1} is maintained as follows. The first case is that there is a k-hub linking v_j to v_{j+1}. As long as such a hub exists, we know that there is a path from v_j to v_{j+1}. If there is no k-hub linking v_j to v_{j+1} anymore, the algorithm computes the path union $\mathscr{P}(v_j, v_{j+1}, 2h_k, G)$. It then recursively maintains the path from v_j to v_{j+1} in $G|\mathscr{P}(v_j, v_{j+1}, 2h_k, G)$ using $(k-1)$-centers and $(k-1)$-hubs. To keep track of this recursive hierarchy, we make, for every $1 \leq j \leq l-1$, each pair (v_j, v_{j+1}) an $(i-1)$-child of (x, y) and (x, y) an i-parent of (v_j, v_{j+1}). The end of the recursion is reached in layer 0, where the set of 0-centers consists of all nodes. The only paths between 0-centers the algorithm considers are the edges between them.

Algorithm 4.1 shows the pseudocode for these procedures. A crucial ingredient of our algorithm is the computation of path unions for pairs of centers. For every $1 \leq i \leq k$ and all i-centers (x, y) the path union from the last recomputation is stored in $Q(x, y, i)$. To simplify the formulation of the algorithm we set $Q(s, t, k+1) = V$ (which is consistent with our choice of $h_{k+1} = n$). Thus, the procedure Refresh$(s, t, k+1)$ simply checks whether there exists a path from s to t in G.

Algorithm 4.1: Algorithm for decremental s-t reachability

Procedure Refresh(x, y, i)
  Compute shortest path π from x to y in $G|Q(x, y, i)$
  if *length of* $\pi > h_i$ **then**
    if $i = k + 1$ **then**
      Stop and output "s cannot reach t"
    foreach $(i + 1)$-*parent* (x', y') *of* (x, y) **do**
      Refresh(x', y', $i + 1$)
  else
    RemoveChildren(x, y, i)
    Determine $(i - 1)$-centers $v_1, \ldots, v_l$ on π in order of appearance on π such that $v_1 = x$, $v_l = y$ and, for each $1 \le j \le l - 2$, v_{j+1} is the first $(i - 1)$-center following v_j on π at distance at least $h_{i-1}/2$ from v_j.
    // If $i = 1$ then the 0-centers are all nodes on π because $h_0 = 1$.
    foreach $1 \le j \le l - 1$ **do**
      Make (v_j, v_{j+1}) an $(i - 1)$-child of (x, y) and (x, y) an i-parent of (v_j, v_{j+1})
      if v_{j+1} *not linked to* v_j *by an* i-*hub* **then**
        ComputePathUnion(v_j, v_{j+1}, $i - 1$)
        Refresh(v_j, v_{j+1}, $i - 1$)

Procedure ComputePathUnion(x, y, i)
  if $Q(x, y, i) = \emptyset$ **then**
    Let (x', y') be any $(i + 1)$-parent of (x, y)
    $Q(x, y, i) \leftarrow \mathscr{P}(x, y, 2h_i, G|Q(x', y', i + 1))$
  else
    $Q(x, y, i) \leftarrow \mathscr{P}(x, y, 2h_i, G|Q(x, y, i))$

Procedure RemoveChildren(x, y, i)
  foreach $(i - 1)$-*child* (x', y') *of* (x, y) **do**
    Remove (x', y') from $(i - 1)$-children of (x, y) and remove (x, y) from i-parents of (x', y')
    if (x', y') *has no* i-*parents anymore and* $i \ge 1$ **then**
      RemoveChildren(x', y', $i - 1$)

Procedure Initialize()
  foreach $1 \le i \le k$ *and all* i-*centers* (x, y) **do** $Q(x, y, i) \leftarrow \emptyset$
  $Q(s, t, k + 1) \leftarrow V$
  Refresh(s, t, $k + 1$)

Procedure Delete(u, v)
  foreach 1-*parent* (x, y) *of* (u, v) **do**
    Refresh(x, y, 1)
  For every $1 \le i \le k$ and every pair of i-centers (x, y) that is not linked by an i-hub anymore: Refresh(x, y, i)

4.3.2 Correctness

It is obvious that the algorithm correctly maintains, for all pairs of i-centers (x, y), the set of i-hubs linking x to y. Furthermore, the algorithm only stops if it has correctly detected that there is no path from s to t in the current graph G. Thus, it remains to show that, as long as the algorithm does not stop, there there is a path from s to t. We say that a pair of i-centers (x, y) is *active* if it has at least one $(i + 1)$-parent. We further define the pair (s, t), the only pair of $(k + 1)$-centers, to be active.

Lemma 4.3.1. *After finishing the delete-operation, for every $0 \leq i \leq k + 1$ and every active pair of i-centers (x, y), there is a path from x to y in the current graph G.*

Proof. The proof is by induction on i. If y is linked to x by some i-hub z, we know that in G there is a path from x to z as well as a path from z to y. Their concatenation is a path from x to y in G.

Consider now the case that there is no i-hub linking x to y. If $i = 0$, we know that there is an edge from x to y in G as otherwise the pair (x, y) would not be active. If $i \geq 1$, let $(v_1, v_2), (v_2, v_3), \ldots, (v_{l-1}, v_l)$ with $v_1 = x$ and $v_l = y$ denote the $(i + 1)$-children of (x, y) found at the time of the last refresh of (x, y). Note that all these children are active since (x, y) is their i-parent. Thus, by the induction hypothesis, we know that there is a path from v_j to v_{j+1} in G for all $1 \leq j \leq l - 1$. The concatenation of these paths gives a path from x to y in G, as desired. □

4.3.3 Running Time Analysis

Important Properties In the running time analysis we will need some properties of the path unions computed by our algorithm.

Lemma 4.3.2. *Let (x', y') be pair of nodes such that $d_G(x', y') \leq h_{i+1}$ and let (x, y) be a pair of nodes such that x and y lie on a shortest path from x to y in G (and x appears before v on this shortest path). Then $\mathcal{P}(x, y, 2h_i, G) \subseteq \mathcal{P}(x', y', 2h_{i+1}, G)$.*

Proof. Let $v \in \mathcal{P}(x, y, 2h_i, G)$. We first apply the triangle inequality:

$$d_G(x', v) + d_G(v, y') \leq d_G(x', x) + d_G(x, v) + d_G(v, y) + d_G(y, y').$$

Since x and y (in this order) lie on the shortest path from x' to y', we have $d_G(x', x) + d_G(y, y') \leq d_G(x', y')$ and by our assumption we have $d_G(x', y') \leq h_{i+1}$. Since $v \in \mathcal{P}(x, y, 2h_i, G)$, we have $d_G(x, v)+d_G(v, y) \leq 2h_i$ As $2h_i \leq h_{i+1}$ by our assumptions on the parameters we get

$$d_G(x', v) + d_G(v, y') \leq h_{i+1} + 2h_i \leq h_{i+1} + h_{i+1} = 2h_{i+1}$$

which implies that $v \in \mathcal{P}(x', y', 2h_{i+1}, G)$. □

Lemma 4.3.3. *For every $1 \leq i \leq k$ and every pair of i-centers (x, y), if $Q(x, y, i) \neq \emptyset$, then $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x, y, i)$.*

Proof. The proof is by induction on i. We first argue that it is sufficient to show that after the first initialization of $Q(x, y, i)$ in Line 19 we have $Q(x, y, i) = \mathscr{P}(x, y, 2h_i, G)$. After the initialization, and before $Q(x, y, i)$ is recomputed, $Q(x, y, i)$ might be "outdated" due to deletions in G, i.e., not be equal to $\mathscr{P}(x, y, 2h_i, G)$ anymore. However, since $\mathscr{P}(x, y, 2h_i, G)$ only "loses" nodes through the deletions in G, it is still the case that $\mathscr{P}(x, y, 2h_i, G) \subseteq Q(x, y, i) \subseteq V$. Thus, this is also true directly before $Q(x, y, i)$ is recomputed for the first time in Line 21. There $Q(x, y, i)$ is updated to $\mathscr{P}(x, y, 2h_i, G|Q(x, y, i))$. By Lemma 4.2.9 we know that $\mathscr{P}(x, y, 2h_i, G|Q(x, y, i)) = \mathscr{P}(x, y, 2h_i, G)$, i.e., after the recomputation, $Q(x, y, i)$ is equal to $\mathscr{P}(x, y, 2h_i, G)$ again. By repeating this argument, we get that $\mathscr{P}(x, y, 2h_i, G) \subseteq Q(x, y, i) \subseteq V$.

We now show that after the first initialization of $Q(x, y, i)$ in Line 19 we have $Q(x, y, i) = \mathscr{P}(x, y, 2h_i, G)$. Let (x', y') be the $(i + 1)$-parent of (x, y) used in the initialization. We will show that $\mathscr{P}(x, y, 2h_i, G|Q(x', y', i + 1)) = \mathscr{P}(x, y, 2h_i, G)$. For $i = k$ the claim is trivially true because (s, t) is the only $(k + 1)$-parent of (x, y) and $Q(s, t, k + 1) = V$. For $i \leq k - 1$, we know by the induction hypothesis that $\mathscr{P}(x', y', 2h_{i+1}, G) \subseteq Q(x', y', i + 1)$. We also know that x and y lie on the shortest path from x' to y' in $G|Q(x', y', i + 1)$ such that x precedes y on this shortest path and the path has length at most h_i. As $G|Q(x', y', i+1)$ is a subgraph of G, this path is also contained in G. Thus, $d_G(x', x) + d_G(y, y') \leq h_i$. By Lemma 4.3.2 this implies that $\mathscr{P}(x, y, 2h_iG) \subseteq \mathscr{P}(x', y', 2h_{i+1}, G)$. It follows that $\mathscr{P}(x, y, 2h_i, G) \subseteq Q(x', y', i + 1)$. Thus, by Lemma 4.2.9, we have $\mathscr{P}(x, y, 2h_i, G|Q(x', y', i + 1)) = \mathscr{P}(x, y, 2h_i, G)$ as desired. □

Lemma 4.3.4. *Let (x, y) be an active pair of i-centers and let (x', y') be an $(i+1)$-parent of (x, y). Then $Q(x, y, i) \subseteq Q(x', y', i + 1)$.*

Proof. Consider the point in time when (x', y') becomes an $(i + 1)$-parent of (x, y). This can only happen in the else-branch of Refresh$(x, y, i+1)$. Then we know that x and y lie on a path from x' to y' of length at most h_{i+1} in $G|Q(x', y', i + 1)$ such that x precedes y on this path and thus $d_G(x', x) + d_G(y, y') \leq h_{i+1}$. From Lemma 4.3.2 it follows that $\mathscr{P}(x, y, 2h_i, G) \subseteq \mathscr{P}(x', y', 2h_{i+1}, G)$. Furthermore, $\mathscr{P}(x', y', 2h_{i+1}, G) \subseteq Q(x', y', i+1)$ by Lemma 4.3.3. As the path-union of (x, y) is recomputed when (x, y) becomes an i-child of (x', y'), we also have $Q(x, y, i) = \mathscr{P}(x, y, 2h_i, G)$. Therefore we get that $Q(x, y, i) \subseteq Q(x', y', i + 1)$. Furthermore, every time the algorithm recomputes $Q(x', y', i + 1)$, (x', y') will stop being an $(i + 1)$-parent of (x, y). It might immediately become in $(i + 1)$-parent again and in this case our argument above applies again. □

Lemma 4.3.5. *For every $1 \leq i \leq k + 1$ and every pair of i-centers (x, y), we have that if $d_{G|Q(x,y,i)}(x, y) > h_i$, then $d_G(x, y) > h_i$.*

Proof. We show that $d_G(x, y) \leq h_i$ implies $d_{G|Q(x,y,i)}(x, y) \leq h_i$. If $d_G(x, y) \leq h_i$, then also $d_{G|\mathscr{P}(x,y,2h_i,G)}(x, y) \leq h_i$ as $\mathscr{P}(x, y, 2h_i, G)$ contains all paths from x to y in G of length at most $2h_i$. Since $\mathscr{P}(x, y, 2h_i, G) \subseteq Q(x, y, i)$ by Lemma 4.3.3, we have $d_{G|Q(x,y,i)}(x, y) \leq d_{G|\mathscr{P}(x,y,2h_iG)}(x, y)$. It follows that $d_{G|Q(x,y,i)}(x, y) \leq h_i$. □

Lemma 4.3.6. *For every $1 \leq i \leq k+1$ and every pair of i-centers (x, y) the graph $G|Q(x, y, i)$ has at most $\min(m/b_i, n^2/b_i^2)$ edges whp.*

Proof. As no edges are ever added to G, we only have to argue that the claim is true at the first initialization of $Q(x, y, i)$ in Line 19, where $Q(x, y, i)$ is equal to $\mathscr{P}(x, y, 2h_i, G)$ (see proof of Lemma 4.3.4). We argue that at that time $G|\mathscr{P}(x, y, 2h_i, G)$ as at most $\min(m/b_i, n^2/b_i^2)$ edges. Note to this end it is sufficient to show that $G|\mathscr{P}(x, y, 2h_i, G)$ has at most n/b_i nodes and m/b_i edges.

Suppose that $\mathscr{P}(x, y, 2h_i, G)$ contains more than n/b_i nodes. Then, by the random sampling of i-hubs, one of these nodes, say v, would have been sampled whp while determining the set B_i at the initialization by Lemma 1.3.2, making v an i-hub. If $G|\mathscr{P}(x, y, 2h_i, G)$ contains more than m/b_i edges, then one of these edges, say (u, v), would have been sampled whp while determining the set B_i at the initialization by Lemma 1.3.2, making both u and v i-hubs.

In both cases, $\mathscr{P}(x, y, 2h_i, G)$ contains some i-hub v, for which $d_G(x, v)+d_G(v, y) \leq 2h_i$ by Lemma 4.2.7. But this means that x is linked to y by the i-hub v and the algorithm would not have executed Line 19, which contradicts our assumption. □

Lemma 4.3.7. *Let $1 \leq i \leq k$ and consider a pair of active $(i+1)$-centers (x', y') and their i-children $(x_j, y_j)_{1\leq j\leq l}$ (which are active i-centers). Then, for every node v, there are at most $q = 8$ pairs of i-children (x_j, y_j) of (x', y') such that $v \in Q(x_j, y_j, i)$.*

Proof. We show that at the last time the algorithm has called Refresh$(x', y', i+1)$ (where it determined the current i-children of (x', y')) there are at most $q = 8$ pairs of i-children (x_j, y_j) of (x', y') such that $v \in \mathscr{P}(x_j, y_j, 2h_i, G)$ as, for each $1 \leq i \leq l$, $Q(x_j, y_j, i)$ (even if initialized later) will always be a subset of this set.

Suppose that v is contained in $q > 8$ path unions $\mathscr{P}(x_j, y_j, 2h_i, G)$ of i-children (x_j, y_j) of (x', y'). Let $j_1, \ldots, j_q$ be the corresponding indices and assume without loss of generality that $j_1 < j_2 \ldots < j_q$. The children of (x', y') all lie on the shortest path from x' to y' and by the way we have selected them we even more have $d_G(x_j, y_j) \geq h_i/2$ for all $1 \leq j \leq l$ and thus

$$d_G(x_{j_1}, y_{j_q}) \geq \sum_{j\in\{j_1,\ldots,j_q\}} d_G(x_j, y_j) \geq \sum_{j\in\{j_1,\ldots,j_q\}} h_i/2 = qh_i/2\,. \tag{4.5}$$

Furthermore, since $v \in \mathscr{P}(x_{j_1}, y_{j_1}, 2h_i, G)$, v lies on a shortest path from x_{j_1} to y_{j_1} of length at most h_i in G and thus $d_G(x_{j_1}, v) \leq 2h_i$. By the same argument we have $d_G(v, y_{j_q}) \leq 2h_i$. We now have

$$d_G(x_{j_1}, y_{j_q}) \leq d_G(x_{j_1}, v) + d_G(v, y_{j_q}) \leq 4h_i \tag{4.6}$$

by the triangle inequality. Observe that the upper bound 4.6 and the lower bound 4.5 contradict each other for $q > 8$ and thus $q \leq 8$. □

Lemma 4.3.8. *For every $1 \leq i \leq k+1$ and every node v, there are at most q^{k-i+1} pairs of active i-centers (x, y) such that $v \in Q(x, y, i)$.*

Proof. The proof is by induction on i. The base case is $i = k + 1$ and is trivially true because the only pair of $(k + 1)$-centers is the pair (s, t). Consider now the case $1 \leq i \leq k$ and fix some node v. Consider a pair (x, y) of active i-centers such that $v \in Q(x, y, i)$. Let (x', y') be a pair of $(i + 1)$-centers that is an $(i + 1)$-parent of (x, y) (such a parent must exist because otherwise (x, y) would not be active). Since $Q(x, y, i) \subseteq Q(x', y', i + 1)$ by Lemma 4.3.4, $v \in Q(x', y', i + 1)$. Thus, $v \in Q(x, y, i)$ only if there is an $(i + 1)$-parent (x', y') such that $v \in Q(x', y', i + 1)$.

By the induction hypothesis, the number of pairs of active $(i + 1)$-centers (x', y') such that $v \in Q(x', y', i + 1)$ is at most $q^{k-(i+1)+1} = q^{k-i}$. Let (x', y') be such a pair of $(i + 1)$-centers. By Lemma 4.3.7, the number of i-children (x, y) of (x', y') such that $v \in Q(x, y, i)$ is at most q. Therefore the total number of active pairs of i-centers (x, y) such that $v \in Q(x, y, i)$ is at most $q \cdot q^{k-i} = q^{k-i+1}$ as desired. □

Lemma 4.3.9. *For every $0 \leq i \leq k$, each active pair of i-centers has at most $q^{k-i} \leq q^k$ $(i + 1)$-parents.*

Proof. Consider any active pair of i-centers (x, y) and fix some node $v \in Q(x, y, i)$. Let the pair of $(i + 1)$-centers (x', y') be an $(i + 1)$-parent of (x, y). By Lemma 4.3.4 $Q(x, y, i) \subseteq Q(x', y', i + 1)$. Thus, $v \in Q(x', y', i + 1)$ for every $(i + 1)$-parent of (x, y). If (x, y) had more than q^{k-i} $(i + 1)$-parents, then v would be contained in more than $q^{k-i} = q^{k-(i+1)+1}$ path unions $Q(x', y', i + 1)$ of $(i + 1)$-parents (x', y'), contradicting Lemma 4.3.8. □

Maintaining Hub Links For every i-hub, we maintain an incoming and an outgoing ES-tree of depth $2h_i = 2n/c_i$, which takes time $O(mn/c_i)$. As there are $\tilde{O}(b_i)$ hubs of priority i, the total time needed for maintaining all these ES-trees is $\tilde{O}(\sum_{1 \leq i \leq k} b_i mn/c_i)$. For every pair of i-centers (x, y), the list of hubs linking x to y is initialized by iterating over all i-hubs. As there are $\tilde{O}(c_i)$ i-centers and $\tilde{O}(b_i)$ i-hubs, this takes time $\tilde{O}(\sum_{1 \leq i \leq k} b_i c_i^2)$. Every time the level of an i-center x in the ES-tree of an i-hub z exceeds $2h_i$, we have to remove z from the set of hub links for every possible partner y of x. As this event can occur only once for every i-center x and every i-hub z over the course of the algorithm, maintaining the sets of linking hubs takes time $\tilde{O}(\sum_{1 \leq i \leq k} b_i c_i^2)$.

Computing Path Unions We now argue about the time needed for maintaining the sets $Q(x, y, i)$ for all pairs of i-centers (x, y) and each $1 \leq i \leq k$. We will show that we can pay for this cost by charging $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ to every pair of i-centers and $O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$ to every refresh operation of the form Refresh$(x', y', i + 1)$ for some pair of $(i + 1)$-centers (x', y'). Note that when $i = k$ then $\min(m/b_{k+1}, n^2/b_{k+1}^2) = m$ since we have set $b_{k+1} = 1$.

Fixing some pair of i-centers (x, y), we first bound the cost for the first initialization of $Q(x, y, i)$, as performed in Line 19 of Algorithm 4.1. There we have to compute $\mathcal{P}(x, y, 2h_i, G|Q(x', y', i+1))$, where (x', y') is an $(i+1)$-parent of (x, y). By Lemma 4.2.7 this takes time proportional to the number of edges in $G|Q(x', y', i + 1)$.

If $i = k$, then $Q(x', y', i+1) = V$ (and actually $x' = s$ and $y' = t$) and thus computing $\mathscr{P}(x, y, 2h_i, G)$ for all pairs of k-centers (x, y) takes time $\tilde{O}(c_k^2 m)$. If $1 \leq i \leq k-1$, then $G|Q(x', y', i+1)$ has $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ edges by Lemma 4.3.6. Thus, the first computation of $Q(x, y, i)$ for all pairs of i-centers (x, y) takes time $\tilde{O}(c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2))$.

Now consider the cost of computing $Q(x, y, i)$ after it has already been initialized for the first time. Let $Q'(x, y, i) = \mathscr{P}(x, y, 2h_i, G|Q(x, y, i))$ denote the updated path union as computed in Line 21 of Algorithm 4.1. The cost of computing $Q'(x, y, i)$ is proportional to $|E(Q(x, y, i))|$, the number of edges in $G|Q(x, y, i)$ before the recomputation. Since $Q'(x, y, i) \subseteq Q(x, y, i)$, we have $E(Q(x, y, i)) = E(Q'(x, y, i)) \cup E(Q(x, y, i)) \setminus E(Q'(x, y, i))$. Note that $Q'(x, y, i)$ is equal to $\mathscr{P}(x, y, 2h_i, G)$. We pay for this cost by charging $O(|E(Q(x, y, i)) \setminus E(Q'(x, y, i))|)$ to the pair (x, y) and $|E(\mathscr{P}(x, y, 2h_i, G))|$ to the refresh operation on the $(i+1)$-parent of (x, y) which causes the recomputation.

As the initial size of $G|Q(x, y, i)$ is $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ and we only charge edges to the pair (x, y) that will never be contained in $G|Q(x, y, i)$ anymore, the total time charged to (x, y) is $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$, which results in a cost of $O(\sum_{1 \leq i \leq k-1} c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2) + c_k^2 m)$ over all path union computations. It remains to bound the total cost charged to each refresh operation Refresh$(x', y', i+1)$ for some pair of $(i+1)$-centers (x', y'). Below we will then separately analyze the total cost of the refresh operations. During the refresh operation we recompute $Q(x, y, i)$ for every i-child (x, y) of (x', y'), for which we have to pay $O(|E(\mathscr{P}(x, y, 2h_i, G))|)$ per child. By Lemma 4.3.4 $Q(x, y, i) \subseteq Q(x', y', i+1)$ for every i-child (x, y) of (x', y'). By Lemma 4.3.8, each node is contained in at most q^k path unions of i-children of (x', y'). Therefore every edge of $G|Q(x', y', i+1)$ is contained in the subgraphs of G induced by the i-path union of at most q^k i-children of (x', y'). Thus, the total cost charged to the refresh operation is $O(q^k|E(Q(x', y', i+1)|)$, which is $O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$.

Cost of Refresh Excluding the recursive calls, each refresh operation of the form Refresh(x, y, i), where $1 \leq i \leq k+1$ and (x, y) is a pair of i-centers, is dominated by two costs: (1) the time needed for computing the shortest path from x to y in $G|Q(x, y, i)$ and (2) the time needed for computing $Q(x', y', i-1)$ for every $(i-1)$-child (x', y') of (x, y) if $i \geq 2$. As $G|Q(x, y, i)$ has at most $\min(m/b_{i+1}, n^2/b_{i+1}^2)$ edges by Lemma 4.3.6, computing the shortest path takes time $\tilde{O}(\min(m/b_{i+1}, n^2/b_{i+1}^2))$. We have argued above that to pay for step (2) the time we charge to that particular refresh is $O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$. It remains to analyze how often the refresh operation is called.

We say that a pair of i-centers (x, y) (for $0 \leq i \leq k$) causes a refresh if the algorithm calls Refresh $(x', y', i+1)$ (in line 7), where (x', y') is an $(i+1)$-parent of (x, y), after detecting that the distance from x to y in $G|Q(x, y, i)$ is more than h_i. If a fixed pair of i-centers (x, y) causes a refresh after some deletion in the graph, it will do so for each of its $(i+1)$-parents. By Lemma 4.3.9 the number of $(i+1)$-parents is at

most q^k. Note that each pair of i-centers (x, y) will only cause these q^k refreshes of its parents once. At the time the algorithm makes (x, y) the child of some $(i+1)$-center we have $d_G(x, y) \leq h_i$ whp as by the initial random sampling of i-centers, every shortest path consisting of $h_i/2 - 1$ edges contains an i-center whp (Lemma 1.3.2). Furthermore, the pair (x, y) will never cause a refresh anymore in the future as the refresh implies that $d_G(x, y) > h_i$ by Lemma 4.3.5 and thus (x, y) will never be active anymore.

Now whenever we refresh a pair of $(i+1)$-centers (x', y'), we charge the running time of $\tilde{O}(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$ to the i-child (x, y) causing the refresh. By the argument above, each pair of i-centers will be charged at most q^k times. Thus, the total time needed for all refresh operations on pairs of $(i+1)$-centers over the course of the algorithm is $\tilde{O}(q^{2k} c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2))$ if $i \geq 1$. For $i = 0$, we can bound this by $\tilde{O}(q^{2k} m \cdot \min(m/b_1, n^2/b_1^2))$ because every node is a 0-center and thus a pair of 0-centers (x, y) can only be active if the graph contains the edge (x, y).

Total Running Time Putting everything together, the total running time of our algorithm using k layers is

$$\tilde{O}\left(\sum_{1\leq i\leq k} b_i c_i^2 + \sum_{1\leq i\leq k} \frac{b_i mn}{c_i} + q^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2}\right) + \sum_{1\leq i\leq k-1} q^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2}\right) + q^{2k} c_k^2 m\right).$$

We first balance the terms to obtain a running time of $\tilde{O}(m^{5/4} n^{1/2})$. We achieve this by setting the parameters to $k = \lceil \log\log m \rceil$ and, for every $1 \leq i \leq k$,

$$b_i = \frac{m^{\frac{3\cdot(2^k - 2^{(i-1)})}{2^{(k+2)}-3}}}{n^{\frac{2^{(k+1)}-2^i}{2^{(k+2)}-3}}}$$
$$c_i = 2^{k-i} m^{\frac{2^{(k+1)}-3\cdot 2^{(i-1)}}{2^{(k+2)}-3}} n^{\frac{2^i-1}{2^{(k+2)}-3}}.$$

With this choice of the parameters we get (for all $1 \leq i \leq k$), $b_i \geq b_{i+1}$, $c_i \geq 2c_{i+1}$,

$$\frac{b_i mn}{c_i} \leq \frac{m^2}{b_1} = m^{\frac{5\cdot 2^k - 3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}},$$
$$\frac{c_i^2 m}{b_{i+1}} \leq c_k^2 m \leq 2^{2k} m^{\frac{5\cdot 2^k - 3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}}, \text{ and}$$
$$b_i c_i^2 \leq 2^{2k} \frac{m^{\frac{7\cdot 2^k - 9\cdot 2^{(i-1)}}{2^{(k+2)}-3}}}{n^{\frac{2^{(k+1)}-3\cdot 2^i+2}{2^{(k+2)}-3}}} \leq 2^{2k} m^{\frac{5\cdot 2^k - 3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}},$$

where the last inequality holds due to $m \le n^2$ and $i \ge 0$. Thus, the total update time is

$$\tilde{O}\left(k2^{2k}q^{2k}m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}}n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}}\right).$$

Now observe that

$$m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}}n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}} = m^{\frac{5}{4}+\frac{3}{4}\cdot\frac{1}{2^{(k+2)}-3}}n^{\frac{1}{2}-\frac{1}{2}\frac{1}{2^{(k+2)}-3}} \le m^{\frac{5}{4}+\frac{3}{4}\cdot\frac{1}{2^{(k+2)}-3}}n^{\frac{1}{2}}.$$

By our choice of $k = \lceil \log\log m \rceil$ we have

$$m^{\frac{3}{4}\cdot\frac{1}{2^{(k+2)}-3}} \le m^{\frac{1}{2^k}} \le m^{\frac{1}{2^{\log\log m}}} = m^{\frac{1}{\log m}} = 2.$$

Thus, the running time of our algorithm is $\tilde{O}(kq^{2k}m^{5/4}n^{1/2})$. With $k = \lceil \log\log m \rceil$ and $q = 8$ this is $\tilde{O}(m^{5/4}n^{1/2})$.

We now balance the terms to obtain a running time of $O(m^{4/3}n^{2/3+o(1)})$. We achieve this by setting the parameters to $k = \lfloor \sqrt{\log n/\log q} \rfloor$ and, for every $1 \le i \le k$,

$$b_i = \frac{n^{(k+2i-1)/(3k+1)}}{m^{(4i-k-3)/(6k+2)}}$$
$$c_i = 2^{k-i}n^{2i/(3k+1)}m^{(3k+1-4i)/(6k+2)}.$$

With this choice of the parameters we get (for all $1 \le i \le k$), $b_i \ge b_{i+1}$, $c_i \ge 2c_{i+1}$,

$$\frac{b_i mn}{c_i} \le \frac{m^2}{b_1} = n^{4k/(3k+1)}m^{(2k+2)/(3k+1)},$$
$$\frac{c_i^2 m}{b_{i+1}} \le c_k^2 m \le 2^{2k}n^{4k/(3k+1)}m^{(2k+2)/(3k+1)},$$

and

$$\begin{aligned} b_i c_i^2 &\le 2^{2k}n^{(k+6i-1)/(3k+1)}m^{(7k+5-12i)/(6k+2)} \\ &= 2^{2k}n^{(k-1)/(3k+1)}m^{(3k+1)/(6k+2)}m^{(2k+2)/(3k+1)}n^{6i/(3k+1)}m^{-6i/(3k+1)} \\ &\le 2^{2k}n^{4k/(3k+1)}m^{(2k+2)/(3k+1)}, \end{aligned}$$

where the last inequality holds due to $n \le m$ and $m \le n^2$. Thus, the total update time is

$$\tilde{O}\left(k2^{2k}q^{2k}n^{4k/(3k+1)}m^{(2k+2)/(3k+1)}\right).$$

Now observe that

$$m^{\frac{2k+2}{3k+1}}n^{\frac{4k}{3k+1}} = m^{\frac{2}{3}+\frac{4}{3}\cdot\frac{1}{3k+1}}n^{\frac{4}{3}-\frac{4}{3}\cdot\frac{1}{3k+1}} \le m^{\frac{2}{3}+\frac{4}{3}\cdot\frac{1}{3k+1}}n^{\frac{4}{3}} \le m^{\frac{2}{3}+\frac{1}{k}}n^{\frac{4}{3}}$$

By our choice of $k = \lfloor \sqrt{\log n/\log q} \rfloor$ we have $2^k \le q^k \le n^{1/k}$. Since $k \le \log n$ and $q = 8$ we thus obtain a total update time of $O(m^{2/3}n^{4/3+O(1/\sqrt{\log n})}) = O(m^{2/3}n^{4/3+o(1)})$.

Theorem 4.3.10. *There is a decremental stR algorithm with constant query time and expected update time*

$$\tilde{O}(\min(m^{5/4}n^{1/2}, m^{2/3}n^{4/3+o(1)})) = O(mn^{6/7+o(1)})$$

that is correct with high probability against an oblivious adversary.

4.3.4 Extension to Single-Source Reachability

The algorithm above maintains reachability from a single source s to a single sink t. We can easily modify the algorithm to maintain reachability for a set of source-sink pairs $(s_i, t_i)_{1\le i\le p}$. We simply run p instances of the algorithm, each with a different source-sink pair. Note however that the algorithm can use the same set of hubs and centers for all instances. Thus, the cost of maintaining the ES-trees does *not* have to be multiplied by p. We therefore get a total running time of

$$\tilde{O}\left(\sum_{1\le i\le k} b_i c_i^2 + \sum_{1\le i\le k} \frac{b_i mn}{c_i} + pq^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2}\right) + \sum_{1\le i\le k-1} pq^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2}\right) + pq^{2k} c_k^2 m\right).$$

By setting

$$b_i = \frac{m^{\frac{3\cdot(2^k-2^{(i-1)})}{2^{(k+2)}-3}} p^{\frac{2^{(k+1)}-2^i}{2^{(k+2)}-3}}}{n^{\frac{2^{(k+1)}-2^i}{2^{(k+2)}-3}}}$$

$$c_i = \frac{m^{\frac{2^{(k+1)}-3\cdot 2^{(i-1)}}{2^{(k+2)}-3}} n^{\frac{2^i-1}{2^{(k+2)}-3}}}{p^{\frac{2^i-1}{2^{(k+2)}-3}}}$$

we get

$$pc_k^2 m = pm^2/b_1 = b_i mn/c_i = pc_i^2 m/b_{i+1} = m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}} p^{\frac{2^{(k+1)}-1}{2^{(k+2)}-3}}.$$

$$\frac{b_i mn}{c_i} \le \frac{pm^2}{b_1} = p^{\frac{2^{(k+1)}-1}{2^{(k+2)}-3}} m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}},$$

$$\frac{pc_i^2 m}{b_{i+1}} \le pc_k^2 m \le 2^{2k} p^{\frac{2^{(k+1)}-1}{2^{(k+2)}-3}} m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}}, \text{ and}$$

$$b_i c_i^2 \le 2^{2k} \frac{p^{\frac{2^{(k+1)}-3\cdot 2^i+2}{2^{(k+2)}-3}} m^{\frac{7\cdot 2^k-9\cdot 2^{(i-1)}}{2^{(k+2)}-3}}}{n^{\frac{2^{(k+1)}-3\cdot 2^i+2}{2^{(k+2)}-3}}} \le 2^{2k} p^{\frac{2^{(k+1)}-1}{2^{(k+2)}-3}} m^{\frac{5\cdot 2^k-3}{2^{(k+2)}-3}} n^{\frac{2^{(k+1)}-2}{2^{(k+2)}-3}},$$

and by setting

$$b_i = \frac{p^{(k+1-i)/(3k+1)} n^{(k+2i-1)/(3k+1)}}{m^{(4i-k-3)/(6k+2)}}$$

$$c_i = \frac{n^{2i/(3k+1)} m^{(3k+1-4i)/(6k+2)}}{p^{i/(3k+1)}}.$$

we get

$$\frac{b_i mn}{c_i} \le \frac{pm^2}{b_1} = p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)},$$
$$\frac{pc_i^2 m}{b_{i+1}} \le pc_k^2 m \le 2^{2k} p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)},$$

and

$$\begin{aligned} b_i c_i^2 &= 2^{2k} p^{(k+1-3i)/(3k+1)} n^{(k+6i-1)/(3k+1)} m^{(7k+5-12i)/(6k+2)} \\ &\le 2^{2k} p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)}. \end{aligned}$$

By the same choices of k as in the s-t reachability algorithm above we get a running times of $\tilde{O}(p^{1/2} m^{5/4} n^{1/2})$ and $O(p^{1/3} m^{2/3} n^{4/3+o(1)})$, respectively, for maintaining reachability between p source-sink pairs.

Corollary 4.3.11. *There is a decremental algorithm for maintaining reachability of p source-sink pairs with constant query time and expected update time*

$$\tilde{O}(\min(p^{1/2} m^{5/4} n^{1/2}, p^{1/3} m^{2/3} n^{4/3+o(1)}))$$

that is correct with high probability against an oblivious adversary.

Using the reduction of Theorem 4.2.14 this immediately implies single-source reachability algorithm with a total update time of $\tilde{O}(m^{7/6} n^{2/3})$ (we balance the terms $p^{1/2} m^{5/4} n^{1/2}$ and mn/p by setting $p = n^{1/3}/m^{1/6}$) and $O(m^{3/4} n^{5/4+o(1)})$ (balance $p^{1/3} m^{2/3} n^{4/3}$ and mn/p by setting $p = m^{1/4}/n^{1/4}$).

Corollary 4.3.12. *There is a decremental SSR algorithm with constant query time and expected update time*

$$\tilde{O}(\min(m^{7/6} n^{2/3}, m^{3/4} n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$$

that is correct with high probability against an oblivious adversary.

Furthermore, the reduction of Theorem 4.2.15 gives a decremental algorithm for maintaining strongly connected components.

Corollary 4.3.13. *There is a decremental SCC algorithm with constant query time and expected update time*

$$\tilde{O}(\min(m^{7/6} n^{2/3}, m^{3/4} n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$$

that is correct with high probability against an oblivious adversary.

4.4 Approximate Shortest Path

In the following we assume that G is a weighted graph with integer edge weights from 1 to W undergoing edge deletions. We are given some parameter $0 < \epsilon \leq 1$ and our goal is to maintain $(1+\epsilon)$-approximate shortest paths. Similar to the reachability algorithm, we first give an algorithm for maintaining a $(1+\epsilon)$ shortest path from a given source s to a given sink t. We then extend it to an algorithm for maintaining $(1+\epsilon)$-approximate shortest paths from a fixed source node to all other nodes. We give an algorithm that maintains a $(1+\epsilon)^{2k+2}$-approximate shortest path from s to t. Note that by running the whole algorithm with $\epsilon' = \epsilon/(4k+2)$ instead of ϵ we obtain a $(1+\epsilon)$-approximate shortest path.

4.4.1 Preliminaries

The first new aspect for approximate shortest paths, compared to reachability, is that we maintain, for each $1 \leq i \leq k+1$ and every pair of i-centers (x, y), an index $r(x, y, i)$ such that $d^h_G(x, y) \geq (1+\epsilon)^{r(x,y,i)}$ and, whenever (x, y) is active, $d_G(x, y) \leq (1+\epsilon)^{r(x,y,i)+2i}$. Thus, $r(x, y, i)$ indicates the current range of the distance from x to y. Roughly speaking, the algorithm will maintain a sequence of centers and for each pair of consecutive centers (x, y) a path from x to y of weight corresponding to the range of the distance from x to y. This will be done in a way such that whenever the algorithm cannot find such a path from x to y, then the range has increased. By charging this particular increase in the distance from x to y, it can afford updating the sequence of centers. As in the case of reachability such paths from x to y will either be found via a hub or in a path union graph.

The second new aspect is that now we maintain approximate shortest paths up to h_i *hops* for certain pairs of i-centers, whereas in the reachability algorithm h_i was an unweighted distance. This motivates the following modification of the path union that limits allowed paths to a fixed number of hops.

Definition 4.4.1. *For all nodes x and y of a graph G and all integers $h \geq 1$ and $D \geq 1$, the h-*hop path union *$\mathscr{P}^h(x, y, D, G)$ is the set containing every node that lies on some path π from x to y in G that has $|\pi| \leq h$ edges (hops) and weight $w(\pi, G) \leq D$.*

Observe that $\mathscr{P}(x, y, D, G) = \mathscr{P}^n(x, y, R, G)$ by Definition 4.2.6. Dealing with exact bounded-hop paths is usually computationally more expensive than dealing with unbounded paths, even in the static setting.[8] The bounded hop path unions will henceforth only be used in the analysis of the algorithm. In the algorithm itself we will compute unbounded (i.e., n-hop) path unions in graphs with modified edge weights. For every $h \geq 1$ and every $r \geq 0$ we define a graph $\tilde{G}^{h,r}$ that has the same nodes and edges as G and in which we round the weight of every edge (u, v) to the

[8]For example, one can compute the shortest paths among all paths with at most h edges from a source node in time $O(mh)$ by running the first h iterations of the Bellman-Ford algorithm. This is not efficient enough for our purposes, as we would only like to spend nearly linear time.

next multiple of $\epsilon(1+\epsilon)^r/h$ by setting

$$w_{\tilde{G}^{h,r}}(u,v) = \left\lceil \frac{w_G(u,v)\cdot h}{\epsilon(1+\epsilon)^r} \right\rceil \cdot \frac{\epsilon(1+\epsilon)^r}{h} .$$

Note that the definition of $\tilde{G}^{h,r}$ implicitly depends on ϵ as well. To gain some intuition for these weight modifications, observe that for every path π in G consisting of h' edges we have

$$w_{\tilde{G}^{h,r}}(\pi) \leq w_G(\pi) + h' \cdot \frac{\epsilon(1+\epsilon)^r}{h} .$$

Thus, for every path π of weight $w_G(\pi) \geq (1+\epsilon)^r$ consisting of at most h edges we have

$$w_{\tilde{G}^{h,r}}(\pi) \leq (1+\epsilon)^r + \epsilon(1+\epsilon)^r = (1+\epsilon)^{r+1} .$$

As additionally $w_G(u,v) \leq w_{\tilde{G}^{h,r}}(u,v)$ for every edge (u,v) we obtain the following guarantees.

Lemma 4.4.2. *Let $h \geq 1$ and $r \geq 0$. For all pairs of nodes x and y we have $d_G(x,y) \leq d_{\tilde{G}^{h,r}}(x,y)$ and if $d^h_G(x,y) \geq (1+\epsilon)^r$, then $d_{\tilde{G}^{h,r}}(x,y) \leq (1+\epsilon)^{r+1}$.*

Now observe that h-hop path unions can be computed "approximately" by computing the (unbounded) path union in $\tilde{G}^{h,r}$, which can be done in nearly linear time (Lemma 4.2.7). In our case the unbounded path union in $\tilde{G}^{h,r}$ will contain the h-hop path union and we will later argue that all unbounded path unions computed by our algorithm have small size. In our analysis the following lemma will have the same purpose as Lemma 4.2.9 for the s-t reachability algorithm.

Lemma 4.4.3. *Let (x,y) be a pair of nodes, let $h \geq 1$, $r \geq 0$, and $\alpha \geq 1$, and let $Q \subseteq V$ be a set of nodes such that $\mathscr{P}^h(x,y,\alpha(1+\epsilon)^r, G) \subseteq Q$. Then $\mathscr{P}^h(x,y,\alpha(1+\epsilon)^r, G) \subseteq \mathscr{P}(x,y,\alpha(1+\epsilon)^{r+1}, \tilde{G}^{h,r}|Q)$.*

Proof. Let $v \in \mathscr{P}^h(x,y,\alpha(1+\epsilon)^r, G)$, i.e., there is a path π from x to y in G with at most h edges and weight at most $\alpha(1+\epsilon)^r$ containing v. The weight of π in $\tilde{G}^{h,r}$ is

$$w_{\tilde{G}^{h,r}}(\pi) \leq w_G(u,v) + h \cdot \frac{\epsilon(1+\epsilon)^r}{h} \leq \alpha(1+\epsilon)^r + \epsilon(1+\epsilon)^r \leq \alpha(1+\epsilon)^{r+1} .$$

By our assumption all nodes of π are contained in Q and thus π is a path in $\tilde{G}^{h,r}|Q$ of weight at most $\alpha(1+\epsilon)^{r+1}$. Thus, all nodes of π, including v, are contained in $\mathscr{P}(x,y,\alpha(1+\epsilon)^{r+1}, \tilde{G}^{h,r}|Q)$. □

The advantage of the graph $\tilde{G}^{h,r}$ is that its edge weights are multiples of $\epsilon(1+\epsilon)^r/h$. Thus, after *scaling down* the edge weights by a factor of $h/(\epsilon(1+\epsilon)^r)$ we still have integer weights. This observation can be used to speed up the pseudopolynomial algorithm of Even and Shiloach at the cost of a $(1+\epsilon)$-approximation [22, 23, 95].

4.4.2 Algorithm Description

Our approximate s-t shortest path algorithm has a parameter $k \geq 1$ and for each $1 \leq i \leq k$ parameters $b_i \leq n$ and $c_i \leq n$. We also set $b_{k+1} = 1$, $c_0 = n$, $c_{k+1} = 1$, and $h_i = n/c_i$ for all $0 \leq i \leq k+1$. Our algorithm determines sets of nodes $B_1 \supseteq B_2 \supseteq \ldots \supseteq B_k$ and $C_0 \supseteq C_1 \supseteq \ldots \supseteq C_{k+1}$ by random sampling as described in Section 4.3.1 for the s-t reachability algorithm. For each $1 \leq i \leq k$ we order the set of i-hubs in an arbitrary way such that $B_i = \{b_1, b_2, \ldots, b_{|B_i|}\}$. Our algorithm uses the following variables and data structures:

- An estimate $\delta(s,t)$ of the distance from s to t in G
- For every pair of i-centers (x,y) (with $1 \leq i \leq k$) an index $r(x,y,i)$ such that $0 \leq r(x,y,i) \leq \log_{1+\epsilon}(nW)$ for the current range of the distance from x to y.
- For every i-hub z (with $1 \leq i \leq k$) and every $0 \leq r \leq \log_{1+\epsilon}(nW)$ an ES-tree up to depth $(1+\epsilon)^{r+i+1}$ in $\tilde{G}^{8h_i/\epsilon,r}$ (*outgoing* tree of z) and an ES-tree up to depth $(1+\epsilon)^{r+i+1}$ in the reverse graph of $\tilde{G}^{8h_i/\epsilon,r}$ (*incoming* tree of z)
- For every pair of i-centers (x,y) (with $1 \leq i \leq k$) an index $l(x,y,i)$ such that $1 \leq l(x,y,i) \leq |B_i|+1$ that either gives the index $l(x,y,i)$ of the i-hub $b_{l(x,y,i)} \in B_i$ that links x to y for the current value of $r(x,y,i)$ or, if no such i-hub exists, is set to $l(x,y,i) = |B_i|+1$.
- For every pair of i-centers (x,y) (with $1 \leq i \leq k$) a set of nodes $Q(x,y,i)$, which is initially empty.
- For every pair of i-centers (x,y) (with $0 \leq i \leq k$) a list of pairs of $(i+1)$-centers called $(i+1)$-parents of (x,y).
- For every pair of i-centers (x,y) (with $1 \leq i \leq k+1$) a list of pairs of $(i-1)$-centers called $(i-1)$-children of (x,y).
- For every $1 \leq i \leq k$, a list A_i of i-centers called *active i-centers*.

Besides the generalizations for weighted graphs introduced in Section 4.4.1, our approximate s-t shortest path algorithm also deviates from the s-t reachability algorithm in the way it maintains the hub links. The main difference to before is that now we maintain the hub links only between pairs of active centers. In the new algorithm we say, for $1 \leq i \leq k$, that an i-hub $z \in B_i$ *links* an i-center x to an i-center y if

$$d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(x,z) + d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(z,y) \leq (1+\epsilon)^{r(x,y,i)+2i+1}.$$

The algorithm can check whether z links x to y by looking up $d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(x,z)$ and $d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(z,y)$ in the incoming and outgoing ES-tree of z, respectively. For every $1 \leq i \leq k$, and every pair of active i-centers x and y, the algorithm uses the index $l(x,y,i)$ to maintain an i-hub $b_{l(x,y,i)}$ that links x to y. If the current i-hub $b_{l(x,y,i)}$ does not link x to y anymore, the algorithm increases $l(x,y,i)$ until either such an

i-hub is found, or $l(x, y, i) = |B_i| + 1$. As soon as $l(x, y, i) = |B_i| + 1$, the algorithm computes the path union between x and y and maintains an approximate shortest path from x to y in the subgraph induced by the path union. If the algorithm does not find such a path anymore it increases the value of $r(x, y, i)$ and sets $l(x, y, i)$ to 1 again, i.e., making the first i-hub the candidate for linking x to y for the new value of $r(x, y, i)$. The generalization of the s-t reachability algorithm is now straightforward and the pseudocode can be found in Algorithm 4.2.

4.4.3 Correctness

To establish the correctness of the algorithm we will show that

$$d_G(s, t) \leq \delta(s, t) \leq (1 + \epsilon)^{2k+1} d_G(s, t) .$$

By running the whole algorithm with $\epsilon' = \epsilon/(4k + 2)$ (instead of ϵ), we then obtain a $(1 + \epsilon)$-approximation (see Lemma 4.4.7 below).

We will achieve this by showing that with the value of $r(x, y, i)$ of a pair of i-centers (x, y) the algorithm keeps track of the range of the distance from x to y. In particular, we will show that $d_G^{h_i}(x, y) \geq (1 + \epsilon)^{r(x,y,i)}$ for every pair of i-centers (x, y) and $d_G(x, y) \leq (1 + \epsilon)^{r(x,y,i)+2i+1}$ for every pair of active i-centers $(x, y) \in A_i$. Both inequalities are obtained by a "bottom up" analysis of the algorithm. In order to prove the lower bound on $d_G^{h_i}(x, y)$ we have to show that the subgraph $Q(x, y, i)$ computed by the algorithm indeed contains the corresponding path union. In fact those two invariants rely on each other and we will prove them mutually.

Lemma 4.4.4. *Algorithm 4.2 correctly maintains the following invariants whp:*

(I1) For every $1 \leq i \leq k$ and every pair of i-centers (x, y),

$$d_G^{h_i}(x, y) \geq (1 + \epsilon)^{r(x,y,i)} .$$

(I2) For every $1 \leq i \leq k$ and every pair of active i-centers $(x, y) \in A_i$,

$$\mathscr{P}^{8h_i/\epsilon}(x, y, (1 + \epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i) .$$

Proof. Both invariants trivially hold directly before the first refresh in Line 36 of the initialization is called because we set $r(x, y, i) = 0$ for all $1 \leq i \leq k$ and no pairs of centers are active yet.

We first give a proof of Invariant (I1). As distances are non-decreasing in G we only have to argue that the invariant still holds after each time the algorithm changes the value of $r(x, y, i)$. The only place where $r(x, y, i)$ is changed is in Line 8 (where it is increased by 1) and if this line is reached the condition

$$d_{\tilde{G}^{h_i, r(x,y,i)}|Q(x,y,i)}(x, y) > (1 + \epsilon)^{r(x,y,i)+2}$$

holds.

Algorithm 4.2: Algorithm for decremental approximate s-t shortest path

Procedure Refresh(x, y, i)
  RemoveChildren(x, y, i)
  if $i = k + 1$ **then**
    Compute shortest path π from s to t in G and set $\delta(s,t) \leftarrow (1+\epsilon)^{2k+1} d_G(s,t)$
  else
    Compute shortest path π from x to y in $\tilde{G}^{h_i, r(x,y,i)} | Q(x,y,i)(x,y)$
    if $d_{\tilde{G}^{h_i, r(x,y,i)} | Q(x,y,i)}(x,y) > (1+\epsilon)^{r(x,y,i)+2}$ **then** // Range has increased
      $r(x,y,i) \leftarrow r(x,y,i) + 1$
      $l(x,y,i) \leftarrow 1$
      foreach $(i+1)$-*parent* (x',y') *of* (x,y) **do** Refresh(x', y', $i+1$)
      break
  Determine $(i-1)$-centers $v_1, \ldots, v_l$ in order of appearance on π
  foreach $1 \le j \le l-1$ **do**
    Add (v_j, v_{j+1}) to A_{i-1}
    Make (v_j, v_{j+1}) an $(i-1)$-child of (x,y) and (x,y) an i-parent of (v_j, v_{j+1})
    if $i \ge 1$ **then**
      UpdateHubLinks(v_j, v_{j+1}, $i-1$, $r(v_j, v_{j+1}, i-1)$)
      if $l_i(v_j, v_{j+1}) = |B_i| + 1$ **then**
        $Q(v_j, v_{j+1}, i-1) \leftarrow$
        $\mathcal{P}(v_j, v_{j+1}, (1+\epsilon)^{r(v_j, v_{j+1}, i-1)+2i-1}, \tilde{G}^{8h_{i-1}/\epsilon, r(v_j, v_{j+1}, i-1)} | Q(x,y,i))$
        Refresh(v_j, v_{j+1}, $i-1$)

Procedure UpdateHubLinks(x, y, i, r)
  while $l(x,y,i) \le |B_i|$ **and**
  $d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(x, b_{l(x,y,i)}) + d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(b_{l(x,y,i)}, y) > (1+\epsilon)^{r(x,y,i)+2i+1}$ **do**
    $l(x,y,i) \leftarrow l(x,y,i) + 1$
    if $l(x,y,i) = |B_i| + 1$ **then**
      Let (x',y') be any $(i+1)$-parent of (x,y)
      $Q(x,y,i) \leftarrow \mathcal{P}(x, y, (1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)} | Q(x',y',i+1))$

Procedure RemoveChildren(x, y, i)
  foreach $(i-1)$-*child* (x',y') *of* (x,y) **do**
    Remove (x',y') from $(i-1)$-children of (x,y) and (x,y) from i-parents of (x',y')
    if (x',y') *has no i-parents anymore* **then**
      Remove (x',y') from A_{i-1}
      if $i \ge 1$ **then** RemoveChildren(x', y', $i-1$)

Procedure Initialize()
  foreach $1 \le i \le k$ *and all* i-*centers* (x,y) **do** $r(x,y,i) \leftarrow 0$, $l(x,y,i) \leftarrow 1$, $Q(x,y,i) \leftarrow \emptyset$
  foreach $1 \le i \le k$ **do** $A_i \leftarrow \emptyset$
  Refresh(s, t, $k+1$)

Procedure Delete(u, v)
  foreach 1-*parent* (x,y) *of* (u,v) **do** Refresh(x, y, 1)
  foreach $1 \le i \le k$ *and* $(x,y) \in A_i$ **do**
    UpdateHubLinks(x, y, i, $r(x,y,i)$)
    if $l(x,y,i) = |B_i| + 1$ **then** Refresh(x, y, i)

Suppose that in G there is a path π from x to y with at most h_i edges and weight less than $(1+\epsilon)^{r(x,y,i)+1}$. The weight of π in $\tilde{G}^{h_i,r(x,y,i)}$ is

$$\begin{aligned} w_{\tilde{G}^{h_i,r(x,y,i)}}(\pi) &\leq w_G(\pi) + h_i \cdot \frac{\epsilon(1+\epsilon)^{r(x,y,i)}}{h_i} \\ &\leq (1+\epsilon)^{r(x,y,i)+1} + \epsilon(1+\epsilon)^{r(x,y,i)} \\ &\leq (1+\epsilon)^{r(x,y,i)+2} \\ &\leq (1+\epsilon)^{r(x,y,i)+2i}. \end{aligned}$$

By Invariant (I2) all nodes of π are contained in $Q(x,y,i)$. Therefore

$$d_{\tilde{G}^{h_i,r(x,y,i)}|Q(x,y,i)}(x,y) \leq w_{\tilde{G}^{h_i,r(x,y,i)}}(\pi) \leq (1+\epsilon)^{r(x,y,i)+2}$$

which contradicts the assumption that the algorithm has reached Line 8. Thus, the path π does not exist and it follows that $d_G^{h_i}(x,y) \geq (1+\epsilon)^{r(x,y,i)+1}$ as desired. Therefore Invariant (I1) still holds when the algorithm increases the value of $r(x,y,i)$ by 1.

We now give a proof of Invariant (I2). The proof is by induction on i. There are two places in the algorithm where the value of $Q(x,y,i)$ is changed: Line 19 and Line 26. Note that $r(x,y,i)$ is non-decreasing and $l(x,y,i)$ decreases only if $r(x,y,i)$ increases. Therefore, for a fixed value of $r(x,y,i)$, the algorithm will "recompute" $Q(x,y,i)$ in Line 19 only if it has "initialized" it in Line 26 before.

Consider first the case that $Q(x,y,i)$ is "initialized" (Line 26) and let (x',y') be the $(i+1)$-parent of (x,y) used in this computation. If $i \leq k-1$, then by the induction hypothesis we know that

$$\mathscr{P}^{8h_{i+1}/\epsilon}(x',y',(1+\epsilon)^{r(x',y',i+1)+2(i+1)},G) \subseteq Q(x',y',i+1). \tag{4.7}$$

If $i = k$, then this inclusion also holds because in that case $x' = s$ and $y' = t$ (as s and t are the only $(k+1)$-centers) and we have set $Q(s,t,k+1) = V$. We will show below that, at the last time the algorithm has called Refresh$(x', y', i+1)$ (where it sets (x,y) as an i-child of (x',y') and (x',y') as an $(i+1)$-parent of (x,y)), we have

$$\mathscr{P}^{8h_i/\epsilon}(x,y,(1+\epsilon)^{r(x,y,i)+2i},G) \subseteq \mathscr{P}^{8h_{i+1}/\epsilon}(x',y',(1+\epsilon)^{r(x',y',i+1)+2(i+1)},G)$$

and thus (together with (4.7))

$$\mathscr{P}^{8h_i/\epsilon}(x,y,(1+\epsilon)^{r(x,y,i)+2i},G) \subseteq Q(x',y',i+1). \tag{4.8}$$

As distances in G are non-decreasing, (4.8) will still hold at the time the algorithm sets

$$Q(x,y,i) = \mathscr{P}(x,y,(1+\epsilon)^{r(x,y,i)+2i+1},\tilde{G}^{8h_i/\epsilon,r(x,y,i)}|Q(x',y',i+1)). \tag{4.9}$$

according to Line 26. By Lemma 4.4.3, (4.8) will imply that

$$\begin{aligned} &\mathscr{P}^{8h_i/\epsilon}(x,y,(1+\epsilon)^{r(x,y,i)+2i},G) \\ &\qquad \subseteq \mathscr{P}(x,y,(1+\epsilon)^{r(x,y,i)+2i+1},\tilde{G}^{8h_i/\epsilon,r(x,y,i)}|Q(x',y',i+1)) = Q(x,y,i) \end{aligned}$$

as desired. Note again that, since distances in G are nondecreasing, the inclusion $\mathscr{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i)$ will then continue to hold until $Q(x, y, i)$ is "recomputed" (Line 19) for the first time. Whenever this happens, it will set new the value of $Q(x, y, i)$ equal to $\mathscr{P}(x, y, (1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)}|Q(x, y, i))$. Thus, by Lemma 4.4.3 we will again have $\mathscr{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i)$ as demanded by Invariant (I2).

It remains to show that at the last time the algorithm has called Refresh(x', y', $i+1$) and sets (x', y') as an $(i+1)$-parent of (x, y), we have

$$\mathscr{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq \mathscr{P}^{8h_{i+1}/\epsilon}(x', y', (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G)\,.$$

Let π' denote the shortest path from x' to y' in $\tilde{G}^{h_{i+1}, r(x',y',i+1)}|Q(x', y', i+1)$ computed at the beginning of the last execution of Refresh(x', y', $i+1$). To enhance the readability of this part of the proof we use the abbreviation $H = \tilde{G}^{h_{i+1}, r(x',y',i+1)}|Q(x', y', i+1)$. By the if-condition in Line 7 we know that $w(\pi', H) = d_H(x', y') \leq (1+\epsilon)^{r(x',y',i+1)+2}$.

Consider some node $v \in \mathscr{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G)$ which means that v lies on a path π from x to y in G that has at most $8h_i/\epsilon$ edges and weight at most $(1+\epsilon)^{r(x,y,i)+2i}$. Remember that x and y are consecutive i-centers on π'. Let π'_1 and π'_2 denote the subpaths of π' from x' to x and y to y', respectively. The concatenation $\pi'' = \pi'_1 \circ \pi \circ \pi'_2$ is a path from x to y in G. We will show that π'' has at most $8h_{i+1}/\epsilon$ edges and weight at most $(1+\epsilon)^{r(x',y',i+1)+2(i+1)}$. This then proves that all nodes on π are contained in $\mathscr{P}^{8h_{i+1}/\epsilon}(x, y, (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G)$.

As $2h_i \leq h_{i+1}$, the number of edges of π is at most $8h_i/\epsilon \leq 4h_{i+1}/\epsilon$. Remember that the edge weights of H are multiples of $\epsilon(1+\epsilon)^{r(x',y',i+1)}/h_{i+1}$. Thus, each edge of H has weight at least $\epsilon(1+\epsilon)^{r(x',y',i+1)}/h_{i+1}$. As the weight of π' in H is at most $(1+\epsilon)^{r(x',y',i+1)+2}$, the number of edges of π' is at most

$$\frac{(1+\epsilon)^{r(x',y',i+1)+2} h_{i+1}}{\epsilon(1+\epsilon)^{r(x',y',i+1)}} = \frac{(1+\epsilon)^2 h_{i+1}}{\epsilon} \leq \frac{4}{\epsilon}\,.$$

It follows that the number of edges of π'' is at most $4h_{i+1}/\epsilon + 4h_{i+1}/\epsilon = 8h_{i+1}/\epsilon$.

By Invariant (I1) we have $d_G^{h_i}(x, y) \geq (1+\epsilon)^{r(x,y,i)}$ and thus

$$w_G(\pi) \leq (1+\epsilon)^{r(x,y,i)+2i} \leq (1+\epsilon)^{2i} d_G^{h_i}(x, y)\,.$$

By the initial random sampling of i-centers, every shortest path consisting of $h_i - 1$ edges contains an i-center whp (Lemma 1.3.2). Thus, the subpath of π' from x to y has at most h_i edges. Since π' is a shortest path in H we have $d_H(x, y) = d_H^{h_i}(x, y)$. It follows that

$$\begin{aligned} w_G(\pi) \leq (1+\epsilon)^{2i} d_G^{h_i}(x, y) \leq (1+\epsilon)^{2i} d^{h_i}_{G|Q(x',y',i+1)}(x, y) &\leq (1+\epsilon)^{2i} d_H^{h_i}(x, y) \\ &= (1+\epsilon)^{2i} d_H(x, y)\,. \end{aligned}$$

Furthermore we have $d_H(x', y') = d_H(x', x) + d_H(x, y) + d_H$ (and in particular $d_H(x, y) \leq d_H(x', y')$). As $w_G(\pi_1') \leq d_H(x', x)$ and $w_G(\pi_2') \leq d_H(y, y')$ the weight of the path π'' in G can be bounded as follows:

$$\begin{aligned}
w_G(\pi'') &= w_G(\pi_1') + w_G(\pi_2') + w_G(\pi) \\
&\leq d_H(x', x) + d_H(y, y') + w_G(\pi) \\
&\leq d_H(x', y') - d_H(x, y) + w_G(\pi) \\
&\leq d_H(x', y') - d_H(x, y) + (1+\epsilon)^{2i} d_H(x, y) \\
&= d_H(x', y') + ((1+\epsilon)^{2i} - 1) d_H(x, y) \\
&\leq d_H(x', y') + ((1+\epsilon)^{2i} - 1) d_H(x', y') \\
&= (1+\epsilon)^{2i} d_H(x', y') \\
&\leq (1+\epsilon)^{2i} (1+\epsilon)^{r(x', y', i+1)+2} \\
&= (1+\epsilon)^{r(x', y', i+1)+2(i+1)}
\end{aligned}$$

□

Lemma 4.4.5. *For every $1 \leq i \leq k$ and every active pair of i-centers $(x, y) \in A_i$ we have $d_G(x, y) \leq (1+\epsilon)^{r(x,y,i)+2i+1}$ whp after the algorithm has finished its updates.*

Proof. The proof is by induction on i. If $l(x, y, i) \leq |B_i|$, then there is some i-hub z that links x to y. By the triangle inequality we then have

$$\begin{aligned}
d_G(x, y) &\leq d_G(x, z) + d_G(z, y) \\
&\leq d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(x, z) + d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(z, y) \\
&\leq (1+\epsilon)^{r(x,y,i)+2i+1}.
\end{aligned}$$

If $l(x, y, i) = |B_i| + 1$, then consider the last time the algorithm has called Refresh(x, y, i). Let G' and $Q'(x, y, i)$ denote the versions of G and $Q(x, y, i)$ at the beginning of the refresh operation and let π be the shortest path from x to y in $(\tilde{G}')^{h_i, r(x,y,i)}|Q'(x, y, i)$ computed by the algorithm. To enhance readability we set $H = (\tilde{G}')^{h_i, r(x,y,i)}|Q'(x, y, i)$ in this proof. The algorithm ensures that $d_H(x, y) \leq (1+\epsilon)^{r(x,y,i)+2}$ as otherwise, by the if-condition in Line 7, (x, y) would have either become inactive (i.e., removed from A_i) or the algorithm would have called Refresh(x, y, i) again. Let $v_1, \ldots, v_l$ denote the $(i-1)$-centers in order of appearance on π, i.e., $d_H(x, y) = \sum_{1 \leq j \leq l-1} d_H(v_j, v_{j+1})$.

If $i = 1$, then remember that the set of 0-centers is equal to V. Note that no edge (u, v) of π has been deleted from G since the last time the algorithm has called Refresh(x, y, i) because (x, y) is a 1-parent of (u, v). Therefore all edges of π still exist in G. Since $r(x, y, i)$ is non-decreasing this means that $d_G(x, y) \leq d_H(x, y) \leq (1+\epsilon)^{r(x,y,i)+2}$ as desired.

If $i \geq 2$, then let $1 \leq j \leq l-1$. First of all, note that the value of $r(v_j, v_{j+1}, i)$ has not changed since the last time the algorithm has called Refresh(x, y, i) because otherwise after Line 8, the algorithm would call Refresh(x, y, i) as (x, y) is an i-parent of (v_j, v_{j+1}). Therefore, by Invariant (I1) of Lemma 4.4.4, we have $d_{G'}^{h_{i-1}}(v_j, v_{j+1}) \geq (1+\epsilon)^{r(v_j, v_{j+1}, i-1)}$. Note that H has the same nodes and edges as G' and the weight of

each edge in H is at least its weight in G'. Therefore $d_H^{h_{i-1}}(v_j, v_{j+1}) \geq d_{G'}^{h_{i-1}}(v_j, v_{j+1})$ and thus $d_H^{h_{i-1}}(v_j, v_{j+1}) \geq (1+\epsilon)^{r(v_j, v_{j+1}, i-1)}$. By the initial random sampling of $(i-1)$-centers, every shortest path consisting of $h_{i-1} - 1$ edges contains an $(i-1)$-center whp (Lemma 1.3.2). Thus, the subpath of π from v_j to v_{j+1} has at most h_{i-1} edges which means that $d_H(v_j, v_{j+1}) = d_H^{h_{i-1}}(v_j, v_{j+1}) \geq (1+\epsilon)^{r(v_j, v_{j+1}, i-1)}$.

By the induction hypothesis we have $d_G(v_j, v_{j+1}) \leq (1+\epsilon)^{r(v_j, v_{j+1}, i-1)+2(i-1)+1}$ in the current version of G, which by the argument above implies that $d_G(v_j, v_{j+1}) \leq (1+\epsilon)^{2(i-1)+1} d_H(v_j, v_{j+1})$. Thus, by the triangle inequality we get

$$\begin{aligned} d_G(x, y) &\leq \sum_{1 \leq j \leq l-1} d_G(v_j, v_{j+1}) \\ &\leq \sum_{1 \leq j \leq l-1} (1+\epsilon)^{2(i-1)+1} d_H(v_j, v_{j+1}) \\ &= (1+\epsilon)^{2i-1} d_H(x, y) \\ &\leq (1+\epsilon)^{2i-1} (1+\epsilon)^{r(x,y,i)+2} \\ &= (1+\epsilon)^{2i+1} \end{aligned}$$

as desired. □

Lemma 4.4.6. *After the algorithm has finished its updates we have $d_G(s,t) \leq \delta(s,t) \leq (1+\epsilon)^{k+1} d_G(s,t)$ whp.*

Proof. Consider the last time the algorithm has called Refresh$(s, t, k+1)$. Let G' denote the versions of G at the beginning of the refresh operation and let π be the shortest path from x to y in G computed by the algorithm. Note that $\delta(s,t) = (1+\epsilon)^{2k+1} d_{G'}(s,t)$. As distances are non-decreasing under deletions in G we trivially have $\delta(s,t) \leq (1+\epsilon)^{k+1} d_G(s,t)$.

Let $v_1, \ldots, v_l$ denote the k-centers in order of appearance on π, i.e., $d_{G'}(x,y) = \sum_{1 \leq j \leq l-1} d_{G'}(v_j, v_{j+1})$, and let $1 \leq j \leq l-1$. The value of $r(v_j, v_{j+1}, k)$ has not changed since the last time the algorithm has called Refresh(x, y, i) because otherwise after Line 8, the algorithm would call Refresh(s, t, k) as (s,t) is a $(k+1)$-parent of (v_j, v_{j+1}). By Lemma 4.4.4, we have $d_{G'}^{h_k}(v_j, v_{j+1}) \geq (1+\epsilon)^{r(v_j, v_{j+1}, k)}$ and by Lemma 4.4.5 we have $d_G(v_j, v_{j+1}) \leq (1+\epsilon)^{r(v_j, v_{j+1}, k)+2k+1}$. It follows that $d_G(v_j, v_{j+1}) \leq (1+\epsilon)^{2k+1} d_{G'}(v_j, v_{j+1})$.

By the initial random sampling of k-centers, every shortest path consisting of $h_k - 1$ edges contains a k-center whp (Lemma 1.3.2). Thus, the subpath of π from v_j to v_{j+1} has at most h_k edges which means that $d_{G'}(v_j, v_{j+1}) = d_{G'}^{h_k}(v_j, v_{j+1})$. Thus, by the triangle inequality we get

$$\begin{aligned} d_G(s, t) &\leq \sum_{1 \leq j \leq l-1} d_G(v_j, v_{j+1}) \\ &\leq \sum_{1 \leq j \leq l-1} (1+\epsilon)^{2k+1} d_{G'}(v_j, v_{j+1}) \\ &= (1+\epsilon)^{2k+1} d_{G'}(s, t) \\ &= \delta(s, t) \end{aligned}$$

□

Lemma 4.4.7. *For all* $0 \le x \le 1$ *and all* $y > 0$,

$$\left(1 + \frac{x}{2y}\right)^y \le 1 + x\,.$$

Proof. Let e denote Euler's constant. We will use the following well-known inequalities: $(1 + 1/z)^z \le e$ (for all $z > 0$), $e^z \le 1/(1 - z)$ (for all $z < 1$), and $1/(1 - z) \le 1 + 2z$ (for all $0 \le z \le 1/2$). We then get:

$$\left(1 + \frac{x}{2y}\right)^y = \left(\left(1 + \frac{x}{2y}\right)^{\frac{2y}{x}}\right)^{\frac{x}{2}} \le e^{\frac{x}{2}} \le \frac{1}{1 - \frac{x}{2}} \le 1 + x\,. \qquad \square$$

4.4.4 Running Time

The running time analysis follows similar arguments as for the s-t reachability algorithm in Section 4.3.3. For every pair of i-centers (x, y), the algorithm never decreases $r(x, y, i)$ and (x, y) causes a refresh only if $r(x, y, i)$ increases, which happens $O(\log W/\epsilon)$ times. Furthermore it can be argued in a straightforward way that for every $1 \le i \le k+1$ and every pair of i-centers (x, y) the graph $G|Q(x, y, i)$ has at most $\min(m/b_i, n^2/b_i^2)$ edges whp. Now the only differences compared to the running time analysis of the s-t reachability algorithm are the number of path unions of active pairs of centers each node is contained in and the time needed for maintaining the hub links, both of which are analyzed below.

Lemma 4.4.8. *Let* $1 \le i \le k$ *and consider a pair of active* $(i + 1)$*-centers* (x', y') *and their* i*-children* $(x_j, y_j)_{1 \le j \le l}$ *(which are active* i*-centers). Then, for every node* v*, there are at most* $q = 2^{2i+2}\lceil \log_{1+\epsilon}(nW)\rceil$ *pairs of* i*-children* (x_j, y_j) *of* (x', y') *such that* $v \in Q(x_j, y_j, i)$.

Proof. We show that at the last time the algorithm has called Refresh$(x', y', i + 1)$ (where it determined the current i-children of (x', y')) there are at most $q = 2^{2i+2}\lceil \log_{1+\epsilon}(nW)\rceil$ pairs of i-children (x_j, y_j) of (x', y') such that

$$v \in \mathscr{P}(x_j, y_j, (1 + \epsilon)^{r(x_j,y_j,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x_j,y_j,i)}|Q(x', y', i + 1))\,.$$

Note that, for each $1 \le i \le l$, $r(x_j, y_j, i)$ has not changed since the last refresh (as such a change implies refreshing (x', y')). Thus, for each $1 \le i \le l$, $Q(x_j, y_j, i)$ is a subset of the path union above, which means that the lemma will be implied by this claim. Now we actually prove the following slightly stronger claim: for every every node v and every $0 \le r \le \lfloor \log_{1+\epsilon}(nW)\rfloor$, there are at most $q' = 2^{2i+2}$ pairs of i-children (x_j, y_j) of (x', y') such that

$$v \in \mathscr{P}(x_j, y_j, (1 + \epsilon)^{r+2i+1}, \tilde{G}^{8h_i/\epsilon, r}|Q(x', y', i + 1)) \text{ and } d_G^{h_i}(x_j, y_j) \ge (1 + \epsilon)^r\,.$$

In this proof we use the abbreviation $H = \tilde{G}^{8h_i/\epsilon, r}|Q(x', y', i + 1)$. Suppose that v is contained in $q' > 2^{2i+2}$ path unions $\mathscr{P}(x_j, y_j, (1 + \epsilon)^{r+2i+1}, H)$ of i-children (x_j, y_j)

such that $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$. Let $j_1, \ldots, j_{q'}$ be the corresponding indices and assume without loss of generality that $j_1 < j_2 \ldots < j_{q'}$. By the initial random sampling of i-centers, every shortest path consisting of $h_i - 1$ edges contains an i-center whp (Lemma 1.3.2). Therefore, for every $1 \leq j \leq l$, there is a shortest path between x_j and y_j in H with at most h_i edges, i.e., $d_H(x_j, y_j) = d_H^{h_i}(x_j, y_j)$. Furthermore, for every $j \in \{j_1, \ldots, j_{q'}\}$, we have $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$ by our assumption and thus

$$\begin{aligned} d_H(x_{j_1}, y_{j_{q'}}) &\geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_H(x_j, y_j) = \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_H^{h_i}(x_j, y_j) \\ &\geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_G^{h_i}(x_j, y_j) \geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} (1+\epsilon)^r = q'(1+\epsilon)^r \end{aligned} \tag{4.10}$$

We now derive an upper bound on $d_H(x_{j_1}, y_{j_{q'}})$ contradicting this lower bound. Since v is contained in the path union $\mathscr{P}(x_j, y_j, (1+\epsilon)^{r+2i+1}, H)$, we know by the definition of the path union that v lies on a shortest path from x_{j_1} to y_{j_1} in H of weight at most $(1+\epsilon)^{r+2i+1}$ and thus $d_H(x_{j_1}, v) \leq (1+\epsilon)^{r+2i+1}$. The same argument shows that $d_H(v, y_{j_{q'}}) \leq (1+\epsilon)^{r+2i+1}$. By the triangle inequality we therefore have

$$d_H(x_{j_1}, y_{j_{q'}}) \leq d_H(x_{j_1}, v) + d_H(v, y_{j_{q'}}) \leq 2(1+\epsilon)^{r+2i+1} \tag{4.11}$$

By combining Inequalities (4.10) and (4.11) we get $q' \leq 2(1+\epsilon)^{2i+1}$, which is a contradictory statement for $q' > 2^{2i+2}$ because $\epsilon \leq 1$. Thus, v is contained in at most $q' = 2^{2i+2}$ path unions $\mathscr{P}(x_j, y_j, (1+\epsilon)^r, H)$ of i-children (x_j, y_j) such that $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$. □

We now bound the time needed for maintaining the hub links as follows. For every i-hub z (with $1 \leq i \leq k$) and every $0 \leq r \leq \log_{1+\epsilon}(nW)$ we maintain both an incoming and an outgoing ES-tree up to depth $(1+\epsilon)^{r+2i+1}$ in $\tilde{G}^{8h_i/\epsilon, r}$. In $\tilde{G}^{8h_i/\epsilon, r}$ every edge weight is a multiple of $\rho = \epsilon^2(1+\epsilon)^r/(8h_i)$ and thus we can scale down the edge weights by the factor $1/\rho$ and maintain the ES-tree in the resulting integer-weighted graph up to depth

$$\frac{(1+\epsilon)^{r+2i+1}}{\rho} \leq \frac{2^{2k+1}(1+\epsilon)^r}{\rho} = \frac{2^{2k+1}(1+\epsilon)^r 8h_i}{\epsilon^2(1+\epsilon)^r} = O(2^{2k}h_i/\epsilon^2)\,.$$

Thus, maintaining these two trees takes time $O(2^{2k}mh_i/\epsilon^2)$ which is $O(2^{2k}mn/(c_i\epsilon^2))$ as $h_i = n/c_i$. As we have $O(\log_{1+\epsilon}(nW)) = \tilde{O}(\log W/\epsilon)$ such trees for every i-hub and there are $\tilde{O}(b_i)$ i-hubs in expectation, maintaining all these trees takes time $\tilde{O}(\sum_{1 \leq i \leq k} 2^{2k} b_i mn \log W/(c_i\epsilon^3))$ in expectation.

We now bound the time needed for maintaining the index $l(x, y, i)$ for every pair of i-centers (x, y). First, observe that $l(x, y, i)$ assumes integer values from 1 to $|B_i|$ (the number of i-hubs) and it only decreases (to 0) if $r(x, y, i)$ increases. The index $r(x, y, i)$ on the other hand is non-decreasing and assumes integer values from 0 to $\lceil \log_{1+\epsilon}(nW) \rceil = \tilde{O}(\log W/\epsilon)$. As $|B_i| = \tilde{O}(b_i)$ in expectation, the value of $l(x, y, i)$

therefore changes $\tilde{O}(b_i \log W/\epsilon)$ times. As there are $\tilde{O}(2^k c_i)$ i-centers in expectation, the indices of all pairs of centers together change at most $\tilde{O}(\sum_{1\le i\le k} 2^{2k} b_i c_i^2 \log W/\epsilon)$ times. It remains to bound the time spent in total for calls of the form UpdateHubLinks(x, y, i, r) in which the index $l(x, y, i)$ does not increase but the algorithm still spends constant time for checking whether $l(x, y, i)$ should increase. In such a case we charge the running time to the pair of i-centers (x, y). Note that the call UpdateHubLinks(x, y, i, r) has either happened because (x, y) is made an i-child of some pair of $i + 1$-centers or because (x, y) is an active pair of i-centers (i.e., in A_i) after the deletion of a node. We only have to focus on the second case because in the first case the charge of $O(1)$ on (x, y) can be neglected. As argued above, at any time, there are at most $O(q^k \log_{1+\epsilon}(nW))$ active i-centers. Furthermore, there are at most m deletions in G. Therefore the total time needed for maintaining all indices $l(x, y, i)$ is $\tilde{O}(\sum_{1\le i\le k} 2^{2k} b_i c_i^2 \log W/\epsilon + q^k m \log W + \epsilon)$.

Putting everything together, the total running time of our algorithm using k layers is

$$\tilde{O}\left(\sum_{1\le i\le k} \frac{b_i c_i^2 \log W}{\epsilon} + \sum_{1\le i\le k} \frac{b_i mn \log W}{c_i \epsilon^3} + q^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2}\right) + \sum_{1\le i\le k-1} q^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2}\right) + 2^{2k} q^{2k} c_k^2 m\right)$$

where $q = 2^{2k+2}\lfloor \log_{1+\epsilon} W \rceil$. Assuming that $W \le 2^{\log^c n}$ and $\epsilon \ge 1/\log^c n$ and setting $k = \lfloor \log^{1/4} n \rfloor$ we get $2^k q = O(n^{O(\log\log n/\log^{1/4} n)}) = O(n^{o(1)})$. We now simply set the parameters $b_i \le n$ and $c_i \le n$ for each $1 \le i \le k$ in the same way as in Section 4.3.3 to obtain the same asymptotic running time (in terms of polynomial factors) as for the s-t reachability algorithm.

Theorem 4.4.9. *For every $W \ge 1$ and every $0 < \epsilon \le 1$ such that $W \le 2^{\log^c n}$ and $\epsilon \ge 1/\log^c n$ for some constant c, there is a decremental $(1 + \epsilon)$-approximate stSP algorithm with constant query time and expected update time*

$$O(\min(m^{5/4} n^{1/2+o(1)}, m^{2/3} n^{4/3+o(1)})) = O(mn^{6/7+o(1)})$$

that is correct with high probability against an oblivious adversary.

Similar to the s-t reachability algorithm we can extend the $(1 + \epsilon)$-approximate stSP algorithm in the following ways.

Corollary 4.4.10. *For every $W \ge 1$ and every $0 < \epsilon \le 1$ such that $W \le 2^{\log^c n}$ and $\epsilon \ge 1/\log^c n$ for some constant c, there is a decremental $(1 + \epsilon)$-approximate algorithm for maintaining shortest paths between p source-sink pairs with constant query time and expected update time*

$$O(\min(p^{1/2} m^{5/4} n^{1/2+o(1)}, p^{1/3} m^{2/3} n^{4/3+o(1)}))$$

that is correct with high probability against an oblivious adversary.

Corollary 4.4.11. *For every $W \geq 1$ and every $0 < \epsilon \leq 1$ such that $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c, there is a decremental $(1+\epsilon)$-approximate SSSP algorithm with constant query time and expected update time*

$$O(\min(m^{7/6}n^{2/3+o(1)}, m^{3/4}n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$$

that is correct with high probability against an oblivious adversary.

4.5 Faster Single-Source Reachability in Dense Graphs

In this section we first introduce a path union data structure that is more efficient than the naive approach for repeatedly computing path unions between one fixed node and other variable nodes. We then show how to combine it with a multilayer approach to obtain a faster decremental single-source reachability algorithm for dense graphs.

4.5.1 Approximate Path Union Data Structure

In the following we present a data structure for a graph G undergoing edge deletions, a fixed node x and a parameter h. It has a procedure ApproximatePathUnion(y). When this procedure is called for any node y, it computes an "approximation" of the path union $\mathscr{P}(x, y, h, G)$. Using a simple static algorithm a path union can be computed in time $O(m)$ for each pair (x, y). We give an (almost) output-sensitive data structure for this problem, i.e., using our data structure the time will be proportional to the size of $G|\mathscr{P}(x, y, h, G)$ which might be $o(m)$. Additionally, we have to pay a global cost of $O(m)$ that is amortized over *all* approximate path union computations for the node x and *all* nodes y. This will be useful because in our reachability algorithm we can use probabilistic arguments to bound the size of the path union we want to compute.

Algorithm Description

Internally, the data structure maintains a set $R(x)$ of nodes such that the following invariant is fulfilled at any time: all nodes that can be reached from x by a path of length at most h are contained in $R(x)$ (but $R(x)$ might contain other nodes as well). Observe that thus $R(x)$ contains the path union $\mathscr{P}(x, y, h, G)$ for every node y. Given some node y, the path union $\mathscr{P}(x, y, h, G)$ can be computed on the subgraph of G induced by $R(x)$ as follows. We first determine all nodes that are at distance at most h to y using breadth-first-search (BFS). Then, on the subgraph induced by these nodes, we run a second BFS to determine all nodes that are at distance at most h from x. The nodes visited by the second BFS are exactly the nodes in the path union $\mathscr{P}(x, y, h, G)$.

The running time of this algorithm would be proportional to the number of edges of the subgraph induced by the first BFS. However, we only want to charge time proportional to the number of edges of the subgraph induced by the second BFS,

which is usually smaller. In our new algorithm we solve this problem by charging the additional work to a set of nodes that will be removed from $R(x)$ and thus will never be visited anymore in any future path union computation. The algorithm ensures that this set only contains nodes that can safely be removed, i.e., nodes that will never be contained in any path union $\mathscr{P}(x, y, h, G)$ for some node y anymore. For this charging scheme to work the algorithm has to make sure that the number of nodes removed from $R(x)$ (more specifically its number of incident edges) is large enough. As long as this is not the case, the algorithm redoes the computation and each time allows an additional h in the depth of the first BFS. Note that the result of the algorithm might now contain more nodes than the actual path union. However, we show that the depth required in the first BFS is at most $O(h \log n)$ and thus the path union is approximated well enough. Procedure 4.3 shows the pseudocode of the algorithm.

Procedure 4.3: ApproximatePathUnion(y)

```
   // All calls of ApproximatePathUnion(y) use fixed x and h.
1  X_0 ← ∅ and m_0 ← 0
2  for i = 1 to ⌈log m⌉ + 1 do
3      Compute B_i = {v ∈ R(x) | d_{G|R(x)}(v, y) ≤ ih} // backward BFS to y in
           subgraph induced by R(x)
4      Compute F_i = {v ∈ B_i | d_{G|B_i}(x, v) ≤ h} // forward BFS from x in
           subgraph induced by B_i
5      X_i ← B_i \ F_i
6      m_i ← |E(B_i, B_i)|
7      if m_i ≤ 2m_{i−1} then
8          Remove X_{i−1} ∩ X_i from R(x)
9          return F_i
```

Correctness

For the algorithm to be correct we have to argue that the algorithm returns a set of nodes F_i that approximates the path union $\mathscr{P}(x, y, h, G)$. Here, approximating the path union means that $\mathscr{P}(x, y, h, G) \subseteq F_i$ and furthermore that for every node $v \in F_i$ the shortest path from x to y through v has length $O(h \log n)$.

Observe first that $B_i \subseteq B_{i+1}$ and $F_i \subseteq F_{i+1}$ for all $1 \leq i \leq \lceil \log m \rceil$. However it is not necessarily the case that $X_i \subseteq X_{i+1}$. We now show the invariant that the set $R(x)$ always contains all nodes that are at distance at most h from x. This is true initially as we initialize $R(x)$ to be the full graph G and we now show that it remains true because we only remove nodes with distance more than h from x. The argument is as follows: Consider a node $v \in X_{i-1} \cap X_i$ (which is necessary for v to be removed). (a) From $v \in X_{i-1} \subseteq B_{i-1}$ we know that there exists a path of length at most $(i-1)h$ from v to y in $R(x)$. (b) From $v \in X_i$ we know that $v \notin F_i$, i.e., there is no path from x to v of length at most h in B_i, which implies that there is no path from x to y

through v of length at most ih in $R(x)$. From (a) and (b) it follows that there is no path from x to v of length at most h in $R(x)$. By the invariant any such path of G would be contained in $R(x)$. Therefore it follows that no such path exists in G. Thus, we have proved the following lemma.

Lemma 4.5.1. *Let $1 \leq i \leq \lceil \log m \rceil + 1$ and assume that $R(x)$ contains every node v such that $d_G(x, v) \leq h$. Then for every node $v \in X_{i-1} \cap X_i$, we have $d_G(x, v) > h$.*

We now complete the correctness proof by first showing that the set of nodes returned by the algorithm approximates the path union.

Lemma 4.5.2. *Procedure 4.3 returns a set of nodes F_{i^*} such that $\mathscr{P}(x, y, h, G) \subseteq F_{i^*}$, where i^* denotes the final value of i in the algorithm.*

Proof. We first argue that the algorithm actually returns some set of nodes F_{i^*}. If $m_1 = 0$, the algorithm returns the set F_1 (as $m_0 = 0$). If $m_1 \geq 1$, the algorithm guarantees that $m_{i^*} \geq 2^{i^*-1}$. As m_{i^*} counts the size of a set of edges and the total number of edges is at most m, the condition $m_i \leq 2m_{i-1}$ must eventually be fulfilled for some $2 \leq i \leq \lceil \log m \rceil + 1$.

Note that $F_j \subseteq F_{j+1}$ for every $1 \leq j \leq i^* - 1$. Therefore it is sufficient to show that $\mathscr{P}(x, y, h, G) \subseteq F_1$. Let $v \in \mathscr{P}(x, y, h, G)$, which means that v lies on a path π from x to y of length at most h. For every node v' on π we have $d_G(x, v') \leq h$, which by the invariant means that $v' \in R(x)$. Thus, the whole path π is contained in $G|R(x)$. Therefore $d_{G|R(x)}(v', y) \leq h$ for every node v' on π which means that π is contained in $G|B_1$. Then clearly we also have $d_{G|B_1}(x, v) \leq h$ which means that $v \in F_1$. □

Lemma 4.5.3. *Let F_{i^*} denote the set of nodes returned by Procedure 4.3. For every node $v \in F_{i^*}$, $d_G(x, v) + d_G(v, y) \leq (\log m + 3)h$.*

Proof. By the definition of F_{i^*} we have $d_{G|B_{i^*}}(x, v) \leq h$. Clearly, $d_G(x, v) \leq d_{G|B_{i^*}}(x, v)$ and thus $d_G(x, v) \leq h$. As $F_{i^*} \subseteq B_{i^*}$ we also have $d_{G|B_{i^*}}(v, y) \leq ih \leq (\lceil \log m \rceil + 1)h \leq (\log m + 2)h$. Clearly, $d_G(v, y) \leq d_{G|B_{i^*}}(v, y)$ and thus $d_G(v, y) \leq (\log m + 2)h$. It follows that $d_G(x, v) + d_G(v, y) \leq (\log m + 3)h$. □

Running Time Analysis

Lemma 4.5.4. *The running time of Procedure 4.3 is*

$$O(|E(F_{i^*}, F_{i^*})| + |E(X_{i^*-1} \cap X_{i^*}, R(x))| + |E(R(x), X_{i^*-1} \cap X_{i^*})|)$$

where i^* denotes the final value of i in the algorithm, F_{i^*} is the set of nodes returned by the algorithm, and $X_{i^*-1} \cap X_{i^*}$ is the set of nodes the algorithm removes from $R(x)$.

Proof. The running time in iteration $j \leq i^*$ is $O(|E(B_j, B_j)|)$ as this is the cost of the breadth-first-search performed to compute B_j and the cost of the breadth-first search

performed to compute F_j is dominated by this cost. In iteration i^*, the running time is proportional to

$$|E(B_{i^*}, B_{i^*})| + |E(X_{i^*-1} \cap X_{i^*}, R(x))| + |E(R(x), X_{i^*-1} \cap X_{i^*})|$$

as we additionally remove the nodes in $X_{i^*-1} \cap X_{i^*}$ from $R(x)$. Thus the total running time is proportional to

$$\sum_{1 \leq j \leq i^*} |E(B_j, B_j)| + |E(X_{i^*-1} \cap X_{i^*}, R(x))| + |E(R(x), X_{i^*-1} \cap X_{i^*})| \,.$$

Remember that $m_j = |E(B_j, B_j)|$ for all $1 \leq j \leq i^*$. By checking the size bound in Line 7 of Procedure 4.3 we have $m_j > 2m_{j-1}$ for all $1 \leq j \leq i^* - 1$ and $m_{i^*} \leq 2m_{i^*-1}$. By repeatedly applying the first inequality it follows that $\sum_{1 \leq j \leq i^*-1} m_j \leq 2m_{i^*-1}$. Therefore we get

$$\begin{aligned}\sum_{1 \leq j \leq i^*} |E(B_j, B_j)| = \sum_{1 \leq j \leq i^*} m_j = \sum_{1 \leq j \leq i^*-1} m_j + m_{i^*} \\ \leq 2m_{i^*-1} + 2m_{i^*-1} = 4m_{i^*-1} = 4|E(B_{i^*-1}, B_{i^*-1})|\end{aligned}$$

and thus the running time is proportional to

$$E(B_{i^*-1}, B_{i^*-1}) + |E(X_{i^*-1} \cap X_{i^*}, R(x))| + |E(R(x), X_{i^*-1} \cap X_{i^*})| \,.$$

Now remember that $F_{i^*-1} \subseteq F_{i^*}$ and $B_{i^*-1} \subseteq B_{i^*}$ and observe that $B_{i^*-1} = (X_{i^*-1} \cap X_{i^*}) \cup F_{i^*}$:

$$\begin{aligned}(X_{i^*-1} \cap X_{i^*}) \cup F_{i^*} &= (X_{i^*-1} \cup F_{i^*}) \cap (X_{i^*} \cup F_{i^*}) \\ &= (B_{i^*-1} \setminus F_{i^*-1} \cup F_{i^*}) \cap (B_{i^*} \setminus F_{i^*} \cup F_{i^*}) = (B_{i^*-1} \cup F_{i^*}) \cap B_{i^*} = B_{i^*} \,.\end{aligned}$$

It follows that

$$\begin{aligned}E(B_{i^*-1}, B_{i^*-1}) &\subseteq E(F_{i^*}, F_{i^*}) \cup E(X_{i^*-1} \cap X_{i^*}, X_{i^*-1} \cap X_{i^*}) \\ &\quad \cup E(X_{i^*-1} \cap X_{i^*}, F_{i^*}) \cup E(F_{i^*}, X_{i^*-1} \cap X_{i^*}) \\ &\subseteq E(F_{i^*}, F_{i^*}) \cup E(X_{i^*-1} \cap X_{i^*}, R(x)) \cup E(R(x), X_{i^*-1} \cap X_{i^*}) \,.\end{aligned}$$

Therefore the running time is proportional to

$$|E(F_{i^*}, F_{i^*})| + |E(X_{i^*-1} \cap X_{i^*}, R(x))| + |E(R(x), X_{i^*-1} \cap X_{i^*})| \,. \qquad \square$$

As each node is removed from $R(x)$ at most once, the time spent on *all* calls of Procedure 4.3 is $O(m)$ plus the sizes of the subgraphs induced by the approximate path unions returned in each call.

4.5.2 Reachability via Center Graph

We now show how to combine the approximate path union data structure with a hierarchical approach to get an improved decremental reachability algorithm for dense graphs. The algorithm has a parameter $1 \leq k \leq \log n$ and for each $1 \leq i \leq k$ a parameter $c_i \leq n$. We determine suitable choices of these parameters in Section 4.5.2. For each $1 \leq i \leq k-1$, our choice will satisfy $c_i \geq c_{i+1}$ and $c_i = \hat{O}(c_{i+1})$. Furthermore, we set $h_i = (3 + \log m)^{i-1} n/c_1$ for $1 \leq i \leq k$. At the initialization, the algorithm determines sets of nodes $C_1 \supseteq C_2 \supseteq \ldots \supseteq C_k$ such that $s, t \in C_1$ as follows. For each $1 \leq i \leq k$, we sample each node of the graph with probability $ac_i \ln n/n$ (for a large enough constant a), where the value of c_i will be determined later. The set C_i then consists of the sampled nodes, and if $i \leq k-1$, it additionally contains the nodes in C_{i+1}. For every $1 \leq i \leq k$ we call the nodes in C_i i-centers. In the following we describe an algorithm for maintaining pairwise reachability between all 1-centers.

Algorithm Description

Data Structures The algorithm uses the following data structures:

- For every i-center x and every $i \leq j \leq k$ an approximate path union data structure (see Section 4.5.1) with parameter h_j.
- For every k-center x an incoming and outgoing ES-tree of depth h_k in G.
- For every pair of an i-center x and a j-center y such that $l := \max(i, j) \leq k-1$, a set of nodes $Q(x, y, l) \subseteq V$. Initially, $Q(x, y, l)$ is empty and at some point the algorithm might compute $Q(x, y, l)$ using the approximate path union data structure of x.
- For every pair of an i-center x and a j-center y such that $l := \max(i, j) \leq k-1$ an ES-tree of depth h_l from x in $Q(x, y, l)$.
- For every pair of an i-center x and a j-center y such that $l := \max(i, j) \leq k-1$ a set of $(l+1)$-centers certifying that x can reach y.

Certified Reachability Between Centers (Links) The algorithm maintains the following limited path information between centers, called *links*, in a top-down fashion. Let x be a k-center and let y be an i-center for some $1 \leq i \leq k-1$. The algorithm links x to y if and only if y is contained in the outgoing ES-tree of depth h_k of x. Similarly the algorithm links y to x if and only if y is contained in the incoming ES-tree of depth h_k of x. Let x be an i-center and let y be a j-center such that $l := \max(i, j) \leq k-1$. If there is an $(l+1)$-center z such that x is linked to z and z is linked to y, the algorithm links x to y (we also say that z links x to y or that z certifies that x can reach y). Otherwise, the algorithm computes $Q(x, y, l)$ using the approximate path union data structure of x and starts to maintain an ES-tree from x up to depth h_l in $G|Q(x, y, l)$. It links x to y if and only if y is contained in

the ES-tree of x. Using a list of centers z certifying that x can reach y, maintaining the links between centers is straightforward.

Center Graph The algorithm maintains a graph called *center graph*. Its nodes are the 1-centers and it contains the edge (x, y) if and only if x is linked to y. The algorithm maintains the transitive closure of the center graph. A query asking whether a center y is reachable from a center x in G is answered by checking the reachability in the center graph. As s and t are 1-centers this answers s-t reachability queries.

Correctness

For the algorithm to be correct we have to show that there is a path from s to t in the center graph if and only if there is a path from s to t in G. We will in fact show in more generality that this is the case for any pair of 1-centers.

Lemma 4.5.5. *For every pair of an i-center x and a j-center y, if x is linked to y, then there is a path from x to y in G.*

Proof. The proof is by induction on $l = \max(i, j)$. As x is linked to y, one of the following three cases applies:

1. $j = k$ and x is contained in the incoming ES-tree of depth h_k of y in G.
2. $i = k$ and y is contained in the outgoing ES-tree of depth h_k of x in G.
3. There is an $(l + 1)$-center z such that x is linked to z and z is linked to y.
4. y is contained in the ES-tree of depth h_l of x in $G|Q(x, y, l)$.

In the first two cases there obviously is a path from x to y in G by the correctness of the ES-tree. In the third case we may apply the induction hypothesis and find a path from x to z as well as a path from z to y in G. The concatenation of these paths is a path from x to y in G. In the fourth case we know by the correctness of the ES-tree that there is a path from x to y in $G|Q(x, y, l)$ and therefore also in G. □

Lemma 4.5.6. *For every pair of an i-center x and a j-center y, if $d_G(x, y) \leq h_l$, then x is linked to y.*

Proof. Set $l = \max(i, j)$. If $l = k$, then assume that $l = i$ (the proof for $l = j$ is symmetric). Since $d_G(x, y) \leq h_k$, y is contained in the outgoing ES-tree of depth h_k of x by the correctness of the ES-tree. Thus, x is linked to y.

If $l \leq k - 1$ and there is an $(l + 1)$-center z such that x is linked to z and z is linked to y, then also x is linked to y. If this is not the case, then the algorithm has computed $Q(x, y, l)$ using the approximate path union data structure and at that time we have $\mathscr{P}(x, y, h_l, G) \subseteq Q(x, y, l)$ by Lemma 4.5.2. By Lemma 4.2.8 the set $Q(x, y, l)$ contains $\mathscr{P}(x, y, h_l, G)$, also in the current version of G. As $d_G(x, y) \leq h_l$,

all nodes on the shortest path from x to y in G are contained in $\mathscr{P}(x, y, h_l, G)$ and thus in $Q(x, y, l)$. Therefore the ES-tree from x up to depth h_l in $G|Q(x, y, l)$ contains y which means that x is linked to y. □

Lemma 4.5.7. *For every pair of* 1*-centers x and y, there is a path from x to y in the center graph if and only if there is a path from x to y in G.*

Proof. Assume that there is a path from x to y in the center graph. Every edge (x', y') in the center graph (where x' and y' are 1-centers) can only exist if the algorithm has linked x' to y'. By Lemma 4.5.5 this implies that there is a path from x' to y' in G. By concatenating all these paths we obtain a path from x to y in G.

Now assume that there is a path from x to y in G. Consider the shortest path π from x to y in G and any two consecutive 1-centers x' and y' on π. By the initial random sampling of 1-centers, every shortest path consisting of $h_1 - 1$ edges contains a 1-center whp (Lemma 1.3.2) and thus $d_G(x', y') \leq h_1$. By Lemma 4.5.6 the algorithm has linked x' to y' and thus there is an edge from x' to y' in the center graph. As such an edge exists for all consecutive 1-centers on π, there is a path from x to y in the center graph. □

Running Time Analysis

The key to the efficiency of the algorithm is to bound the size of the graphs $Q(x, y, l)$.

Lemma 4.5.8. *Let x be an i-center and let y be a j-center such that $l := \max(i, j) \leq k-1$. If x is not linked to y by an $(l+1)$-center, then $Q(x, y, l)$ contains at most n/c_{l+1} nodes with high probability.*

Proof. Suppose that $Q(x, y, l)$ contains more than n/c_{l+1} nodes. Then by the random sampling of centers, $Q(x, y, l)$ contains an $(l+1)$-center z with high probability by Lemma 1.3.2. By Lemma 4.5.3 we have $d_G(x, z) + d_G(z, y) \leq (3 + \log m)h_l = h_{l+1}$ and thus $d_G(x, z) \leq h_{l+1}$ and $d_G(z, y) \leq h_{l+1}$. It follows that x is linked to z and z is linked to y by Lemma 4.5.6. But then x is linked to y, contradicting our assumption. □

With the help of this lemma we first analyze the running time of each part of the algorithm and argue that our choice of parameters gives the desired total update time.

Parameter Choice We carry out the running time analysis with regard to two parameters $1 \leq b \leq c \leq n$ which we will set at the end of the analysis. We set

$$k = \left\lceil \frac{\log(c/b)}{\sqrt{\log n \cdot \log\log n}} \right\rceil + 1$$

$c_k = b$ and $c_i = 2^{\sqrt{\log n \cdot \log\log n}} c_{i+1} = \hat{O}(c_{i+1})$ for $1 \le i \le k-1$. Note that the number of i-centers is $\tilde{O}(c_i)$ in expectation. Observe that

$$(3+\log m)^{k-1} = O((\log n)^k) \le O((\log n)^{\sqrt{\log n/\log\log n}}) \\ = O(2^{\sqrt{\log n \cdot \log\log n}}) = O(n^{\sqrt{\log\log n/\log n}}) = O(n^{o(1)}).$$

Furthermore we have

$$c_1 = \left(2^{\sqrt{\log n \cdot \log\log n}}\right)^{k-1} c_k \ge 2^{\log(c/b)} b = \frac{c}{b} \cdot b = c$$

and by setting $k' = (\log(c/b))/(\sqrt{\log n \cdot \log\log n})$ we have $k \le k' + 2$ and thus

$$c_1 = \left(2^{\sqrt{\log n \cdot \log\log n}}\right)^{k-1} c_k \le \left(2^{\sqrt{\log n \cdot \log\log n}}\right)^{k'+1} c_k = 2^{\sqrt{\log n \cdot \log\log n}} c = \hat{O}(c).$$

Remember that $h_i = (3+\log m)^{i-1} n/c_1$ for $1 \le i \le k$. Therefore we have $h_i = \hat{O}(n/c_1) = \hat{O}(n/c)$.

Maintaining ES-Trees For every k-center we maintain an incoming and an outgoing ES-tree of depth h_k, which takes time $O(mh_k)$. As there are $\tilde{O}(c_k)$ k-centers, maintaining all these trees takes time $\tilde{O}(c_k m h_k) = \hat{O}(bmn/c)$.

For every i-center x and every j-center y such that $l := \max(i,j) \le k-1$, we maintain an ES-tree up to depth h_l in $G|Q(x,y,l)$. By Lemma 4.5.8 $Q(x,y,l)$ has at most n/c_{l+1} nodes and thus $G|Q(x,y,l)$ has at most n^2/c_{l+1}^2 edges. Maintaining this ES-tree therefore takes time $O((n^2/c_{l+1}^2) \cdot h_l) = \hat{O}(n^2/c_{l+1}^2 (n/c_1)) = \hat{O}(n^3/(c_1 c_{l+1}^2))$. In total, maintaining all these trees takes time

$$\hat{O}\left(\sum_{1\le i\le k-1}\sum_{1\le j\le i} c_i c_j \frac{n^3}{c_1 c_{i+1}^2}\right) = \hat{O}\left(\sum_{1\le i\le k-1}\sum_{1\le j\le i} \frac{c_i c_1 n^3}{c_{i+1} c_1 c_k}\right) \\ = \hat{O}\left(\sum_{1\le i\le k-1}\sum_{1\le j\le i} \frac{n^3}{c_k}\right) = \hat{O}\left(k^2 \frac{n^3}{c_k}\right) = \hat{O}\left(\frac{n^3}{b}\right).$$

Computing Approximate Path Unions For every i-center x and every $i \le j \le k$ we maintain an approximate path union data structure with parameter h_j. As a consequence of Lemma 4.5.4 this data structures has a total running time of $O(m)$ and an additional cost of $O(|E(G|Q_j(x,y))|)$ each time the approximate path union $Q(x,y,j)$ is computed for some j-center y. By Lemma 4.5.8 the number of nodes of $Q(x,y,j)$ is n/c_{j+1} with high probability and thus its number of edges is n^2/c_{j+1}^2 Therefore, computing all approximate path unions takes time

$$\tilde{O}\left(\sum_{1\le i\le k-1}\sum_{i\le j\le k}\left(c_i m + c_i c_j \frac{n^2}{c_{j+1}^2}\right)\right) = \tilde{O}\left(\sum_{1\le i\le k-1}\sum_{i\le j\le k}\left(c_1 m + \frac{c_1 c_j n^2}{c_{j+1} c_k}\right)\right) \\ = \hat{O}\left(\sum_{1\le i\le k-1}\sum_{i\le j\le k}\left(c_1 m + \frac{c_1 n^2}{c_k}\right)\right) = \hat{O}(k^2 c_1 m + k^2 c_1 n^2/c_k) = \hat{O}(cm + cn^2/b).$$

Maintaining Links Between Centers For each pair of an i-center x and a j-center y there are at most $\tilde{O}(c_{l+1})$ $(l+1)$-centers that can possibly link x to y. Each such $(l+1)$-center is added to and removed from the list of $(l+1)$-centers linking x to y at most once. Thus, the total time needed for maintaining all these links is $\tilde{O}(\sum_{1\le i\le k-1}\sum_{1\le j\le i} c_i c_j c_{i+1}) = \tilde{O}(k^2 c_1^3) = \tilde{O}(c^3)$.

Maintaining Transitive Closure in Center Graph The center graph has $\tilde{O}(c_1)$ nodes and thus $\tilde{O}(c_1^2)$ edges. During the algorithm edges are only deleted from the center graph and never inserted. Thus we can use known $O(mn)$-time decremental algorithms for maintaining the transitive closure [90, 114] in the center graph in time $\tilde{O}(c_1^3) = \tilde{O}(c^3)$.

Total Running Time Since the term cn^2/b is dominated by the term n^3/b, we obtain a total running time of $\hat{O}\left(bmn/c + n^3/b + cm + c^3\right)$. By setting $b = n^{5/3}/m^{2/3}$ and $c = n^{4/3}/m^{1/3}$ the running time is $\hat{O}(m^{2/3}n^{4/3} + n^4/m)$ and by setting $b = n^{9/7}/m^{3/7}$ and $c = m^{1/7}n^{4/7}$ the running time is $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7})$.

Decremental Single-Source Reachability

The algorithm above works for a set of randomly chosen centers. Note that the algorithm stays correct if we add any number of nodes to C_1, thus increasing the number of 1-centers for which the algorithm maintains pairwise reachability. If the number of additional centers does not exceed the expected number of randomly chosen centers, then the same running time bounds still apply. Thus, from the analysis above, we obtain the following result.

Theorem 4.5.9. *Let $S \subseteq V$ be a set of nodes. If $|S| \le n^{4/3}/m^{1/3}$, then there is a decremental algorithm for maintaining pairwise reachability between all nodes in S with constant query time and a total update time of $\hat{O}(m^{2/3}n^{4/3}+n^4/m)$. If $|S| \le m^{1/7}n^{4/7}$, then there is a decremental algorithm for maintaining pairwise reachability between all nodes in S with constant query time and a total update time of $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7})$.*

Using the reduction of Theorem 4.2.14 this also gives us a single-source reachability algorithm.

Corollary 4.5.10. *There is a decremental single-source reachability algorithm with constant query time and a total update time of $\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$.*

Proof. First, set $c_1 = n^{4/3}/m^{1/3}$ and observe that by Theorem 4.5.9 we have an algorithm that can maintain reachability from a source to c_1 sinks with a total update time of $\hat{O}(m^{2/3}n^{4/3} + n^4/m)$. By Theorem 4.2.14 this implies a decremental single-source reachability algorithm with a total update time of $\hat{O}(m^{2/3}n^{4/3} + n^4/m + mn/c_1)$. The same argument gives a decremental single-source reachability algorithm with a total update time of $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7} + mn/c_2)$ where $c_2 = m^{1/7}n^{4/7}$.

If $m \geq n^{8/5}$ we have $m^{2/3}n^{4/3} \leq n^4/m$ and $mn/c_1 = m^{4/3}/n^{1/3} \leq m^{2/3}n^{4/3}$. If $m \leq n^{8/5}$ we have $m^{8/7}n^{4/7} \leq m^{3/7}n^{12/7}$ and $mn/c_2 = m^{6/7}/n^{3/7} \leq m^{3/7}n^{12/7}$. Thus, by running the first algorithm if $m \geq n^{8/5}$ and the second one if $m < n^{8/5}$ we obtain a total update time of $\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$ (Note that $m^{3/7}n^{12/7} \leq m^{2/3}n^{4/3}$ if and only if $m \geq n^{8/5}$). □

Furthermore, the reduction of Theorem 4.2.15 gives a decremental algorithm for maintaining strongly connected components.

Corollary 4.5.11. *There is a decremental SCC algorithm with constant query time and expected update time*

$$\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$$

that is correct with high probability against an oblivious adversary.

4.6 Conclusion

In this chapter we have presented decremental algorithms for maintaining approximate single-source shortest paths (and thus also single-source reachability) with constant query time and a total update time of $o(mn)$. This first step motivates the search for faster and simpler algorithms for this problem. Further motivation comes from the fact that decremental approximate SSSP in *undirected* graphs can be solved with almost linear total update time (Chapter 3).

Given recent progress in lower bounds for dynamic graph problems [1, 70, 104], it might also be possible that an almost linear update time is not possible for decremental approximate SSSP and SSR in directed graphs. However, it might be challenging to prove this as all known constructions give the same lower bounds for both the incremental and the decremental version of a problem. As incremental SSR has a total update time of $O(m)$ we cannot hope for lower bounds for decremental SSR using these known techniques. This only leaves the possibility of finding a lower bound for decremental approximate SSR or of developing stronger techniques. It also motivates studying the incremental approximate SSSP problem which is intuitively easier than its decremental counterpart, but no as easy as incremental SSR.

Finally, we ask whether it is possible to remove the assumption that the adversary is oblivious, i.e., to allow an *adaptive adversary* that may choose each new update or query based on the algorithm's previous answers. This would automatically be guaranteed by a deterministic algorithm.

CHAPTER 5

Sublinear-Time Maintenance of Breadth-First Spanning Trees in Partially Dynamic Networks

We study the problem of maintaining a *breadth-first spanning tree* (BFS tree) in *partially dynamic* distributed networks modeling a sequence of either failures or additions of communication links (but not both). We present deterministic $(1+\epsilon)$-approximation algorithms whose amortized time (over some number of link changes) is *sublinear* in D, the *maximum diameter* of the network.

Our technique also leads to a deterministic $(1+\epsilon)$-approximate incremental algorithm for single-source shortest paths (SSSP) in the sequential (usual RAM) model. Prior to our work, the state of the art was the classic *exact* algorithm of Even and Shiloach [49] that is optimal under some assumptions [70, 115]. Our result is the first to show that, in the incremental setting, this bound can be beaten in certain cases if some approximation is allowed.

5.1 Introduction

Complex networks are among the most ubiquitous models of interconnections between a multiplicity of individual entities, such as computers in a data center, human beings in society, and neurons in the human brain. The connections between these entities are constantly changing; new computers are gradually added to data centers, or humans regularly make new friends. These changes are usually *local* as they are known only to the entities involved. Despite their locality, they could affect the network *globally*; a single link failure could result in several routing path losses or destroy the network connectivity. To maintain its robustness, the network has to quickly respond to changes and repair its infrastructure. The study of such tasks has

been the subject of several active areas of research, including dynamic, self-healing, and self-stabilizing networks.

One important infrastructure in distributed networks is the *breadth-first spanning (BFS) tree* [94, 105]. It can be used, for instance, to approximate the network diameter and to provide a communication backbone for broadcast, routing, and control. In this chapter, we study the problem of maintaining a BFS tree from a root node on dynamic distributed networks. Our main interest is repairing a BFS tree as fast as possible after each topology change.

Model We model the communication network by the CONGEST model [105], one of the major models of (locality-sensitive) distributed computation. Consider a synchronous network of processors modeled by an undirected unweighted graph $G = (V, E)$, where nodes model the processors and edges model the bounded-bandwidth links between the processors. We let V and E denote the set of nodes and edges of G, respectively, and let s be a specified *root node*. For any node u and v, we denote by $d_G(u, v)$ the distance between u and v in G. The processors (henceforth, nodes) are assumed to have unique IDs of $O(\log n)$ bits and infinite computational power. Each node has limited topological knowledge; in particular, it only knows the IDs of its neighbors and knows *no* other topological information (such as whether its neighbors are linked by an edge or not). The communication is synchronous and occurs in discrete pulses, called *rounds*. All the nodes wake up simultaneously at the beginning of each round. In each round each node u is allowed to send an arbitrary message of $O(\log n)$ bits through each edge (u, v) that is adjacent to u, and the message will reach v at the end of the current round. There are several measures to analyze the performance of such algorithms, a fundamental one being the *running time*, defined as the worst-case number of rounds of distributed communication.

We model dynamic networks by a sequence of *attack* and *recovery* stages following the initial *preprocessing*. The dynamic network starts with a preprocessing on the initial network denoted by G_0, where nodes communicate on G_0 for some number of rounds. Once the preprocessing is finished, we begin the first attack stage where we assume that an adversary, who sees the current network G_0 and the states of all nodes, inserts and deletes an arbitrary number of edges in G_0. We denote the resulting network by G_1. This is followed by the first recovery stage where we allow nodes to communicate on G_1. After the nodes have finished communicating, the second attack stage starts, followed by the second recovery stage, and so on. For any algorithm, we let the *total update time* be the total number of rounds needed by nodes to communicate during all recovery stages. Let the *amortized update time* be the total time divided by q which is defined as the number of edges inserted and deleted. Important parameters in analyzing the running time are n, the number of nodes (which remains the same throughout all changes) and D, the *maximum diameter*, defined to be the maximum diameter among all networks in $\{G_0, G_1, \ldots\}$. If some network G_t is not connected, we define its diameter as the diameter of the connected component containing the root node. Note that $D \leq n$ according to this definition.

Following the convention from the area of (sequential) dynamic graph algorithms, we say that a dynamic network is *fully dynamic* if both insertions and deletions can occur in the attack stages. Otherwise, it is *partially dynamic*. Specifically, if only edge insertions can occur, it is an *incremental dynamic network*. If only edge deletions can occur, it is *decremental*.

Our model highlights two aspects of dynamic networks: (1) How quickly a network can recover its infrastructure after changes and (2) how edge failures and additions affect the network. These aspects have been studied earlier but we are not aware of any previous model identical to ours. To highlight these aspects, a few assumptions are inherent in our model. First, it is assumed that the network remains static in each recovery stage. This assumption is often used (e.g., [57, 84, 87, 96]) and helps to emphasize the running time aspect of dynamic networks. Also note that we assume that the network is synchronous, but our algorithms will also work in an asynchronous model under the same asymptotic time bounds, using a synchronizer [11, 105]. Furthermore, we consider amortized update time which is similar in spirit to the amortized communication complexity heavily studied earlier (e.g., [13]). Finally, the results in this chapter are on partially dynamic networks. While fully dynamic algorithms are more desirable, we believe that the partially dynamic setting is worth studying, for two reasons. The first reason, which is our main motivation, comes from an experience in the study of sequential dynamic algorithms, where insights from the partially dynamic setting often lead to improved fully dynamic algorithms. Moreover, partially dynamic algorithms can be useful in cases where one type of changes occurs much more frequently than the other type. For example, links constantly fail in physical networks, and it might not be necessary that the network has to be fixed (by adding a link) immediately. Instead, the network can try to maintain its infrastructures under a sequence of failures until the quality of service cannot be guaranteed anymore, e.g., the network diameter becomes too large. Partially dynamic algorithms for maintaining a BFS tree, which in turn maintains the approximate network diameter, are quite suitable for this type of applications.

Problem We are interested in maintaining an approximate BFS tree. Our definition of approximate BFS trees below is a modification of the definition of BFS trees in [105, Definition 3.2.2].

Definition 5.1.1 (Approximate BFS tree)**.** *For any $\alpha \geq 1$, an* α-approximate BFS tree *of a graph G with respect to a given root s is a spanning tree T such that for every node v connected to s, $d_T(v, s) \leq \alpha d_G(v, s)$. If $\alpha = 1$, then T is an* (exact) BFS tree.

Note that, for any spanning tree T of G, $d_T(v, s) \geq d_G(v, s)$. Our goal is to maintain an approximate BFS tree T_t at the end of each recovery stage t in the sense that every node v knows its approximate distance to the preconfigured root s in G_t and, for each neighbor u of v, v knows if u is its parent or child in T_t. Note that for convenience we will usually consider $d_G(v, s)$, the distance of v *to* the root, instead

of $d_G(s, v)$, the distance of v *from* the root. In an undirected graph both values are the same.

Naive Algorithm As a toy example, observe that we can maintain a BFS tree simply by recomputing a BFS tree from scratch in each recovery stage. By using the standard algorithm (see, e.g., [94, 105]), we can do this in time $O(D_t)$, where D_t is the diameter of the graph G_t. Thus, the update time is $O(D)$.

Results Our main results are partially dynamic algorithms that break the naive update time of $O(D)$ in the long term. They can maintain, for any constant $0 < \epsilon \leq 1$, a $(1+\epsilon)$-approximate BFS tree in time that is *sublinear in* D when amortized over $\omega(n/D)$ edge changes. To be precise, the amortized update time over q edge changes is

$$O\left(\frac{n^{1/3}D^{2/3}}{\epsilon^{2/3}q^{1/3}}\right) \quad \text{and} \quad O\left(\frac{n^{1/5}D^{4/5}}{\epsilon q^{1/5}}\right)$$

in the incremental and decremental setting, respectively. For the particular case of $q = \Omega(n)$, we get amortized update times of $O(D^{2/3}/\epsilon^{2/3})$ and $O(D^{4/5}/\epsilon)$ for the incremental and decremental cases, respectively. Our algorithms do not require any prior knowledge about the dynamic network, e.g., D and q. We have formulated the algorithms for a setting that allows insertions or deletions of edges. The guarantees of our algorithms also hold when we allow insertions or deletions of *nodes*, where the insertion of a node also inserts all its incident edges and the deletion of a node also deletes all its incident edges. In the running time, the parameter q then counts the number of node insertions or node deletions, respectively.

We note that, while there is no previous literature on this problem, one can parallelize the algorithm of Even and Shiloach [49] (see also [79, 115]) to obtain an amortized update time of $O(nD/q+1)$ over q changes in both the incremental and the decremental setting. This bound is sublinear in D when $q = \omega(n)$. Our algorithms give a sublinear time guarantee for a smaller number of changes, especially in applications where D is large. They are faster than the Even-Shiloach algorithm when $q = \omega(\epsilon n\sqrt{D})$ (incremental) and $q = \omega(\epsilon^{7/12}nD^{1/6})$ (decremental).

In the sequential (usual RAM) model, our technique also gives an $(1+\epsilon)$-approximation algorithm for the incremental single-source shortest paths (SSSP) problem with an amortized update time of $O(mn^{1/4}\log n/\sqrt{\epsilon q})$ per insertion and $O(1)$ query time, where m is the number of edges in the final graph, and q is the number of edge insertions. Prior to this result, only the classic exact algorithm of Even and Shiloach [49] from the 80s, with $O(mn/q)$ amortized update time, was known. No further progress has been made in the last three decades. Roditty and Zwick [115] provided an explanation for this by showing that the algorithm of Even and Shiloach [49] is likely to be the fastest combinatorial *exact* algorithm, assuming that there is no faster combinatorial algorithm for Boolean matrix multiplication. More recently Henzinger et al. [70] showed that by assuming a different conjecture, called Online Matrix-Vector Multiplication Conjecture, this statement can be extended to any

algorithm (including non-combinatorial ones). Bernstein and Roditty [24] showed that, in the decremental setting, this bound can be broken if some approximation is allowed. Our result is the first one of the same spirit in the *incremental* setting; i.e., we break the bound of Even and Shiloach for the case $q = o(n^{3/2})$, which in particular applies when $m = o(n^{3/2})$. The techniques introduced in this chapter (first presented in the preliminary version [69]) together with techniques from Chapter 2 also led to a decremental algorithm [63] that improves the result of [24]. We finally obtained a near-optimal algorithm in the decremental setting [64] (see Chapter 3), which is a significant improvement over [24]. We note that the algorithm in this chapter is still the fastest deterministic algorithm when $q = o(n^{3/2})$ (and thus when $m = o(n^{3/2})$). In fact, there is no deterministic algorithm faster than Even and Shiloach's algorithm, even for some range of parameters, except this one.

Related Work The problem of computing on dynamic networks is a classic problem in the area of distributed computing, studied from as early as the 70s; see, e.g., [13] and references therein. The main motivation is that dynamic networks better capture real networks, which experience failures and additions of new links. There is a large number of models of dynamic networks in the literature, each emphasizing different aspects of the problem. Our model closely follows the model of the sequential setting and, as discussed earlier, highlights the amortized update time aspect. It is closely related to the model in [86] where the main goal is to optimize the amortized update time using static algorithms in the recovery stages. The model in [86] is still slightly different from ours in terms of allowed changes. For example, the model in [86] considers weighted networks and allows small weight changes but no topological changes; moreover, the message size can be unbounded (i.e., the static algorithm in the recovery stage operates under the so-called LOCAL model). Another related model the *controlled dynamic model* (e.g., [5, 85]) where the topological changes do not happen instantaneously but are delayed until getting a permit to do so from the resource controller. Our algorithms can be used in this model as well since we can delay the changes until each recovery stage is finished. Our model is similar to, and can be thought of as a combination of, two types of models: those in, e.g., [57, 84, 87, 96] whose main interest is to determine how fast a network can recover from changes using static algorithms in the recovery stages, and those in, e.g., [4, 13, 43], which focus on the amortized cost per edge change. Variations of partially dynamic distributed networks have also been considered (e.g., [28, 29, 73, 107]).

The problem of constructing a BFS tree has been studied intensively in various distributed settings for decades (see [105, Chapter 5], [94, Chapter 4] and references therein). The studies were also extended to more sophisticated structures such as minimum spanning trees (e.g., [34, 42, 46, 53, 83, 88, 91, 92, 106]) and Steiner trees [78]. These studies usually focus on *static* networks, i.e., they assume that the network never changes and want to construct a BFS tree once, from scratch. While we are not aware of any results on maintaining a BFS tree on dynamic networks,

there are a few related results. Much attention (e.g., [13]) has previously been given to the problem of *maintaining a spanning tree*. In a seminal paper by Awerbuch, Cidon, and Kutten [13], it was shown that the amortized message complexity of maintaining a spanning tree can be significantly smaller than the cost of the previous approach of recomputing from scratch [4].[1] Our result is in the same spirit as [13] in breaking the cost of recomputing from scratch. An attempt to maintain spanning trees of small diameter has also motivated a problem called *best swap*. The goal is to replace a failed edge in the spanning tree by a new edge in such a way that the diameter is minimized. This problem has recently gained considerable attention in both sequential (e.g., [7, 35, 54, 75, 76, 99, 100, 116]) and distributed (e.g., [51, 55]) settings.

In the sequential dynamic graph algorithms literature, a problem similar to ours is the single-source shortest paths (SSSP) problem on undirected graphs. This problem has been studied in partially dynamic settings and has applications to other problems, such as all-pairs shortest paths and reachability. As we have mentioned earlier, the classic bound of [49], which might be optimal [70, 115], has recently been improved by randomized decremental approximation algorithms [24, 63, 64], and we achieve a similar result in the incremental setting with a deterministic algorithm. Since our algorithms use the algorithm of [49] as a subroutine, we formally state its guarantees in the following. As mentioned above, this algorithm has not been considered in the distributed model before, but its analysis from the sequential model immediately carries over to the distributed model.[2] Since we will need this result later in this chapter, we state it here (see also Theorem 1.3.1)

Theorem 5.1.2 ([49])**.** *There is a partially dynamic algorithm for maintaining a shortest paths tree from a given root node up to depth $X \leq n$ under edge insertions (deletions) in an unweighted, undirected graph. Its total running time over q insertions (deletions) is $O(mX)$ in the sequential model and $O(n \min(X, D) + q)$ in the distributed model.*

5.2 Main Technical Idea

All our algorithms are based on a simple idea of modifying the algorithm of Even and Shiloach [49] with lazy updates, which we call *lazy Even-Shiloach tree*. Implementing this idea on different models requires modifications to cope with difficulties and to maximize efficiency. In this section, we explain the main idea by sketching a

[1] A variant of their algorithm was later implemented as a part of the PARIS networking project at IBM [30] and slightly improved [89].

[2] In the sequential model, the algorithm has to perform work proportional to the degree of each node whose distance to the root decreases (increases). As each node's distance to the root can increase (decrease) at most X times, the total running time is $O(mX)$. In the distributed model, sending a message to all neighbors takes one round and thus we only charge constant time to each level increase (decrease) of a node, resulting in a total time of $O(n \min(X, D) + q)$. The additional q comes from the fact that we have to spend constant time per insertion (deletion), which in the sequential model is dominated by other running time aspects.

simple algorithm and its analysis for the incremental setting in the sequential and the distributed model. We start with an algorithm that has *additive error*: Let κ and δ be parameters. For every recovery stage t, we maintain a tree T_t such that $d_{T_t}(v,s) \leq d_{G_t}(v,s) \leq d_{T_t}(v,s)+\kappa\delta$ for every node v. We will do this by recomputing a BFS tree from scratch repeatedly, specifically $O(q/\kappa + nD/\delta^2)$ times during q updates.

During the preprocessing, our algorithm constructs a BFS tree of G_0, denoted by T_0. This means that every node u knows its parent and children in T_0 and the value of $d_{T_0}(u,s)$. Suppose that, in the first attack stage, an edge is inserted, say (u,v) where $d_{G_0}(u,s) > d_{G_0}(v,s)$. As a result, the distance from u to s might decrease, i.e. $d_{G_1}(u,s) < d_{G_0}(u,s)$. In this case, the distances from s to some other nodes (e.g., the children of v in T_0) could decrease as well, and we may wish to recompute the BFS tree. Our approach is to do this *lazily*: We recompute the BFS tree only when the distance from v to s decreases by at least δ; otherwise, we simply do nothing! In the latter case, we say that v *is lazy*. Additionally, we regularly "clean up" by recomputing the BFS tree after each κ insertions.

To prove an additive error of $\kappa\delta$, observe that errors occur for this single insertion only when v is lazy. Intuitively, this causes an additive error of δ since we could have decreased the distance of v and other nodes by at most δ, but we did not. This argument can be extended to show that if we have i lazy nodes, then the additive error will be at most $i\delta$. Since we do the cleanup each κ insertions, the additive error will be at most $\kappa\delta$ as claimed.

To bound the number of BFS tree recomputations, first observe that the cleanup clearly contributes $O(q/\kappa)$ recomputations in total, over q insertions. Moreover, a recomputation also could be caused by some node v, whose distance to s decreases by at least δ. Since every time a node v causes a recomputation, its distance decreases by at least δ, and $d_{G_0}(v,s) \leq D$, v will cause the recomputation at most D/δ times. This naive argument shows that there are nD/δ recomputations (caused by n different nodes) in total. This analysis is, however, *not* enough for our purpose. A tighter analysis, which is crucial to all our algorithms relies on the observation that when v causes a recomputation, the distance from v's neighbor, say v', to s also decreases by at least $\delta - 1$. Similarly, the distance of v''s neighbor to s decreases by at least $\delta - 2$, and so on. This leads to the conclusion that one recomputation corresponds to $(\delta + (\delta-1) + (\delta-2) + \ldots) = \Omega(\delta^2)$ distance decreases. Thus, the number of recomputations is at most nD/δ^2. Combining the two bounds, we get that the number of BFS tree computations is $O(q/\kappa + nD/\delta^2)$ as claimed above. We get a bound on the total time when we multiply this number by the time needed for a single BFS tree computation. In the sequential model this takes time $O(m)$, where m is the final number of edges, and in the distributed model this takes time $O(D)$, where D is the dynamic diameter of the network.

To convert the additive error into a multiplicative error of $(1+\epsilon)$, we execute the above algorithm only for nodes whose distances to s are greater than $\kappa\delta/\epsilon$. For other nodes, we can use the algorithm of Even and Shiloach [49] to maintain a BFS tree of depth $\kappa\delta/\epsilon$. This requires an additional time of $O(m\kappa\delta/\epsilon)$ in the sequential model and $O(n\kappa\delta/\epsilon)$ in the distributed model.

By setting κ and δ appropriately, the above incremental algorithm immediately gives total update times of $O(mn^{2/5}q^{2/5}/\epsilon^{2/5})$ and $O(q^{2/5}n^{3/5}D^{4/5}/\epsilon^{2/5})$ in the sequential and distributed model, respectively. To obtain the running time bounds claimed in the introduction of this chapter, we need one more idea called *layering*, where we use different values of δ and κ depending on the distance of each node to s. In the decremental setting, the situation is much more difficult, mainly because it is expensive for a node v to determine how much its distance to s has increased after a deletion. Moreover, unlike the incremental case, nodes cannot simply "do nothing" when an edge is deleted. We have to cope with this using several other ideas, e.g., constructing an virtual tree (in which edges sometimes represent paths).

5.3 Incremental Algorithm

In this section we present a framework for an incremental algorithm that allows up to q edge insertions and provides an additive approximation of the distances to a distinguished node s. Subsequently we will explain how to use this algorithm to get $(1+\epsilon)$-approximations in the sequential model and the distributed model, respectively. For simplicity we assume that the initial graph is connected. In Section 5.3.4 we explain how to remove this assumption.

5.3.1 General Framework

The algorithm (see Algorithm 5.1) works in *phases*. At the beginning of every phase we compute a BFS tree T_0 of the current graph, say G_0. Every time an edge (u, v) is inserted, the distances of some nodes to s in G might decrease. Our algorithm tries to be *as lazy as possible*. That is, when the decrease does not exceed some parameter δ, our algorithm keeps its tree T_0 and accepts an *additive error* of δ for every node. When the decrease exceeds δ, our algorithm starts a new phase and recomputes the BFS tree. It also starts a new phase after each κ edge insertions to keep the additive error limited to $\kappa\delta$. The algorithm will answer a query for the distance from a node x to s by returning $d_{G_0}(x, s)$, the distance from x to s at the beginning of the current phase. It can also return the path from x to s in T_0 of length $d_{G_0}(x, s)$. Besides δ and κ, the algorithm has a third parameter X which indicates up to which distance from s the BFS tree will be computed. In the following we denote by G_0 the state of the graph at the beginning of the current phase and by G we denote the current state of the graph after all insertions.

As we show below the algorithm gives the desired additive approximation by considering the shortest path of a node x to the root s in the current graph G. By the main rule in Line 4 of the algorithm, the inequality $d_{G_0}(u, s) \leq d_{G_0}(v, s) + \delta$ holds for every edge (u, v) that was inserted since the beginning of the current phase (otherwise a new phase would have been started). Since at most κ edges have been inserted, the additive error is at most $\kappa\delta$.

Algorithm 5.1: Incremental algorithm

```
1 Procedure Insert(u, v)
2     k ← k + 1
3     if k = κ then Initialize()
4     if d_{G_0}(u, s) > d_{G_0}(v, s) + δ then Initialize()
5 Procedure Initialize()                                  // Start new phase
6     k ← 0
7     Compute BFS tree T of depth X rooted at s and current distances d_{G_0}(·, s)
```

Lemma 5.3.1 (Additive Approximation)**.** *For every $\kappa \geq 1$ and $\delta \geq 1$, Algorithm 5.1 provides the following approximation guarantee for every node x such that $d_{G_0}(x, s) \leq X$:*

$$d_G(x, s) \leq d_{G_0}(x, s) \leq d_G(x, s) + \kappa\delta\,.$$

Proof. The algorithm can only provide the approximation guarantee for every node x such that $d_{G_0}(x, s) \leq X$ because other nodes are not contained in the BFS tree of the current phase. It is clear that $d_G(x, s) \leq d_{G_0}(x, s)$ because G is the result of inserting edges into G_0. In the following we argue about the second inequality.

Consider the shortest path $\pi = x_l, x_{l-1}, \dots x_0$ of length l from x to s in G (where $x_l = x$ and $x_0 = s$). Let S_j (with $0 \leq j \leq l$) denote the number of edges in the subpath $x_j, x_{j-1}, \dots, x_0$ that were inserted since the beginning of the current phase.

Claim 5.3.2. *For every integer j with $0 \leq j \leq l$ we have $d_{G_0}(x_j, s) \leq d_G(x_j, s) + S_j\delta$.*

Clearly the claim already implies the inequality we want to prove since there are at most κ edges that have been inserted since the beginning of the current phase which gives the following chain of inequalities:

$$d_{G_0}(x, s) = d_{G_0}(x_l, s) \leq d_G(x_l, s) + S_l\delta \leq d_G(x, s) + \kappa\delta\,.$$

Now we proceed with the inductive proof of the claim The induction base $j = 0$ is trivially true because $x_j = s$. Now consider the induction step where we assume that the inequality holds for j and we have to show that it also holds for $j + 1$.

Consider first the case that the edge (x_{j+1}, x_j) is one of the edges that have been inserted since the beginning of the current phase. By the rule of the algorithm we know that $d_{G_0}(x_{j+1}, s) \leq d_{G_0}(x_j, s) + \delta$ and by the induction hypothesis we have $d_{G_0}(x_j, s) \leq d_G(x_j, s) + S_j\delta$. By combining these two inequalities we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_j, s) + (S_j + 1)\delta$. The desired inequality now follows because $S_{j+1} = S_j + 1$ and because $d_G(x_j, s) \leq d_G(x_{j+1}, s)$ (on the shortest path π, x_j is closer to s than x_{j+1}).

Now consider the case that the edge (x_{j+1}, x_j) is not one of the edges that have been inserted since the beginning of the current phase. In that case the edge (x_{j+1}, x_j) in contained in the graph G_0 and thus $d_{G_0}(x_{j+1}, s) \leq d_{G_0}(x_j, s) + 1$. By the induction hypothesis we have $d_{G_0}(x_j, s) \leq d_G(x_j, s) + S_j\delta$. By combining these two inequalities we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_j, s) + 1 + S_j\delta$. Since x_{j+1} and x_j are neighbours on the

shortest path π in G we have $d_G(x_{j+1}, s) = d_G(x_j, s) + 1$. Therefore we get $d_{G_0}(x_{j+1}, s) \leq d_G(x_{j+1}, s) + S_j\delta$. Since $S_{j+1} = S_j$, the desired inequality follows. □

Remark 5.3.3. In the proof of Lemma 5.3.1 we need the property that at most κ edges on the shortest path to the root have been inserted since the beginning of the current phase. If we allow inserting $\kappa/2$ nodes (together with their set of incident edges) we will see at most κ inserted edges on the shortest path to the root as each node appears at most once on this path and contributes at most 2 incident edges. Thus, we can easily modify our algorithms to deal with node insertions with the same approximation guarantee and asymptotic running time.

If we insert an edge (u, v) such that the inequality $d_{G_0}(u, s) \leq d_{G_0}(v, s) + \delta$ does not hold, we cannot guarantee our bound on the additive error anymore. Nevertheless the algorithm makes progress in some sense: After the insertion, u has an edge to v whose initial distance to s was significantly smaller than the one from u to s. This implies that the distance from u to s has decreased by at least δ since the beginning of the current phase. Thus testing whether $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$ is a fast way of testing whether $d_{G_0}(u, s) \geq d_G(u, s) + \delta$, i.e., whether the distance between u and s has decreased so much that a rebuild is necessary.

Lemma 5.3.4. *If an edge (u, v) is inserted such that $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$, then $d_{G_0}(u, s) \geq d_G(u, s) + \delta$.*

Proof. We have inserted an edge (u, v) such that $d_{G_0}(u, s) > d_{G_0}(v, s) + \delta$ (which is equivalent to $d_{G_0}(v, s) \leq d_{G_0}(u, s) - \delta - 1$). In the current graph G, we already have inserted the edge (u, v) and therefore $d_G(u, s) \leq d_G(v, s) + 1$. Since G is the result of inserting edges into G_0, distances in G are not longer than in G_0, and in particular $d_G(v, s) \leq d_{G_0}(v, s)$. Therefore we arrive at the following chain of inequalities:

$$d_G(u, s) \leq d_G(v, s) + 1 \leq d_{G_0}(v, s) + 1 \leq d_{G_0}(u, s) - \delta - 1 + 1 = d_{G_0}(u, s) - \delta$$

Thus, we get $d_{G_0}(u, s) \geq d_G(u, s) + \delta$. □

Since we consider undirected, unweighted graphs, a large decrease in distance for one node also implies a large decrease in distance for many other nodes.

Lemma 5.3.5. *Let $H = (V, E)$ and $H' = (V, E')$ be unweighted, undirected graphs such that H is connected and $E \subseteq E'$. If there is a node $y \in V$ such that $d_H(y, s) \geq d_{H'}(y, s) + \delta$, then $\sum_{x \in V} d_H(x, s) \geq \sum_{x \in V'} d_{H'}(x, s) + \Omega(\delta^2)$.*

Proof. Let π denote the shortest path from y to s of length $d_H(y, s)$ in H. We first bound the distance change for single nodes.

Claim 5.3.6. *For every node x on π we have $d_H(x, s) \geq d_{H'}(x, s) + \delta - d_H(x, y) - d_{H'}(x, y)$.*

Proof of Claim. By the triangle inequality we have $d_{H'}(x,s) \le d_{H'}(x,y) + d_{H'}(y,s)$, which is equivalent to $d_{H'}(y,s) \ge d_{H'}(x,s) - d_{H'}(x,y)$. By this inequality and the fact that x lies on the shortest path from y to s we have

$$d_H(y,x) + d_H(x,s) = d_H(y,s) \ge d_{H'}(y,s) + \delta \ge d_{H'}(x,s) - d_{H'}(x,y) + \delta\,.$$

Since $d_H(y,x) = d_H(x,y)$ the claimed inequality follows. □

From the claim and the fact that $d_{H'}(x,y) \le d_H(x,y)$ we conclude that

$$\begin{aligned}
\sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_H(x,s) &\ge \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} (d_{H'}(x,s) + \delta - 2d_H(x,y)) \\
&= \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_{H'}(x,s) + \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} (\delta - 2d_H(x,y)) \\
&\ge \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_{H'}(x,s) + \sum_{j=1}^{\lfloor\delta/2\rfloor} (\delta - 2j) \\
&= \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_{H'}(x,s) + \delta(\lfloor\delta/2\rfloor) - \lfloor\delta/2\rfloor(\lfloor\delta/2\rfloor + 1) \\
&= \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_{H'}(x,s) + \Omega(\delta^2)
\end{aligned}$$

Finally, we get:

$$\begin{aligned}
\sum_{x\in V} d_H(x,s) &= \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_H(x,s) + \sum_{\substack{x\notin\pi \text{ or} \\ d_H(x,y)\ge\delta/2}} \underbrace{d_H(x,s)}_{\ge d_{H'}(x,s)} \\
&\ge \sum_{\substack{x\in\pi \\ d_H(x,y)<\delta/2}} d_{H'}(x,s) + \Omega(\delta^2) + \sum_{\substack{x\notin\pi \text{ or} \\ d_H(x,y)\ge\delta/2}} d_{H'}(x,s) \\
&= \sum_{x\in V} d_{H'}(x,s) + \Omega(\delta^2)
\end{aligned}$$

□

The quadratic distance decrease is the key observation for the efficiency of our algorithm as it limits the number of times a new phase starts, which is the expensive part of our algorithm.

Lemma 5.3.7 (Running Time)**.** *For every $\kappa \ge 1$ and $\delta \ge 1$, the total update time of Algorithm 5.1 is $O(T_{\text{BFS}}(X) \cdot (q/\kappa + nX/\delta^2 + 1) + q)$, where $T_{\text{BFS}}(X)$ is the time needed for computing a BFS tree up to depth X.*

Proof. Besides the constant time per insertion we have to compute a BFS tree of depth X at the beginning of every phase. The first cause for starting a new phase is that the number of edge deletions in a phase reaches κ, which can happen at most q/κ times. The second cause for starting a new phase is that we insert an edge (u,v) such that

$d_{G_0}(u,s) > d_{G_0}(v,s) + \delta$. By Lemmas 5.3.4 and 5.3.5 this implies that the sum of the distances of all nodes to s has increased by at least $\Omega(\delta^2)$ since the beginning of the current phase. There are at most n nodes of distance at most X to s which means that the sum of the distances is at most nX. Therefore such a decrease can occur at most $O(nX/\delta^2)$ times. The overall running time thus is $O(T_{\text{BFS}}(X) \cdot (q/\kappa + nX/\delta^2 + 1) + q)$. The 1-term is just a technical necessity as the BFS tree has to be computed at least once. □

The algorithm above provides an additive approximation. In the following we turn this into a multiplicative approximation for a fixed distance range. Using a multi-layer approach we enhance this to a multiplicative approximation for the full distance range in Section 5.3.2 (sequential model) and in Section 5.3.3 (distributed model).

Lemma 5.3.8 (Multiplicative Approximation)**.** *Let $0 < \epsilon \leq 1$, $X \leq n$, and set $\gamma = \epsilon/4$. If $\gamma^2 qX \geq n$ and $\gamma nX^2 \geq q$, then by setting $\kappa = q^{1/3}X^{1/3}\gamma^{2/3}/n^{1/3}$ and $\delta = n^{1/3}X^{2/3}\gamma^{1/3}/q^{1/3}$, Algorithm 5.1 has a total update time of*

$$O\left(T_{\text{BFS}}(X) \cdot \frac{q^{2/3}n^{1/3}}{\epsilon^{2/3}X^{1/3}} + q\right),$$

where $T_{\text{BFS}}(X)$ is the time needed for computing a BFS tree up to depth X. Furthermore, it provides the following approximation guarantee: $d_{G_0} \geq d_G(x,s)$ for every node x and $d_{G_0}(x,s) \leq (1+\epsilon)d_G(x,s)$ for every node x such that $X/2 \leq d_{G_0}(x,s) \leq X$.

Proof. To simplify the notation a bit we define $A = \kappa\delta$, which gives $A = \gamma X$.

$$O\left(T_{\text{BFS}}(X) \cdot \left(\frac{q}{\kappa} + \frac{nX}{\delta^2} + 1\right) + q\right).$$

It is easy to check that by our choices of κ and δ the two terms appearing in the running time are balanced and we get

$$\frac{q}{\kappa} = \frac{nX}{\delta^2} = \frac{q^{2/3}n^{1/3}}{\gamma^{2/3}X^{1/3}} = O\left(\frac{q^{2/3}n^{1/3}}{\epsilon^{2/3}X^{1/3}}\right).$$

Furthermore the inequalities $\gamma^2 qX \geq n$ and $\gamma nX^2 \geq q$ ensure that $\kappa \geq 1$ and $\delta \geq 1$.

We now argue that the approximation guarantee holds. By Lemma 5.3.1, we already know that

$$d_G(x,s) \leq d_{G_0}(x,s) \leq d_G(x,s) + A$$

for every node x such that $d_{G_0}(x,s) \leq X$. We now show that our choices of κ and δ guarantee that $A \leq \epsilon d_G(x,s)$, for every node x such that $d_{G_0}(x,s) \geq X/2$, which immediately gives the desired inequality.

Assume that $d_{G_0}(x,s) \leq d_G(x,s) + A$ and that $d_{G_0}(x,s) \geq X/2$. We first show that

$$\gamma \leq \frac{1}{2(1+\frac{1}{\epsilon})}.$$

Since $\epsilon \leq 1$ we have $2(\epsilon + 1) \leq 4$. It follows that

$$\frac{1}{2(1 + \frac{1}{\epsilon})} \geq \frac{\epsilon}{4} = \gamma .$$

Therefore we get the following chain of inequalities:

$$\left(1 + \frac{1}{\epsilon}\right) A = \left(1 + \frac{1}{\epsilon}\right) \gamma X \leq \frac{\left(1 + \frac{1}{\epsilon}\right) X}{2(1 + \frac{1}{\epsilon})} = \frac{X}{2} \leq d_{G_0}(x, s) .$$

We now subtract A from both sides and get

$$\frac{A}{\epsilon} \leq d_{G_0}(x, s) - A .$$

Since $d_{G_0}(x, s) - A \leq d_G(x, s)$ by assumption, we finally get $A \leq \epsilon d_G(x, s)$. □

5.3.2 Sequential model

It is straightforward to use the abstract framework of Section 5.3.1 in the sequential model. First of all, note that in the sequential model computing a BFS tree takes time $O(m)$, regardless of the depth. We run $O(\log n)$ "parallel" instances of Algorithm 5.1, where each instance provides a $(1 + \epsilon)$-approximation for nodes in some distance range from $X/2$ to X. However, when X is small enough, then instead of maintaining the approximate distance with our own algorithm it is more efficient to maintain the exact distance using the algorithm of Even and Shiloach [49].

Theorem 5.3.9. *In the sequential model, there is an incremental $(1 + \epsilon)$-approximate SSSP algorithm for inserting up to q edges with total update time $O(mn^{1/4}\sqrt{q}\log n/\sqrt{\epsilon})$, where m is the number of edges in the final graph. It answers distance and path queries in optimal worst-case time.*

Proof. If $q \leq 8n^{1/2}/\epsilon$, we recompute a BFS tree from scratch after every insertion. This takes time $O(mq) = O(mq^{1/2}q^{1/2}) = O(mn^{1/4}q^{1/2}/\epsilon^{1/2})$.

If $q > 8n^{1/2}/\epsilon$, the algorithm is as follows. Let X^* be the smallest power of 2 greater than or equal to $2n^{1/4}q^{1/2}/\epsilon^{1/2}$ (i.e., $X^* = 2^{\lceil \log (2n^{1/4}q^{1/2}/\epsilon^{1/2}) \rceil}$). First of all, we maintain an Even-Shiloach tree up to depth X^*, which takes time $O(mX^*) = O(mn^{1/4}q^{1/2}/\epsilon^{1/2})$ by Theorem 5.1.2. Additionally, we run $O(\log n)$ instances of Algorithm 5.1, one for each $\log X^* \leq i \leq \lceil \log n \rceil$. For the i-th instance we set the parameter X to $X_i = 2^i$ and κ and δ as in Lemma 5.3.8. Every time we start a new phase for instance i, we also start a new phase for every instance j such that $j \leq i$. This guarantees that if a node leaves the range $[X_i/2, X_i]$ it will immediately be covered by a lower range. Since the graph is connected we now have the following property: for every node v with distance more than X^* to s there is an index i such that at the beginning of the current phase of instance i the distance from v to s was between $X_i/2$ and X_i. By Lemma 5.3.8 this previous distance is a $(1 + \epsilon)$-approximation of the current distance.

The cost of starting a new phase for every instance $j \leq i$ is $O(m \log n)$ since we have to construct a BFS tree up to depth X_j for all $j \leq i$. By Lemma 5.3.8 the running time of the i-th instance of Algorithm 5.1 therefore is $O(mq^{2/3}n^{1/3} \log n/(\epsilon^{2/3}X_i^{1/3}))$, which over all instances gives a running time of

$$O\left(\sum_{\log X^* \leq i \leq \lceil \log n \rceil} \frac{mq^{2/3}n^{1/3} \log n}{\epsilon^{2/3}X_i^{1/3}}\right) = O\left(\frac{mq^{2/3}n^{1/3} \log n}{\epsilon^{2/3}X^{*1/3}}\right)$$
$$= O\left(\frac{mn^{1/4}q^{1/2} \log n}{\epsilon^{2/3}}\right).$$

Note that for each instance i of Lemma 5.3.8 only applies if $\gamma^2 qX_i \geq n$ and $\gamma n X_i^2 \geq q$. These two inequalities hold because q and X^* are large enough:

$$\gamma^2 qX_i = \epsilon^2 qX_i/16 \geq \epsilon^2 qX^*/16 \geq \epsilon^{3/2}q^{3/2}n^{1/4}/8 \geq \epsilon^{3/2}n^{3/4}n^{1/4}/\epsilon^{3/2} = n$$
$$\gamma n X_i^2 = \epsilon n X_i^2/4 \geq \epsilon n (X^*)^2/4 \geq 4\epsilon n^{3/2}q/(4\epsilon) = n^{3/2}q \geq q$$

Finally, we argue that the number q of insertions does not have to be known beforehand. We use a doubling approach for guessing the value of q where the i-th guess is $q_i = 2^i$. When the number of insertions exceeds our guess q_i, we simply restart the algorithm and use the guess $q_{i+1} = 2q_i$ from now on. The total running time for this approach is $O(\sum_{i=0}^{\lceil \log q \rceil} mn^{1/4}q_i^{1/2} \log n/\epsilon^{1/2})$ which is $O(mn^{1/4}q^{1/2} \log n/\epsilon^{1/2})$. □

5.3.3 Distributed Model

In the distributed model we use the same multi-layer approach as in the sequential model. However, we have to consider some additional details for implementing the algorithm because not all information is globally available to every node in the distributed model. Computing a BFS tree up to depth X takes time $T_{\text{BFS}} = O(X)$ in the distributed model. In the running time analysis of Lemma 5.3.7 we thus charge time $O(X)$ to every phase and constant time to every insertion. We now argue that this is enough to implement the algorithm in the distributed model.

After the insertion of an edge (u, v) the nodes u and v have to compare their initial distances $d_{G_0}(u, s)$ and $d_{G_0}(v, s)$. They can exchange these numbers with a constant number of messages which we account for by charging constant time to every insertion.

The root node s has to coordinate the phases and thus needs to gather some special information. The first cause for starting a new phase is when the level of some node decreases by at least δ. If a node detects a level decrease by at least δ, it has to inform the root s about the decrease so that s can initiate the beginning of the next phase. The tree maintained by our algorithm, which has depth at most X, can be used to send this message. Therefore the total time needed for sending this message is $O(X)$, which we charge to the current phase. Note that, similar to recomputing the BFS tree, this happens in a recovery stage during which no new edges are inserted.

The second cause for starting a new phase is that the number of edge insertions since the beginning of the current phase exceeds κ. Therefore it is necessary that the root s knows the number of edges that have been inserted. After the insertion of an edge (u, v) the node v sends a message to the root to inform it about the edge insertion. Again, the tree maintained by our algorithm, which has depth at most X, can be used to send these messages. During each recovery stage we spend time $O(X/\kappa)$ to ensure that every insertion message that has not arrived at the root yet decreases its level in the tree by at least X/κ. We can avoid congestion by aggregating insertion messages because it only matters how many edges have been inserted. Accumulated over 2κ insertions the total time for sending the insertion messages is $O(\kappa X/\kappa) = O(X)$, which we charge to the current phase. In this way, the first insertion message arrives at the root after κ recovery stages and after κ insertions the first $\kappa/2$ messages have arrived. Thus, to get the same approximation guarantee and the same asymptotic running time as in Section 5.3.3 we slightly modify the algorithm to start a new phase if the root has received $\kappa/2$ insertion messages.

Theorem 5.3.10. *In the distributed model, there is an incremental algorithm for maintaining a $(1+\epsilon)$-approximate BFS tree under up to q insertions with total update time $O(q^{2/3}n^{1/3}D^{2/3}/\epsilon^{2/3})$, where D is the dynamic diameter.*

Proof. Our algorithm consists of $O(\log D)$ layers. For each $0 \le i \le \lceil \log D \rceil$ we set $X_i = 2^i$ and do the following: (1) If $q \le 16n/(\epsilon^2 X_i)$, we recompute a BFS tree up to depth X_i from scratch after every insertion. (2) If $q > 16n/(\epsilon^2 X_i)$ and $X_i \le 4\sqrt{q}/\sqrt{\epsilon n}$, we maintain an Even-Shiloach tree up to depth X_i. (3) If $q > 16n/(\epsilon^2 X_i)$ and $X_i > 4\sqrt{q}/\sqrt{\epsilon n}$ we run an instance of Algorithm 5.1 with parameters $X_i = 2^i$ and κ_i and δ_i as in Lemma 5.3.8. We use the following slight modification of Algorithm 5.1: every time a new phase starts for instance i, we re-initialize all instances j of Algorithm 5.1 such that $j \le i$ by computing a BFS tree up to depth X_j. Note that if D is not known in advance, our algorithm can simply increase the number of layers until the BFS tree computed at the initialization of the current layer contains all nodes of the graph.

We first argue that this algorithm provides a $(1+\epsilon)$-approximation. The algorithm maintains the exact distances for all nodes that are at distance at most $16n/(\epsilon^2 q)$ or $4\sqrt{q}/\sqrt{\epsilon n}$ from the root as in these cases the distances are obtained by recomputing the BFS tree from scratch or by the Even-Shiloach tree. For all other nodes we have to argue that our multi-layer version of Algorithm 5.1 provides a $(1+\epsilon)$-approximation. Note that for each instance i the result of Lemma 5.3.8 only applies if $\gamma^2 q X_i \ge n$ and $\gamma n X_i^2 \ge q$. These two inequalities hold because q and X_i are large enough:

$$\gamma^2 q X_i = \epsilon^2 q X_i/16 \ge \epsilon^2 (16n/(\epsilon^2 X_i)) X_i/16 = n$$
$$\gamma n X_i^2 = \epsilon n X_i^2/4 \ge \epsilon n (4\sqrt{q}/\sqrt{\epsilon n})^2/4 = q\,.$$

In each instance i, the approximation guarantee of Lemma 5.3.8 holds for all nodes whose distance to the root was between $X_i/2$ and X_i since the last initialization of instance i. Every time we re-initialize instance i, some nodes that before were in the range $[X_i/2, X_i]$ might now have a smaller distance and will thus not be "covered" by

instance i anymore. By re-initializing all instances $j \leq i$ as well we guarantee that such nodes will immediately be "covered" by some other instance of the algorithm (or by the exact BFS tree we maintain for small depths). Since the graph is connected we thus have the following property: for every node v with distance more than $4\sqrt{q}/\sqrt{\epsilon n}$ to the root there is an index i such that at the beginning of the current phase of instance i the distance from v to the root was between $X_i/2$ and X_i. By Lemma 5.3.8 this previous distance is a $(1+\epsilon)$-approximation of the current distance.

We will now bound the running time. We will argue that the running time in every layer i is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$. If the number of insertions is at most $q \leq 16n/(\epsilon^2 X_i)$, then computing a BFS tree from scratch up to depth X_i after very insertion takes time $O(qX_i)$ in total, which we can bound as follows:

$$qX_i = q^{2/3}q^{1/3}X_i = \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}\,.$$

By Theorem 5.1.2 maintaining an Even-Shiloach tree up to depth $X_i \leq 4\sqrt{q}/\sqrt{\epsilon n}$ takes time $O(nX_i) = O(\sqrt{qn}/\sqrt{\epsilon})$. Since we only do this in the case $q > 16n/(\epsilon^2 X_i)$, we can use the inequality

$$n < \frac{\epsilon^2 qX_i}{16} \leq \frac{qX_i^4}{\epsilon}$$

to obtain

$$nX_i = \frac{\sqrt{qn}}{\sqrt{\epsilon}} = \frac{n^{1/3}n^{1/6}\sqrt{q}}{\sqrt{\epsilon}} \leq \frac{n^{1/3}q^{1/6}X_i^{4/6}\sqrt{q}}{\sqrt{\epsilon}\cdot\epsilon^{1/6}} = \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}\,.$$

Finally we bound the running time of our slight modification of Algorithm 5.1 in layer i. Every time we start a new phase in layer i, we re-initialize the instances of Algorithm 5.1 in all layers $j \leq i$. The re-initialization takes in each layer j takes time $O(X_j)$ as we have to compute a BFS tree up to depth X_j. Thus, the cost of starting a new phase in layer i is proportional to

$$\sum_{0\leq j\leq i} X_j = \sum_{0\leq j\leq i} 2^j \leq 2^{i+1} = 2X_i$$

which asymptotically is the same as only the time needed for computing a BFS tree up to depth X_i. Thus, by Lemma 5.3.8 the running time of instance i of Algorithm 5.1 is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3} + q)$. Since $q \leq \epsilon n X_i^2/4$ as argued above we have $q \leq nX_i^2$ and thus

$$\frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}} \geq q^{2/3}n^{1/3}X_i^{2/3} = q^{2/3}(nX_i^2)^{1/3} \geq q^{2/3}q^{1/3} = q\,.$$

It follows that the running time of instance i is $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$ and the total running time over all layers is

$$\begin{aligned} O\left(\sum_{0\leq i\leq\lceil\log D\rceil} \frac{q^{2/3}n^{1/3}X_i^{2/3}}{\epsilon^{2/3}}\right) &= O\left(\sum_{0\leq i\leq\lceil\log D\rceil} \frac{q^{2/3}n^{1/3}(2^i)^{2/3}}{\epsilon^{2/3}}\right) \\ &= O\left(\frac{q^{2/3}n^{1/3}D^{2/3}}{\epsilon^{2/3}}\right)\,. \end{aligned}$$

By using a doubling approach for guessing the value of q we can run the algorithm with the same asymptotic running time without knowing the number of insertions beforehand. □

5.3.4 Removing the Connectedness Assumption

The algorithm above assumes that the graph is connected. We now explain how to adapt the algorithm to handle graphs where this is not the case.

Note that an insertion might connect one or several nodes to the root node. For each newly connected node, every path to the root goes through an edge that has just been inserted. In such a situation we extend the tree maintained by the Algorithm 5.1 by performing a breadth-first search among the newly connected nodes. Using this modified tree, the argument of Lemma 5.3.1 to prove the additive approximation guarantee still goes through. Note that each node can become connected to the root node at most once. Thus, we can amortize the cost of the breadth-first searches performed to extend the tree over all insertions.

This results in the following modification of the running time of Lemma 5.3.7: In the sequential model we have an additional cost of $O(m)$ as each edge has to be explored at most once in one of the breadth-first searches. In the distributed model we have an additional cost of $O(n)$ as every node is explored at most once in one of the breadth-first searches. The total update time of the $(1+\epsilon)$-approximation in the sequential model (Theorem 5.3.9) clearly stays unaffected from this modification as we anyway have to consider the cost of $O(m)$ for computing a BFS tree. In the distributed model the argument is as follows. In the proof of Theorem 5.3.10 we bound the running time of each instance i of Algorithm 5.1 by $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$. Since q and X_i satisfy the inequality $q > 16n/(\epsilon^2 X_i) \geq n/X_i$ we have $q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3} \geq q^{2/3}n^{1/3}X_i^{2/3} \geq n$. Thus the additional $O(n)$ is already dominated by $O(q^{2/3}n^{1/3}X_i^{2/3}/\epsilon^{2/3})$ and the total update time stays the same as stated in Theorem 5.3.10. Note that if the number of nodes n is not known in advance because of the graph not being connected we can use a doubling approach to guess the right range of n.

5.4 Decremental Algorithm

In the decremental setting we use an algorithm of the same flavor as in the incremental setting (see Algorithm 5.2). However, the update procedure is more complicated because it is not obvious how to repair the tree after a deletion. Our solution exploits the fact that in the distributed model it is relatively cheap to examine the local neighborhood of a node. As in the incremental setting, the algorithm has the parameters κ, δ, and X.

The Procedure RepairTree of Algorithm 5.2 either computes a (weighted) tree T that approximates the true distances with an additive error of $\kappa\delta$, or it reports a distance increase by at least δ since the beginning of the current phase. Let T_0 denote the BFS tree computed at the beginning of the current phase, let F_0 be the forest

Algorithm 5.2: Decremental algorithm

```
// At any time, T_0 is the BFS tree computed at the beginning of
   the current phase, F_0 is the forest resulting from removing
   all deleted edges from T_0 and T is the current approximate
   BFS tree
1  Procedure Delete(u, v)
2      k ← k + 1
3      if k = κ then
4          Initialize()
5      else
6          Remove edge (u, v) from F_0
7          RepairTree()
8          if RepairTree() reports distance increase by at least δ then Initialize()
9  Procedure Initialize()                                  // Start new phase
10     k ← 0
11     Compute BFS tree T_0 of depth X rooted at s.
12     Compute current distances d_{G_0}(·, s)
13     T ← T_0
14     F_0 ← T_0
15 Procedure RepairTree()
16     F ← F_0
17     U ← {u ∈ V | u has no outgoing edge in F and u ≠ s}
18     foreach u ∈ U do                                    // Search process
19         Perform breadth-first search from u up to depth δ and try to find a node v
           such that (1) d_{G_0}(v, s) < d_{G_0}(u, s) and (2) d_G(u, v) ≤ δ.
20         if such a node v could be found then
21             Add edge (u, v) of weight d^w_F(u, v) = d_G(u, v) to F
22         else
23             return "distance increase by at least δ"
24     T ← F
```

resulting from removing those edges from T_0 that have already been deleted in the current phase, and the let U be the set of nodes (except for s) that have no parent in F_0. After every deletion, the Procedure RepairTree tries to construct a tree T by starting with the forest F_0. Every node $u \in U$ tries to find a "good" node v to reconnect to and if successful will use v as its new parent with a weighted edge (u, v) (whose weight corresponds to the current distance between u and v). Algorithm 5.2 imposes two conditions (Line 19) on a "good" node v. Condition (1) avoids that the reconnection introduces any cycles and Condition (2) guarantees that the error introduced by each reconnection is at most δ and that a suitable node v can be found at distance at most δ to u. As δ is relatively small, this is the key to efficiently finding such a node. In the following, we denote the distance between two nodes x and y in a graph F with weighted edges by $d^w_F(x, y)$. Note that here we have formulated

the algorithm in a way such that the Procedure RepairTree always starts with a forest F_0 that is the result of removing all edges from T_0 that have been deleted so far in the current phase, regardless of trees previously computed by the Procedure RepairTree.

5.4.1 Analysis of Procedure for Repairing the Tree

In the following we first analyze only the Procedure RepairTree. Its guarantees can be summarized as follows.

Lemma 5.4.1. *The Procedure* RepairTree *of Algorithm 5.2 either reports "distance increase" and guarantees that there is a node x with $d_{G_0}(x,s) \leq X$ such that*

$$d_G(x,s) \geq d_{G_0}(x,s) + \delta\,,$$

or it returns a tree T such that for every node x with $d_{G_0}(x,s) \leq X$ we have

$$d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^{\mathrm{w}}(x,s) \leq d_{G_0}(x,s) + \kappa\delta\,.$$

It runs in time $O(\kappa\delta)$ after every deletion.

We first observe that the graph returned by the Procedure RepairTree is actually a tree. The input of the procedure is the forest F_0 obtained from removing some edges from the BFS tree T_0. In this forest we have $d_{G_0}(v,s) = d_{G_0}(u,s) - 1$ for every child u and parent v. In the Procedure RepairTree, we add, for every node u that is missing a parent, an edge to a parent v such that $d_{G_0}(v,s) < d_{G_0}(u,s)$. Thus, the decreasing label $d_{G_0}(v,s)$ for every node v guarantees that T is a tree.

Lemma 5.4.2. *The graph T computed by the Procedure* RepairTree *is a tree.*

We will show next that the Procedure RepairTree is either successful, i.e., every node in U finds a new parent, or the algorithm makes progress because there is some node whose distance to the root has increased significantly.

Lemma 5.4.3. *For every node $u \in U$, if $d_G(u,s) < d_{G_0}(u,s) + \delta$, then there is a node $v \in V$ such that*

(1) $d_{G_0}(v,s) < d_{G_0}(u,s)$ and

(2) $d_G(u,v) \leq \delta$.

Proof. If $d_G(u,s) \leq \delta$, then set $v = s$. As $d_{G_0}(s,s) = 0$ and $u \neq s$, this satisfies both conditions.

If $d_G(u,s) > \delta$, then consider the shortest path from u to s and define v as the node that is at distance δ from u on this path, i.e., such that $d_G(v,s) = d_G(u,s) - \delta$. We then have

$$d_{G_0}(v,s) \leq d_G(v,s) = d_G(u,s) - \delta < d_{G_0}(u,s) + \delta - \delta = d_{G_0}(u,s)\,. \qquad \square$$

Note that in the proof above we know exactly which node v we can pick for every node $u \in U$. In the algorithm however the node u does not know its shortest path to s in the current graph and thus it would be expensive to specifically search for the node v on the shortest path from u to s defined above. However, we know that v is contained in the local search performed by u. Therefore u either finds v or some other node that fulfills Conditions (1) and (2).

We now show that every reconnection made by the Procedure RepairTree adds an additive error of δ, which sums up to $\kappa\delta$ for at most κ reconnections (one per previous edge deletion).

Lemma 5.4.4. *For the tree T computed by the Procedure* RepairTree *and every node x such that $d_{G_0}(x, s) \leq X$ we have*

$$d_G(x, s) \leq d_T^{\mathrm{w}}(x, s) \leq d_{G_0}(x, s) + \kappa\delta\,.$$

Proof. We call the weighted edges inserted by the Procedure RepairTree *artificial edges*. In the tree T there are two types of edges: those that were already present in the BFS tree T_0 from the beginning of the current phase and artificial edges added in the Procedure RepairTree.

First, we prove the inequality $d_G(x, s) \leq d_T^{\mathrm{w}}(x, s)$. Consider the unique path from x to s in the tree T consisting of the nodes $x = x_l, x_{l-1}, \dots x_0 = s$. We know that every edge (x_{j+1}, x_j) in T either was part of the initial BFS tree T_0, which means that $d_T^{\mathrm{w}}(x_{j+1}, x_j) = 1 = d_G(x_{j+1}, x_j)$, or was inserted later by the algorithm, which means that $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$. This means that in any case we have $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$ and therefore we get

$$d_T^{\mathrm{w}}(x, s) = \sum_{j=0}^{l-1} d_T^{\mathrm{w}}(x_{j+1}, x_j) = \sum_{j=0}^{l-1} d_G(x_{j+1}, x_j) \geq d_G(x, s)\,.$$

Second, we prove the inequality $d_T^{\mathrm{w}}(x, s) \leq d_{G_0}(x, s) + \kappa\delta$. Consider the shortest path $\pi = x_l, x_{l-1}, \dots x_0$ from x to s in T, where $x_l = x$ and $x_0 = s$. Let S_j (with $0 \leq j \leq l$) denote the number of artificial edges on the subpath $x_j, x_{j-1}, \dots x_0$. For every edge deletion we add at most one artificial edge on π. Therefore we have $S_j \leq \kappa$ for all $0 \leq j \leq l$. Now consider the following claim.

Claim 5.4.5. *For every $0 \leq j \leq l$ we have $d_T^{\mathrm{w}}(x_j, s) \leq d_{G_0}(x_j, s) + S_j\delta$.*

Assuming the truth of the claim, the desired inequality follows straightforwardly since $x_l = x$, $x_0 = s$, and $S_l \leq \kappa$.

In the following we prove the claim by induction on j. In the induction base we have $j = 0$ and thus $x_j = s$ and $S_j = 0$. The inequality then trivially holds due to $d_T^{\mathrm{w}}(s, s) = 0$. We now prove the inductive step from j to $j + 1$. Note that $S_j \leq S_{j+1} \leq S_j + 1$ since the path is exactly one edge longer. Consider first the case that (x_{j+1}, x_j) is an edge from the BFS tree T_0 of the graph G_0. In that case we have $d_T^{\mathrm{w}}(x_{j+1}, x_j) = d_{G_0}(x_{j+1}, x_j) = 1$. Furthermore, since (x_{j+1}, x_j) is an edge in the BFS tree T_0 we know that x_j lies on a shortest path from x_{j+1} to s in G_0. Therefore we

have $d_{G_0}(x_{j+1}, s) = d_{G_0}(x_{j+1}, x_j) + d_{G_0}(x_j, s)$. Together with the induction hypothesis we get:

$$\begin{aligned} d_T^{\mathrm{W}}(x_{j+1}, s) &= \underbrace{d_T^{\mathrm{W}}(x_{j+1}, x_j)}_{=d_{G_0}(x_{j+1}, x_j)} + \underbrace{d_T^{\mathrm{W}}(x_j, s)}_{\leq d_{G_0}(x_j, s) + S_j \cdot \delta \text{ (by IH)}} \\ &\leq \underbrace{d_{G_0}(x_{j+1}, x_j) + d_{G_0}(x_j, s)}_{=d_{G_0}(x_{j+1}, s)} + \underbrace{S_j}_{=S_{j+1}} \cdot \delta \\ &= d_{G_0}(x_{j+1}, s) + S_{j+1} \cdot \delta \,. \end{aligned}$$

The second case is that (x_{j+1}, x_j) is an artificial edge. In that case we have $d_T^{\mathrm{W}}(x_{j+1}, x_j) = d_G(x_{j+1}, x_j)$ and by the algorithm the inequality $d_G(x_{j+1}, x_j) + d_{G_0}(x_j, s) + \leq d_{G_0}(x_{j+1}, s) + \delta$ holds. Note also that $S_{j+1} = S_j + 1$. We therefore get the following:

$$\begin{aligned} d_T^{\mathrm{W}}(x_{j+1}, s) &= \underbrace{d_T^{\mathrm{W}}(x_{j+1}, x_j)}_{=d_G(x_{j+1}, x_j)} + \underbrace{d_T^{\mathrm{W}}(x_j, s)}_{\leq d_{G_0}(x_j, s) + S_j \cdot \delta \text{ (by IH)}} \\ &\leq \underbrace{d_G(x_{j+1}, x_j) + d_{G_0}(x_j, s)}_{\leq d_{G_0}(x_{j+1}, s) + \delta} + S_j \cdot \delta \\ &= d_{G_0}(x_{j+1}, s) + \underbrace{(S_j + 1)}_{=S_{j+1}} \cdot \delta \\ &= d_{G_0}(x_{j+1}, s) + S_{j+1} \cdot \delta \,. \end{aligned}$$

□

Remark 5.4.6. In the proof of Lemma 5.4.4 we need the property that after up to κ edge deletions there are at most κ "artificial" edges on the shortest path to the root in T. This also holds if we allow deleting nodes (together with their set of incident edges). Thus, we can easily modify our algorithm to deal with node deletions with the same approximation guarantee and asymptotic running time.

To finish the proof of Lemma 5.4.1 we analyze the running time of the Procedure RepairTree and clarify some implementation details for the distributed setting. In the search process, every node $u \in U$ tries to find a node v to connect to that fulfills certain properties. We search for such a node v by examining the neighborhood of u in G up to depth δ using breadth-first search, which takes time $O(\delta)$ for a single node. Whenever local searches of nodes in U "overlap" and two messages have to be sent over an edge, we arbitrarily allow to send one of these messages and delay the other one to the next round. As there are at most κ nodes in U, we can simply bound the time needed for all searches by $O(\kappa\delta)$.

Weighted Edges The tree computed by the algorithm contains weighted edges. Such an edge e corresponds to a path π of the same distance in the network. We implement weighted edges by a routing table for every node v that stores the next node on π if a message is sent over v as part of the weighted edge e.

Broadcasting Deletions The nodes that do not have a parent in F_0 before the procedure RepairTree starts do not necessarily know that a new edge deletion has happened. Such a node only has to become active and do the search if there is a change in its neighborhood within distance δ, otherwise it can still use the weighted edge in the tree T that it previously used because the two conditions imposed by the algorithm will still be fulfilled. After the deletion of an edge (x, y), the nodes x and y can inform all nodes at distance δ about this event. This takes time $O(\delta)$ per deletion, which is within our projected running time.

5.4.2 Analysis of Decremental Distributed Algorithm

The Procedure RepairTree provides an additive approximation of the shortest paths and a means for detecting that the distance of some node to s has increased by at least δ since the beginning of the current phase. Using this procedure as a subroutine we can provide a running time analysis for the decremental algorithm that is very similar to the one of the incremental algorithm.

Lemma 5.4.7. *For every $X \geq 1$, $\kappa \geq 1$, and $\delta \geq 1$ the total update time of Algorithm 5.2 is $O(qX/\kappa + nX^2/\delta^2 + q\kappa\delta + n)$ and it provides the following approximation guarantee: If $d_{G_0}(x, s) \leq X$, then*

$$d_{G_0}(x, s) \leq d_G(x, s) \leq d_T^{\mathrm{w}}(x, s) \leq d_{G_0}(x, s) + \kappa\delta\,.$$

Proof. Using the distance increase argument of Lemma 5.3.5, we can bound the number of phases by $O(q/\kappa + nX^2/\delta^2)$. To every phase, we charge a running time of $O(X)$, which is the time needed for computing a BFS tree up to depth X at the beginning of the phase. Additionally we charge a running time of $\kappa\delta$ to every deletion since the Procedure RepairTree, which is called after every deletion, has a running time of $O(\kappa\delta)$ by Lemma 5.4.1.

As in the incremental distributed algorithm we have to enrich the decremental algorithm with a mechanism that allows the root node to coordinate the phases. We explain these implementation details and analyze their effects on the running time in the following.

Reporting Distance Increase When a node v detects a distance increase by more than δ, it tries to inform the root about the distance increase by sending a special message. It sends the message to all nodes at distance at most $2X$ from v in a breadth-first manner, which takes time $O(X)$. If the root is among these nodes, the root initiates a new phase and the cost of $O(X)$ is charged to the new phase. Otherwise, the nodes at distance at most X from v know that their distance to the root is more than X. In that case in particular all nodes in the subtree of v in F_0 have received the message and know that their distance to the root is more than X now. All nodes that are at distance at most X from v do not have to participate in the algorithm anymore. Thus, we can charge the time of $O(X)$ for sending the message to these at least X nodes. This give a one-time charge of $O(1)$ to every node and

adds $O(n)$ to the total update time. A special case is when v becomes disconnected from the root and its new component has size less than X. In that case the time for sending the message to all nodes in the component takes time proportional to the size of the component, which again results in a charge of $O(1)$ to each node.

Counting Deletions The root has to count the number of deletions. Observe that we do not have to count those deletions that result in a distance increase by more than δ because after such an event either a new phase starts or the deletion only affects nodes whose distance to the root has increased to more than X after the deletion. The remaining deletions can be counted by using the tree maintained by the algorithm to send one message per deletion to the root. Note that the level of a node in the tree might increase by at most $\kappa\delta$ with every deletion. Therefore we spend time $O(X/\kappa + \kappa\delta)$ during each recovery stage to ensure that every deletion message that has not arrived at the root yet decreases its level by at least X/κ. In this way, the first deletion message arrives after κ deletions and after 2κ deletions all κ messages have arrived and the root starts a new phase. This process takes time $O(X + \kappa^2\delta)$ for 2κ deletions. We can charge $O(X)$ to the current phase and $O(\kappa\delta)$ to each deletion of the phase. To obtain an additive approximation of exactly $\kappa\delta$, we slightly modify the algorithm to start a new phase if the root has received $\kappa/2$ deletion messages. □

We use a similar approach as in the incremental setting for the $(1+\epsilon)$-approximation. We run i "parallel" instances of the algorithm where each instance covers the distance range from 2^i to 2^{i+1}. By an appropriate choice of the parameters κ and δ for each instance we can guarantee a $(1+\epsilon)$-approximation.

Lemma 5.4.8. *Let $0 < \epsilon \leq 1$ and assume that $\epsilon^5 qX/32 \geq n$ and $nX^{3/2} \geq q$. Then, by setting $\kappa = q^{1/5}X^{1/5}/n^{1/5}$ and $\delta = n^{2/5}X^{3/5}/q^{2/5}$, Algorithm 5.2 runs in time $O(q^{4/5}n^{1/5}X^{4/5})$. Furthermore, it provides the following approximation guarantee: For every node x such that $d_{G_0}(x,s) \leq X$ we have*

$$d_{G_0}(x,s) \leq d_G(x,s) \leq d_T^{\mathrm{w}}(x,s)$$

and for every node x such that $d_G(x,s) \geq X/2$ we additionally have

$$d_T^{\mathrm{w}}(x,s) \leq (1+\epsilon)d_{G_0}(x,s)\,.$$

Proof. Since $\epsilon^5 qX/32 \geq n$ implies $qX \geq n$ we have $\kappa \geq 1$ and since $nX^{3/2} \geq q$ we have $\delta \geq 1$. It is easy to check that by our choices of κ and δ the three terms in the running time of Lemma 5.4.7 are balanced and we get:

$$\frac{q}{\kappa}\cdot X = \frac{nX}{\delta^2}\cdot X = q\kappa\delta = q^{4/5}n^{1/5}X^{4/5}\,.$$

Furthermore, since $qX \geq n$ we have $q^{4/5}n^{1/5}X^{4/5} \geq n^{1/5}(qX)^{4/5} \geq n^{1/5}n^{4/5} = n$ and therefore the running time of the algorithm is $O(q^{4/5}n^{1/5}X^{4/5})$.

We now argue that the approximation guarantee holds. By Lemma 5.4.7, we already know that

$$d_{G_0}(x,s) \le d_G(x,s) \le d_T^{\mathrm{w}}(x,s) \le d_{G_0}(x,s) + \kappa\delta$$

for every node x such that $d_{G_0}(x,s) \le X$. We now show that our choices of κ and δ guarantee that $\kappa\delta \le \epsilon d_{G_0}(x,s)$, for every node x such that $d_{G_0}(x,s) \ge X/2$, which immediately gives the desired inequality. By our assumptions we have $n \le \epsilon^5 qX/32$ and therefore we get

$$\kappa\delta = \frac{q^{1/5}X^{1/5}}{n^{1/5}} \cdot \frac{n^{2/5}X^{3/5}}{q^{2/5}} = \frac{n^{1/5}X^{4/5}}{q^{1/5}} \le \frac{\epsilon q^{1/5}X^{1/5}X^{4/5}}{2q^{1/5}} = \frac{\epsilon X}{2} \le \epsilon d_{G_0}(x,s)\,. \qquad \square$$

Theorem 5.4.9. *In the distributed model, there is a decremental algorithm for maintaining a $(1+\epsilon)$-approximate BFS tree over q deletions with a total update time of $O(q^{4/5}n^{1/5}D^{4/5}/\epsilon)$, where D is the dynamic diameter.*

Proof. Our algorithm consists of $O(\log D)$ layers. For each $0 \le i \le \lceil \log D \rceil$ we set $X_i = 2^i$ and do the following: If $q \le 32n/(\epsilon^5 X_i)$, we recompute a BFS tree up to depth X_i from scratch after every deletion. If $q > 32n/(\epsilon^5 X_i)$ and $X_i \le (q/n)^{2/3}$, we maintain an Even-Shiloach tree up to depth X_i. If $q > 32n/(\epsilon^5 X_i)$ and $X_i > (q/n)^{2/3}$ we run an instance of Algorithm 5.2 with parameters $X_i = 2^i$ and κ_i and δ_i as in Lemma 5.4.8. Note that D might increase over the course of the algorithm due to edge deletions (or might not be known in advance). Therefore, whenever we initialize the algorithm in the layer with the current largest index, we do a full BFS tree computation. If the depth of the BFS tree exceeds X_i, we increase the number of layers accordingly and charge the running time of the BFS tree computation to the layer with new largest index.

We first argue that this algorithm provides a $(1+\epsilon)$-approximation. The algorithm maintains the exact distances for all nodes that are at distance at most $32n/(\epsilon^5 q)$ or $(q/n)^{3/4}$ from the root as in these cases the distances are obtained by recomputing the BFS tree from scratch or by the Even-Shiloach tree. For all other nodes we have to argue that our multi-layer version of Algorithm 5.2 provides a $(1+\epsilon)$-approximation. Note that the approximation guarantee of Lemma 5.3.8 only applies if $\epsilon^5 qX_i/32 \ge n$ and $nX_i^{3/2} \ge q$. These two inequalities hold because q and X_i are large enough:

$$\epsilon^5 qX_i/32 \ge \epsilon^5(32n/(\epsilon^5 X_i))X_i/32 = n$$
$$nX_i^{3/2} \ge n((q/n)^{2/3})^{3/2} = q\,.$$

In each instance i of Algorithm 5.2, the approximation guarantee of Lemma 5.3.8 holds for all nodes whose distance to the root at the beginning of the current phase of instance i was at most X_i and whose current distance to the root is at least $X_i/2$. Whenever an instance i starts a new phase there might be some nodes who before were contained in the tree of instance i, but are not contained in the new tree anymore because their distance to the root has increased to more than X_i. Since $X_i = X_{i+1}/2$

we know that those node will immediately be "covered" by an instance with larger index. Thus, after each recovery stage every node that is connected to the root will be contained in the tree of some instance i such that the preconditions of Lemma 5.3.8 apply and thus the distance to the root in that tree provides a $(1+\epsilon)$-approximation. In particular each node simply has to pick the tree of the smallest index containing it.

We will now bound the running time. We will argue that the running time in every layer i is $O(q^{4/5}n^{1/5}X_i^{4/5}/\epsilon)$. If the number of insertions is at most $q \leq 32n/(\epsilon^5 X_i)$, then computing a BFS tree from scratch up to depth X_i after very insertion takes time $O(qX_i)$ in total, which we can bound as follows:

$$qX_i = q^{4/5}q^{1/5}X_i = \frac{q^{4/5}32^{1/5}n^{1/5}X_i^{4/5}}{\epsilon} = O\left(\frac{q^{4/5}n^{1/5}X_i^{4/5}}{\epsilon}\right).$$

By Theorem 5.1.2 maintaining an Even-Shiloach tree up to depth $X_i \leq (q/n)^{2/3}$ takes time $O(nX_i) = O(q^{2/3}n^{1/3})$. Since we only do this in the case $q > 32n/(\epsilon^5 X_i)$, we can use the inequality

$$n < \frac{\epsilon^5 qX_i}{32} \leq \epsilon^5 qX_i \leq \frac{qX_i^6}{\epsilon^{15/2}}$$

to obtain

$$nX_i = q^{2/3}n^{1/3} = q^{2/3}n^{1/5}n^{2/15} \leq q^{2/3}n^{1/5}\frac{q^{2/15}X_i^{4/5}}{\epsilon} = \frac{q^{4/5}n^{1/5}X_i^{4/5}}{\epsilon}.$$

Finally we use Lemma 5.4.8 to bound the running time of Algorithm 5.1 in layer i by $O(q^{4/5}n^{1/4}X_i^{4/5}/\epsilon)$ as well. Thus, the running time over all layers is

$$\begin{aligned} O\left(\sum_{0\leq i\leq \lceil \log D\rceil} \frac{q^{4/5}n^{1/4}X_i^{4/5}}{\epsilon}\right) &= O\left(\sum_{0\leq i\leq \lceil \log D\rceil} \frac{q^{4/5}n^{1/4}(2^i)^{4/5}}{\epsilon}\right) \\ &= O\left(\frac{q^{4/5}n^{1/4}D^{4/5}}{\epsilon}\right). \end{aligned}$$

By using a doubling approach for guessing the value of q we can run the algorithm with the same asymptotic running time without knowing the number of deletions beforehand. □

5.5 Conclusion and Open Problems

In this chapter, we showed that an approximate breadth-first search spanning tree can be maintained in amortized time per update that is sublinear in the diameter D in partially dynamic distributed networks. Many problems remain open. For example, can we get a similar result for the case of *fully-dynamic* networks? How about *weighted* networks (even partially dynamic ones)? Can we also get a sublinear time bound for the *all-pairs shortest paths* problem? Moreover, in addition to the

sublinear-time complexity achieved in this chapter, it is also interesting to obtain algorithms with small bounds on message complexity and memory.

We believe that the most interesting open problem is whether the sequential algorithm in this chapter can be improved to obtain a deterministic incremental algorithm with near-linear total update time. As noted earlier, techniques from this chapter have led to a *randomized* decremental algorithm with near-linear total update time as presented in Chapter 3 (the same algorithm also works in the incremental setting). Whether this algorithm can be derandomized was left as a major open problem. As the incremental case is usually easier than the decremental case, it is worth obtaining this result in the incremental setting first.

Bibliography

[1] Amir Abboud and Virginia Vassilevska Williams. "Popular conjectures imply strong lower bounds for dynamic problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443.

[2] Ittai Abraham and Shiri Chechik. "Dynamic Decremental Approximate Distance Oracles with $(1 + \epsilon, 2)$ stretch". In: *CoRR* abs/1307.1516 (2013).

[3] Ittai Abraham, Shiri Chechik, and Kunal Talwar. "Fully Dynamic All-Pairs Shortest Paths: Breaking the $O(n)$ Barrier". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. 2014, pp. 1–16.

[4] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. "Applying Static Network Protocols to Dynamic Networks". In: *Symposium on Foundations of Computer Science (FOCS)*. 1987, pp. 358–370.

[5] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael E. Saks. "Local Management of a Global Resource in a Communication Network". In: *Journal of the ACM* 43.1 (1996). Announced at FOCS'87, pp. 1–19.

[6] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. "Fast Estimation of Diameter and Shortest Paths (Without Matrix Multiplication)". In: *SIAM Journal on Computing* 28.4 (1999). Announced at SODA'96, pp. 1167–1181.

[7] Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Maintaining Information in Fully Dynamic Trees with Top Trees". In: *ACM Transactions on Algorithms* 1.2 (2005). Announced at ICALP'97 and SWAT'00, pp. 243–264.

[8] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. "Small Stretch Spanners on Dynamic Graphs". In: *Journal of Graph Algorithms and Applications* 10.2 (2006). Announced at ESA'05, pp. 365–385.

[9] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. "Incremental Algorithms for Minimal Length Paths". In: *Journal of Algorithms* 12.4 (1991). Announced at SODA'90, pp. 615–638.

[10] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. "On-Line Computation of Minimal and Maximal Length Paths". In: *Theoretical Computer Science* 95.2 (1992), pp. 245–261.

[11] Baruch Awerbuch. "Complexity of Network Synchronization". In: *Journal of the ACM* 32.4 (1985). Announced at STOC'84, pp. 804–823.

[12] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. "Near-Linear Time Construction of Sparse Neighborhood Covers". In: *SIAM Journal on Computing* 28.1 (1998). Announced at FOCS'93, pp. 263–277.

[13] Baruch Awerbuch, Israel Cidon, and Shay Kutten. "Optimal Maintenance of a Spanning Tree". In: *Journal of the ACM* 55.4 (2008). Announced at FOCS'90, 18:1–18:45.

[14] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. "Graph Expansion and Communication Costs of Fast Matrix Multiplication". In: *Journal of the ACM* 59.6 (2012). Announced at SPAA'11, 32:1–32:23.

[15] Surender Baswana, Manoj Gupta, and Sandeep Sen. "Fully Dynamic Maximal Matching in $O(\log n)$ Update Time". In: *SIAM Journal on Computing* 44.1 (2015). Announced at FOCS'11, pp. 88–113.

[16] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths". In: *Journal of Algorithms* 62.2 (2007). Announced at STOC'02, pp. 74–92.

[17] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Maintaining All-Pairs Approximate Shortest Paths Under Deletion of Edges". In: *Symposium on Discrete Algorithms (SODA)*. 2003, pp. 394–403.

[18] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. "Fully Dynamic Randomized Algorithms for Graph Spanners". In: *ACM Transactions on Algorithms* 8.4 (2012). Announced at ESA'04, and SODA'08, 35:1–35:51.

[19] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. "On the Power of Randomization in On-Line Algorithms". In: *Algorithmica* 11.1 (1994). Announced at STOC'90, pp. 2–14.

[20] Michael A. Bender, Jeremy T. Fineman, and Seth Gilbert. "A New Approach to Incremental Topological Ordering". In: *Symposium on Discrete Algorithms (SODA)*. 2009, pp. 1108–1115.

[21] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert Endre Tarjan. "A New Approach to Incremental Cycle Detection and Related Problems". In: *CoRR* abs/1112.0784 (2011).

[22] Aaron Bernstein. "Fully Dynamic $(2 + \epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 693–702.

[23] Aaron Bernstein. "Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 725–734.

[24] Aaron Bernstein and Liam Roditty. "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions". In: *Symposium on Discrete Algorithms (SODA)*. 2011, pp. 1355–1365.

[25] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. "Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching". In: *Symposium on Discrete Algorithms (SODA)*. 2015, pp. 785–804.

[26] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998, pp. I–XVIII, 1–414.

[27] Lubos Brim, Jakub Chaloupka, Laurent Doyen, Raffaella Gentilini, and Jean-François Raskin. "Faster algorithms for mean-payoff games". In: *Formal Methods in System Design* 38.2 (2011). Announced at MEMICS'09, and GAMES'09, pp. 97–118.

[28] Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni. "Partially dynamic efficient algorithms for distributed shortest paths". In: *Theoretical Computer Science* 411.7-9 (2010). Announced at ICCTA'07, pp. 1013–1037.

[29] Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, and Alberto Petricola. "Partially Dynamic Algorithms for Distributed Shortest Paths and their Experimental Evaluation". In: *Journal of Computers* 2.9 (2007). Announced at ICCTA'07, pp. 16–26.

[30] Israel Cidon, Inder S. Gopal, Marc A. Kaplan, and Shay Kutten. "A Distributed Control Architecture of High-Speed Networks". In: *IEEE Transactions on Communications* 43.5 (1995). Announced at PODC'90, pp. 1950–1960.

[31] Edith Cohen. "Fast Algorithms for Constructing t-Spanners and Paths with Stretch t". In: *SIAM Journal on Computing* 28.1 (1998). Announced at FOCS'93, pp. 210–236.

[32] Edith Cohen. "Polylog-Time and Near-Linear Work Approximation Scheme for Undirected Shortest Paths". In: *Journal of the ACM* 47.1 (2000). Announced at STOC'94, pp. 132–166.

[33] Edith Cohen and Uri Zwick. "All-Pairs Small-Stretch Paths". In: *Journal of Algorithms* 38.2 (2001). Announced at SODA'97, pp. 335–353.

[34] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. "Distributed Verification and Hardness of Distributed Approximation". In: *SIAM Journal on Computing* 41.5 (2012). Announced at STOC'11, pp. 1235–1265.

[35] Shantanu Das, Beat Gfeller, and Peter Widmayer. "Computing All Best Swaps for Minimum-Stretch Tree Spanners". In: *Journal of Graph Algorithms and Applications* 14.2 (2010). Announced at ISAAC'08, pp. 287–306.

[36] Camil Demetrescu and Giuseppe F. Italiano. "A New Approach to Dynamic All Pairs Shortest Paths". In: *Journal of the ACM* 51.6 (2004). Announced at STOC'03, pp. 968–992.

[37] Camil Demetrescu and Giuseppe F. Italiano. "Fully dynamic all pairs shortest paths with real edge weights". In: *Journal of Computer and System Sciences* 72.5 (2006). Announced at FOCS'01, pp. 813–837.

[38] Camil Demetrescu and Giuseppe F. Italiano. "Improved Bounds and New Trade-Offs for Dynamic All Pairs Shortest Paths". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2002, pp. 633–643.

[39] Camil Demetrescu and Giuseppe F. Italiano. "Trade-offs for Fully Dynamic Transitive Closure on DAGs: Breaking through the $O(n^2)$ Barrier". In: *Journal of the ACM* 52.2 (2005). Announced at FOCS'00, pp. 147–156.

[40] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. "Dynamic Perfect Hashing: Upper and Lower Bounds". In: *SIAM Journal on Computing* 23.4 (1994). Announced at FOCS'88, pp. 738–761.

[41] Dorit Dor, Shay Halperin, and Uri Zwick. "All-Pairs Almost Shortest Paths". In: *SIAM Journal on Computing* 29.5 (2000). Announced at FOCS'96, pp. 1740–1759.

[42] Michael Elkin. "A faster distributed protocol for constructing a minimum spanning tree". In: *Journal of Computer and System Sciences* 72.8 (2006). Announced at SODA'04, pp. 1282–1308.

[43] Michael Elkin. "A Near-Optimal Distributed Fully Dynamic Algorithm for Maintaining Sparse Spanners". In: *Symposium on Principles of Distributed Computing (PODC)*. 2007, pp. 185–194.

[44] Michael Elkin. "Computing Almost Shortest Paths". In: *ACM Transactions on Algorithms* 1.2 (2005). Announced at PODC'01, pp. 283–323.

[45] Michael Elkin. "Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners". In: *ACM Transactions on Algorithms* 7.2 (2011). Announced at ICALP'07, 20:1–20:17.

[46] Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. "Can Quantum Communication Speed Up Distributed Computation?" In: *Symposium on Principles of Distributed Computing (PODC)*. 2014, pp. 166–175.

[47] Michael Elkin and David Peleg. "$(1+\epsilon, \beta)$-Spanner Constructions for General Graphs". In: *SIAM Journal on Computing* 33.3 (2004). Announced at STOC'01, pp. 608–631.

[48] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. "Sparsification—A Technique for Speeding Up Dynamic Graph Algorithms". In: *Journal of the ACM* 44.5 (1997). Announced at FOCS'92, pp. 669–696.

[49] Shimon Even and Yossi Shiloach. "An On-Line Edge-Deletion Problem". In: *Journal of the ACM* 28.1 (1981), pp. 1–4.

[50] Jittat Fakcharoenphol and Satish Rao. "Planar graphs, negative weight edges, shortest paths, and near linear time". In: *Journal of Computer and System Sciences* 72.5 (2006). Announced at FOCS'01, pp. 868–889.

[51] Paola Flocchini, Antonio Mesa Enriques, Linda Pagli, Giuseppe Prencipe, and Nicola Santoro. "Point-of-Failure Shortest-Path Rerouting: Computing the Optimal Swap Edges Distributively". In: *IEICE Transactions on Information and Systems* 89-D.2 (2006), pp. 700–708.

[52] Greg N. Frederickson. "Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications". In: *SIAM Journal on Computing* 14.4 (1985). Announced at STOC'83, pp. 781–798.

[53] Juan A. Garay, Shay Kutten, and David Peleg. "A sublinear time distributed algorithm for minimum-weight spanning trees". In: *SIAM Journal on Computing* 27 (1998). Announced at FOCS'93, pp. 302–316.

[54] Beat Gfeller. "Faster Swap Edge Computation in Minimum Diameter Spanning Trees". In: *Algorithmica* 62.1-2 (2012). Announced at ESA'08, pp. 169–191.

[55] Beat Gfeller, Nicola Santoro, and Peter Widmayer. "A Distributed Algorithm for Finding All Best Swap Edges of a Minimum-Diameter Spanning Tree". In: *IEEE Transactions on Dependable and Secure Computing* 8.1 (2011). Announced at DISC'07, pp. 1–12.

[56] Manoj Gupta and Richard Peng. "Fully Dynamic $(1 + \epsilon)$-Approximate Matchings". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 548–557.

[57] Thomas P. Hayes, Jared Saia, and Amitabh Trehan. "The Forgiving Graph: a distributed data structure for low stretch under adversarial attack". In: *Distributed Computing* 25.4 (2012). Announced at PODC'09, pp. 261–278.

[58] Monika Rauch Henzinger and Valerie King. "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation". In: *Journal of the ACM* 46.4 (1999). Announced at STOC'95, pp. 502–516.

[59] Monika Rauch Henzinger, Philip N. Klein, Satish Rao, and Sairam Subramanian. "Faster Shortest-Path Algorithms for Planar Graphs". In: *Journal of Computer and System Sciences* 55.1 (1997). Announced at STOC'94, pp. 3–23.

[60] Monika Rauch Henzinger and Mikkel Thorup. "Sampling to provide or to bound: With applications to fully dynamic graph algorithms". In: *Random Structures & Algorithms* 11.4 (1997). Announced at ICALP'96, pp. 369–379.

[61] Monika Henzinger and Valerie King. "Fully Dynamic Biconnectivity and Transitive Closure". In: *Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 664–672.

[62] Monika Henzinger and Valerie King. "Maintaining Minimum Spanning Forests in Dynamic Graphs". In: *SIAM Journal on Computing* 31.2 (2001). Announced at ICALP'97, pp. 364–374.

[63] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Dynamic Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072.

[64] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155.

[65] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 538–547.

[66] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *SIAM Journal on Computing* (forthcoming). Announced at FOCS'13.

[67] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Improved Algorithms for Decremental Single-Source Reachability on Directed Graphs". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2015, forthcoming.

[68] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 674–683.

[69] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2013, pp. 607–619.

[70] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. "Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture". In: *Symposium on Theory of Computing (STOC)*. 2015.

[71] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *Journal of the ACM* 48.4 (2001). Announced at STOC'98, pp. 723–760.

[72] Giuseppe F. Italiano. "Amortized Efficiency of a Path Retrieval Data Structure". In: *Theoretical Computer Science* 48.3 (1986), pp. 273–281.

[73] Giuseppe F. Italiano. "Distributed Algorithms for Updating Shortest Paths". In: *International Workshop on Distributed Algorithms on Graphs (WDAG/DISC)*. 1991, pp. 200–211.

[74] Giuseppe F. Italiano. "Finding Paths and Deleting Edges in Directed Acyclic Graphs". In: *Information Processing Letters* 28.1 (1988), pp. 5–11.

[75] Giuseppe F. Italiano and Rajiv Ramaswami. "Maintaining Spanning Trees of Small Diameter". In: *Algorithmica* 22.3 (1998). Announced at ICALP'94, pp. 275–304.

[76] Hiro Ito, Kazuo Iwama, Yasuo Okabe, and Takuya Yoshihiro. "Single backup table schemes for shortest-path routing". In: *Theoretical Computer Science* 333.3 (2005). Announced at SIROCCO'03, pp. 347–353.

[77] Bruce M. Kapron, Valerie King, and Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1131–1142.

[78] Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. "Efficient distributed approximation algorithms via probabilistic tree embeddings". In: *Distributed Computing* 25.3 (2012). Announced at PODC'08, pp. 189–205.

[79] Valerie King. "Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 1999, pp. 81–91.

[80] Valerie King. "Fully Dynamic Transitive Closure". In: *Encyclopedia of Algorithms*. 2008.

[81] Valerie King and Garry Sagert. "A Fully Dynamic Algorithm for Maintaining the Transitive Closure". In: *Journal of Computer and System Sciences* 65.1 (2002). Announced at STOC'00, pp. 150–167.

[82] Valerie King and Mikkel Thorup. "A Space Saving Trick for Directed Dynamic Transitive Closure and Shortest Path Algorithms". In: *International Computing and Combinatorics Conference (COCOON)*. 2001, pp. 268–277.

[83] Liah Kor, Amos Korman, and David Peleg. "Tight Bounds for Distributed Minimum-Weight Spanning Tree Verification". In: *Theory of Computing Systems* 53.2 (2013). Announced at STACS'11, pp. 318–340.

[84] Amos Korman. "Improved Compact Routing Schemes for Dynamic Trees". In: *Symposium on Principles of Distributed Computing (PODC)*. 2008, pp. 185–194.

[85] Amos Korman and Shay Kutten. "Controller and Estimator for Dynamic Networks". In: *Information and Computation* 223 (2013). Announced at PODC'07, pp. 43–66.

[86] Amos Korman and David Peleg. "Dynamic Routing Schemes for Graphs with Low Local Density". In: *ACM Transactions on Algorithms* 4.4 (2008). Announced at ICALP'06, 41:1–41:18.

[87] Danny Krizanc, Flaminia L. Luccio, and Rajeev Raman. "Compact Routing Schemes for Dynamic Ring Networks". In: *Theory of Computing Systems* 37.5 (2004). Announced at IPPS/SPDP'99, pp. 585–607.

[88] Shay Kutten and David Peleg. "Fast Distributed Construction of Small k-Dominating Sets and Applications". In: *Journal of Algorithms* 28.1 (1998). Announced at PODC'95, pp. 40–66.

[89] Shay Kutten and Avner Porat. "Maintenance of a Spanning Tree in Dynamic Networks". In: *International Symposium on Distributed Computing (DISC)*. 1999, pp. 342–355.

[90] Jakub Łącki. "Improved Deterministic Algorithms for Decremental Reachability and Strongly Connected Components". In: *ACM Transactions on Algorithms* 9.3 (2013). Announced at SODA'11, p. 27.

[91] Zvi Lotker, Boaz Patt-Shamir, and David Peleg. "Distributed MST for constant diameter graphs". In: *Distributed Computing* 18.6 (2006). Announced at PODC'01, pp. 453–460.

[92] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. "Minimum-Weight Spanning Tree Construction in $O(\log \log n)$ Communication Rounds". In: *SIAM Journal on Computing* 35.1 (2005). Announced at SPAA'03, pp. 120–131.

[93] P.S. Loubal and Bay Area Transportation Study Commission. *A Network Evaluation Procedure*. Bay Area Transportation Study Commission, 1967.

[94] Nancy A. Lynch. *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 1996.

[95] Aleksander Mądry. "Faster Approximation Schemes for Fractional Multicommodity Flow Problems via Dynamic Graph Algorithms". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 121–130.

[96] Navneet Malpani, Jennifer L. Welch, and Nitin H. Vaidya. "Leader Election Algorithms for Mobile Ad Hoc Networks". In: *Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL-M)*. 2000, pp. 96–103.

[97] John D. Murchland. *The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph*. Tech. rep. LBS-TNT-26. London Business School, Transport Network Theory Unit, 1967.

[98] Danupon Nanongkai. "Distributed Approximation Algorithms for Weighted Shortest Paths". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 565–573.

[99] Enrico Nardelli, Guido Proietti, and Peter Widmayer. "Finding All the Best Swaps of a Minimum Diameter Spanning Tree Under Transient Edge Failures". In: *Journal of Graph Algorithms and Applications* 5.5 (2001). Announced at ESA'98, pp. 39–57.

[100] Enrico Nardelli, Guido Proietti, and Peter Widmayer. "Swapping a Failing Edge of a Single Source Shortest Paths Tree Is Good and Fast". In: *Algorithmica* 35.1 (2003). Announced at COCOON'99, pp. 56–74.

[101] Ofer Neiman and Shay Solomon. "Simple Deterministic Algorithms for Fully Dynamic Maximal Matching". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 745–754.

[102] Krzysztof Onak and Ronitt Rubinfeld. "Maintaining a Large Matching and a Small Vertex Cover". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 457–464.

[103] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004). Announced at ESA'01, pp. 122–144.

[104] Mihai Patrascu. "Towards Polynomial Lower Bounds for Dynamic Problems". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 603–610.

[105] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Philadelphia, PA, USA: SIAM, 2000.

[106] David Peleg and Vitaly Rubinovich. "A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction". In: *SIAM Journal on Computing* 30.5 (2000). Announced at FOCS'99, pp. 1427–1442.

[107] K. V. S. Ramarao and S. Venkatesan. "On Finding and Updating Shortest Paths Distributively". In: *Journal of Algorithms* 13.2 (1992). Announced at Allerton Conference'86, pp. 235–257.

[108] Liam Roditty. "A Faster and Simpler Fully Dynamic Transitive Closure". In: *ACM Transactions on Algorithms* 4.1 (2008). Announced at SODA'03.

[109] Liam Roditty. "Decremental maintenance of strongly connected components". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1143–1150.

[110] Liam Roditty, Mikkel Thorup, and Uri Zwick. "Deterministic Constructions of Approximate Distance Oracles and Spanners". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2005, pp. 261–272.

[111] Liam Roditty and Roei Tov. "Approximating the Girth". In: *ACM Transactions on Algorithms* 9.2 (2013). Announced at SODA'11, 15:1–15:13.

[112] Liam Roditty and Uri Zwick. "A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time". In: *Symposium on Theory of Computing (STOC)*. 2004, pp. 184–191.

[113] Liam Roditty and Uri Zwick. "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs". In: *SIAM Journal on Computing* 41.3 (2012). Announced at FOCS'04, pp. 670–683.

[114] Liam Roditty and Uri Zwick. "Improved Dynamic Reachability Algorithms for Directed Graphs". In: *SIAM Journal on Computing* 37.5 (2008). Announced at FOCS'02, pp. 1455–1471.

[115] Liam Roditty and Uri Zwick. "On Dynamic Shortest Paths Problems". In: *Algorithmica* 61.2 (2011). Announced at ESA'04, pp. 389–401.

[116] Aleksej Di Salvo and Guido Proietti. "Swapping a failing edge of a shortest paths tree by minimizing the average stretch factor". In: *Theoretical Computer Science* 383.1 (2007). Announced at SIROCCO'04, pp. 23–33.

[117] Piotr Sankowski. "Dynamic Transitive Closure via Dynamic Matrix Inverse". In: *Symposium on Foundations of Computer Science (FOCS)*. 2004, pp. 509–517.

[118] Piotr Sankowski. "Faster Dynamic Matchings and Vertex Connectivity". In: *Symposium on Discrete Algorithms (SODA)*. 2007, pp. 118–126.

[119] Piotr Sankowski. "Subquadratic Algorithm for Dynamic Shortest Distances". In: *International Computing and Combinatorics Conference (COCOON)*. 2005, pp. 461–470.

[120] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972). Announced at SWAT'71 (FOCS), pp. 146–160.

[121] Mikkel Thorup. "Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles". In: *Scandinavian Workshop on Algorithm Theory (SWAT)*. 2004, pp. 384–396.

[122] Mikkel Thorup. "Near-optimal fully-dynamic graph connectivity". In: *Symposium on Theory of Computing (STOC)*. 2000, pp. 343–350.

[123] Mikkel Thorup. "Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time". In: *Journal of the ACM* 46.3 (1999). Announced at FOCS'97, pp. 362–394.

[124] Mikkel Thorup. "Worst-Case Update Times for Fully-Dynamic All-Pairs Shortest Paths". In: *Symposium on Theory of Computing (STOC)*. 2005, pp. 112–119.

[125] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *Journal of the ACM* 52.1 (2005). Announced at STOC'01, pp. 74–92.

[126] Mikkel Thorup and Uri Zwick. "Spanners and emulators with sublinear distance errors". In: *Symposium on Discrete Algorithms (SODA)*. 2006, pp. 802–809.

[127] Jeffrey D. Ullman and Mihalis Yannakakis. "High-Probability Parallel Transitive-Closure Algorithms". In: *SIAM Journal on Computing* 20.1 (1991). Announced at SPAA'90, pp. 100–125.

[128] Virginia Vassilevska Williams and Ryan Williams. "Subcubic Equivalences between Path, Matrix and Triangle Problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 645–654.

[129] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. "All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time". In: *Theory of Computing* 5.1 (2009). Announced at STOC'07, pp. 173–189.

[130] Uri Zwick. "All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication". In: *Journal of the ACM* 49.3 (2002). Announced at FOCS'98, pp. 289–317.

Curriculum Vitæ

Education

2011-2015 *PhD studies*, Computer Science, University of Vienna, Austria
Thesis title: *Faster Approximation Algorithms for Partially Dynamic Shortest Paths Problems*
Thesis supervisor: Prof. Monika Henzinger

2008–2011 *Master studies*, Computational Intelligence, Vienna University of Technology, Austria
Thesis title: *Combining Supervaluation and Fuzzy Logic Based Theories of Vagueness*
Thesis supervisor: Ao.Prof. Christian Fermüller

2005–2008 *Bachelor studies*, Computer Science, University of Passau, Germany
Thesis title: *On-line Identification Using Handwritten Passwords*
Thesis supervisor: Prof. Bernhard Sick

Employment

2014 *Intern*, Microsoft Research, Silicon Valley Lab, Mountain View, USA
Mentors: Ittai Abraham and Shiri Chechik

since 2011 *Research assistant*, Theory and Applications of Algorithms group, University of Vienna, Austria

2010–2011 *Student assistant*, Theory and Applications of Algorithms group, University of Vienna, Austria

2008 *Student assistant*, Computationally Intelligent Systems group, University of Passau, Germany

2007 *Student assistant*, Institute for Information Systems and Software Technology, University of Passau, Germany

2005–2008 *Intern during semester breaks*, InfraServ Gendorf, Burgkirchen, Germany

Scholarships

2011 Diploma thesis scholarship, Vienna University of Technology

2008–2011 Max Weber-Program (Bavarian scholarship system)

2008–2010 German National Academic Foundation (Studienstiftung des Deutschen Volkes)