Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs

Monika Henzinger* Sebastian Krinninger* Danupon Nanongkai†

Abstract

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 (JACM 1981); 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 paper, we answer this question affirmatively. We obtain a randomized algorithm which, in a simplified form, achieves an $\widetilde{O}(mn^{0.984})$ expected total update time for SSR and $(1+\epsilon)$-approximate SSSP, where $\widetilde{O}(\cdot)$ hides $\operatorname{poly}\log n$. We also extend our algorithm to achieve roughly the same running time for Strongly Connected Components (SCC), improving the algorithm of Roditty and Zwick (FOCS 2002), and an algorithm that improves the $\widetilde{O}(mn\log W)$-time algorithm of Bernstein (STOC 2013) for approximating SSSP on weighted directed graphs, where the edge weights are integers from 1 to W. All our algorithms have constant query time in the worst case.

*University of Vienna, Faculty of Computer Science, Austria. Supported by the Austrian Science Fund (FWF): P23499-N23 and the University of Vienna (IK I049-N). The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506 and from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532.

†ICERM, Brown University, USA. Work done while at Nanyang Technological University, Singapore 637371, and while supported in part by the following research grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082, and Singapore MOE AcRF Tier 1 grant MOE2012-T1-001-094.

Contents

Part I
Overview

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 the minimum spanning forest can be maintained *very quickly*, i.e., in polylogarithmic amortized time per operation ([9, 11, 21, 12]), 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 [7].

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 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). It also has applications in various other contexts, including computing maximum flow, computing multicommodity flow and deadlock detection in operating systems. Yet, no algorithm with sublinear update time was known for this problem.

The fastest algorithm for this problem was published in 1981 and takes $O(mn)$ total update time [6][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. In 1999 King [13] showed how to extend this data structure 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)[2]. In a recent breakthrough, Bernstein [3] presented in 2013 a $(1+\epsilon)$-approximate single-source shortest path algorithm with total update time $\widetilde{O}(mn \log W)$, where the $\widetilde{O}(\cdot)$ notation hides factors polylogarithmic in n. The situation is similar for decremental strongly connected components: The fastest decremental SCC algorithms take total update time $O(mn)$ ([18, 15, 17]). 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 [14, 15, 18].

Our Results. We improve the previous $O(mn)$-time algorithms for SSR, approximate SSSP, and SCC. We also give an algorithm for *s-t reachability* (stR), where we want to maintain whether node s can reach node t, and *s-t* shortest path (stSP) where we want to maintain the distance from s to t. In particular, we develop several algorithms, each of them is faster than $O(mn)$ for different values of m, as summarized in Table 1. We can combine these algorithms by checking the value of m (compared to n) before the first deletion happens and use an appropriate algorithm to handle deletions and queries. Combining these algorithms gives the following result.

[1] It was actually published for undirected graphs and it was observed by Henzinger and King [8] that it can be easily adapted to work for directed graphs.

[2] The total update time is actually $O(md)$, where d is the maximum distance, which could be $O(nW)$.

Problems	Running Time	$o(mn)$ when	Fastest when
$O(\log n)$-approx. unweighted stSP,	$\widetilde{O}(m^{7/5}n^{2/5})$	$m = o(n^{1.5})$	$n \leq m \leq n^{1.45}$
	$\widetilde{O}(m^{191/207}n^{226/207})$	$m = \omega(n^{1.19})$	$n^{1.45} \leq m \leq n^{1.46}$
	$\widetilde{O}(m^{25/33}n^{4/3})$	$m = \omega(n^{1.38})$	$n^{1.46} \leq m \leq n^{1.62}$
	$\widetilde{O}(m^{1/2}n^{7/4})$	$m = \omega(n^{1.5})$	$n^{1.62} \leq m \leq n^2$
$O(\log n)$-approx. unweighted SSSP, SSR, and SCC	$\widetilde{O}(m^{5/4}n^{5/8})$	$m = o(n^{1.5})$	$n \leq m \leq n^{1.43}$
	$\widetilde{O}(m^{191/207}n^{226/207})$	$m = \omega(n^{1.19})$	$n^{1.43} \leq m \leq n^{1.46}$
	$\widetilde{O}(m^{25/33}n^{4/3})$	$m = \omega(n^{1.38})$	$n^{1.46} \leq m \leq n^{1.62}$
	$\widetilde{O}(m^{1/2}n^{7/4})$	$m = \omega(n^{1.5})$	$n^{1.62} \leq m \leq n^2$
$(1+\epsilon)$-approx. weighted stSP	$\widetilde{O}(m^{7/5}n^{2/5})$	$m = o(n^{1.5})$	$n \leq m \leq n^{1.46}$
	$\widetilde{O}(m^{25/27}n^{206/189})$	$m = \omega(n^{1.22})$	$n^{1.46} \leq m \leq n^{1.53}$
	$\widetilde{O}(m^{23/30}n^{4/3})$	$m = \omega(n^{1.43})$	$n^{1.53} \leq m \leq n^{1.82}$
	$\widetilde{O}(m^{5/7}n^{10/7})$	$m = \omega(n^{1.5})$	$n^{1.82} \leq m \leq n^2$
$(1+\epsilon)$-approx. weighted SSSP	$\widetilde{O}(m^{5/4}n^{5/8})$	$m = o(n^{1.5})$	$n \leq m \leq n^{1.43}$
	$\widetilde{O}(m^{25/27}n^{206/189})$	$m = \omega(n^{1.22})$	$n^{1.43} \leq m \leq n^{1.53}$
	$\widetilde{O}(m^{23/30}n^{4/3})$	$m = \omega(n^{1.43})$	$n^{1.53} \leq m \leq n^{1.82}$
	$\widetilde{O}(m^{5/7}n^{10/7})$	$m = \omega(n^{1.5})$	$n^{1.82} \leq m \leq n^2$

Table 1: Summary of our algorithms. The running time in the second column is the expected total update time omitting $O(\text{poly}\log n)$ factors. For $(1+\epsilon)$-approximate SSSP and $(1+\epsilon)$-approximate stSP on weighted graphs we assume that ϵ is a constant and that the largest edge weight is polynomial in n. The third column shows roughly where these algorithms give an improvement over the previous $O(mn)$-time algorithms. The fourth column indicates the density of the graph for which the algorithm gives the fastest running time.

Theorem 1.1. *There are the following algorithms for decremental reachability and shortest path problems in directed graphs with constant query time. They are correct with high probability and their total update time is in expectation.*

- stR and $O(\log n)$-approximate unweighted stSP: *The minimum of*
  - $\widetilde{O}(m^{7/5}n^{2/5})$,
  - $\widetilde{O}(m^{191/207}n^{226/207})$,
  - $\widetilde{O}(m^{25/33}n^{4/3})$, *and*
  - $\widetilde{O}(m^{1/2}n^{7/4})$.

  This can be simplified to, e.g., $\widetilde{O}(mn^{242/247}) = \widetilde{O}(mn^{0.980})$.
- SSR, SCC, and $O(\log n)$-approximate unweighted SSSP: *The minimum of*
  - $\widetilde{O}(m^{5/4}n^{5/8})$,
  - $\widetilde{O}(m^{191/207}n^{226/207})$,
  - $\widetilde{O}(m^{25/33}n^{4/3})$, *and*
  - $\widetilde{O}(m^{1/2}n^{7/4})$.

  This can be simplified to, e.g., $\widetilde{O}(mn^{266/271}) = \widetilde{O}(mn^{0.982})$.
- $(1+\epsilon)$-approximate weighted stSP: *The minimum of*
  - $\widetilde{O}(m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$,
  - $\widetilde{O}(m^{25/27}n^{206/189}(\log W)^3/\epsilon^4)$,
  - $\widetilde{O}(m^{23/30}n^{4/3}(\log W)^2/\epsilon^3)$, *and*
  - $\widetilde{O}((m^{5/7}n^{10/7}\log W)/\epsilon^2)$.

  This can be simplified to, e.g., $\widetilde{O}(mn^{55/56}(\log W)^3/\epsilon^4) = \widetilde{O}(mn^{0.982}(\log W)^3/\epsilon^4)$.
- $(1+\epsilon)$-approximate weighted SSSP: *The minimum of*
  - $\widetilde{O}(m^{5/4}n^{5/8}(\log W)^2/\epsilon^2)$,
  - $\widetilde{O}(m^{25/27}n^{206/189}(\log W)^3/\epsilon^4)$,
  - $\widetilde{O}(m^{23/30}n^{4/3}(\log W)^2/\epsilon^3)$, *and*
  - $\widetilde{O}((m^{5/7}n^{10/7}\log W)/\epsilon^2)$.

  This can be simplified to, e.g., $\widetilde{O}(mn^{241/245}(\log W)^3/\epsilon^4) = \widetilde{O}(mn^{0.984}(\log W)^3/\epsilon^4)$.

As a byproduct of our results we obtain algorithms for the following two problems, which might be of independent interest.

Theorem 1.2. *Given a set of nodes S of size k, we can, for all nodes x and y in S, maintain a $(1+\epsilon)$-approximation of the distance from x to y with constant query time and a total update time of $\widetilde{O}((k^2m + m^{5/7}n^{10/7}\log W)/\epsilon^2)$.*

Theorem 1.3. *Given k source/sink pairs $(s_i, t_i)_{1\le i\le k}$, we can, for all $1 \le i \le k$, maintain a $(1+\epsilon)$-approximation of the distance from s_i to t_i with constant query time and a total update time of $\widetilde{O}(k^{3/5}m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$.*

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. We do not know of any combinatorial algorithm (not relying on fast matrix multiplication) that gives a polynomial improvement over this running time. Thus, we cannot hope for a combinatorial algorithm for decremental all-pairs reachability with a total update time of $o(mn)$ and small query time.

- *(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 [23], 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) a worst case update and query time of $o(m)$ 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 [19] 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. This is the reason why approximation is necessary if we want to break the $O(mn)$ barrier.

We need at least three slightly different algorithms to break the $O(mn)$ bound. The obvious conceptual question is whether the same can be achieved using only one algorithm. In particular it would be interesting to see an algorithm that performs well when $m = \Theta(n^{1.5})$ as this seems to be the most challenging case. Our algorithms heavily use randomization. It remains open whether a total update time of $o(mn)$ can be achieved by a deterministic algorithm.

Organization. In Sections 2 and 3, we explain the main ideas behind our algorithms. In particular, we show how to obtain $o(mn)$ time for s-t reachability in Section 2. Then, in Section 3, we show how to extend this result to other problems, and present some ideas for speeding up our algorithms. In later sections we explain our algorithms in details. We provide necessary definitions in Section 4. In Section 5, we explain how to obtain $(1+\epsilon)$-approximate SSSP for dense weighted graphs. Then, In Section 6, we explain another such algorithm, which works well for sparse graphs. In Section 7, we show how to speed up the algorithm for dense unweighted graphs at the cost of a worse approximation guarantee. We extend our algorithms for stSP to SSSP in Section 8. Finally, in Section 9, we extend our algorithms for SSSP to maintain SCC.

2 Algorithm Overview for s-t Reachability

In this section, we illustrate the main ideas of our algorithms. We will focus on simple algorithms for the s-t reachability problem (stR), which might not be the most efficient ones. The total update time that we obtain at the end of this section is $\widetilde{O}(mn^{132/133})$.

Throughout, we let G be the input directed unweighted graph. For any nodes u and v, we let $\text{dist}_G(u, v)$ denote the distance from u to v in G. We say that an event happens *with high probability* (whp) if it happens with probability at least $1 - 1/n^c$, where c is an arbitrary constant. We abbreviate the breadth-first search tree by BFS-tree. Our algorithm will call as a subroutine the algorithm of Even and Shiloach [6], which we call *ES-tree*. This algorithm, in its general form, has a parameter D and works under edge deletions. It can maintain the distances from any node s to all other nodes up to distance D; i.e., for any node v, the query on v outputs $\text{dist}_G(s, v)$ if $\text{dist}_G(s, v) \leq D$ and ∞ otherwise. This algorithm takes $O(mD)$ total update time.

2.1 Path Union Graph, Hub-Distance, and Bounded-Hop Multi-Pair Reachability

At the heart of all our algorithms is a new way of maintaining reachability or distances between some *nearby* pairs of nodes using the ideas of *path union graph* and *hub-distance*.

Definition 2.1 (Hub-Distance $\text{dist}_{\mathcal{H}}(u, v)$ with parameter b)**.** *For any set $\mathcal{H}$ of nodes (called* hubs*), let* hub-distance *from any node u to node v, denoted by $\text{dist}_{G,\mathcal{H}}(u, v)$, be $\text{dist}_{G,\mathcal{H}}(u, v) = \min_{x \in \mathcal{H}} \text{dist}_G(u, x) + \text{dist}_G(x, v)$. When G is clear from the context, we use $\text{dist}_{\mathcal{H}}(u, v)$ instead of $\text{dist}_{G,\mathcal{H}}(u, v)$.*

Throughout this section, we define $\mathcal{H}$ as follows. Let b be a parameter. Let Y be a set of $b \operatorname{poly}\log n$ randomly selected nodes and edges. Let $\mathcal{H}$ be the set of nodes that are either in Y or incident to an edge in Y. (Note that $\mathcal{H}$ contains only nodes.) Thus, $\text{dist}_{\mathcal{H}}(u, v)$ is a random variable whose distribution depends on the parameter b.

Definition 2.2 (Path-Union Graph $\mathcal{P}(u, v, h, G)$ and distance $\text{dist}_{\mathcal{P}}(u, v)$)**.** *For any nodes u and v in a graph G and every integer h, we call a u-v path P an h-*hop *path if it contains at most h edges. We let the h-*hop *u-v* path union graph*, denoted by $\mathcal{P}(u, v, h, G)$, be the subgraph of G resulting from a union of all h-hop u-v paths in G; i.e. $V(\mathcal{P}(u, v, h, G))$ (respectively $E(\mathcal{P}(u, v, h, G))$) is the set of all nodes (respectively edges) that lie on some u-v paths of length at most h. When G and h are clear from the context, we use $\mathcal{P}(u, v)$ and $\text{dist}_{\mathcal{P}}(u, v)$ to denote $\mathcal{P}(u, v, h, G)$ and $\text{dist}_{\mathcal{P}(u,v,h,G)}(u, v)$ respectively.*

Bounded-Hop Multi-Pair Reachability Problem. Our algorithms solve the following problem as a subroutine. 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 i, whether $\text{dist}_G(s_i, t_i) \leq h$.

Our Framework. Intuitively, instead of maintaining whether $\text{dist}_G(s_i, t_i) \leq h$, we can maintain whether $\text{dist}_{\mathcal{P}}(s_i, t_i) \leq h$ since $\mathcal{P}(s_i, t_i)$ contains all s_i-t_i paths of length at most h in G. This could be helpful when $\mathcal{P}(s_i, t_i)$ is much smaller than G. Our first key idea is the observation that all (s_i, t_i) pairs with large $\mathcal{P}(s_i, t_i)$ can use their paths through a small number of hubs to check whether $\text{dist}_{\mathcal{P}}(s_i, t_i) \leq h$ (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.

- We maintain $\text{dist}_{\mathcal{H}}(s_i, t_i)$, for all $1 \leq i \leq k$, as long as $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq h$.
- Once $\text{dist}_{\mathcal{H}}(s_i, t_i) > h$, we construct $\mathcal{P}(s_i, t_i)$ and maintain $\text{dist}_{\mathcal{P}}(s_i, t_i)$ up to h.

Our algorithm will output $\text{dist}_G(s_i, t_i) \leq h$ if and only if either $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq h$ or $\text{dist}_{\mathcal{P}}(s_i, t_i) \leq h$. The correctness of this algorithm is obvious: either $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq h$ which already implies that $\text{dist}_G(s_i, t_i) \leq h$ or otherwise we maintain $\text{dist}_{\mathcal{P}}(s_i, t_i)$ 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 2.3 (Either hub-distance or path-union graph is small)**.** *With high probability, for all i, either $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq h$, or $\mathcal{P}(s_i, t_i)$ has at most n/b nodes and m/b edges.*

Proof Sketch. By a standard hitting set argument [5, 22], if $\mathcal{P}(s_i, t_i)$ has more than n/b nodes, then one of these nodes, say x, will be in $\mathcal{H}$ whp. By definition, x lies on some s_i-t_i path of length at most h. Thus, $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq \text{dist}_G(s_i, x) + \text{dist}_G(x, t_i) \leq h$. Similarly, if $\mathcal{P}(s_i, t_i)$ has more than m/b edges, then one of these edges, say (x, y), will be among the $b \operatorname{poly}\log n$ random edges we use to construct $\mathcal{H}$ whp. This means that x will be in $\mathcal{H}$, and thus $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq \text{dist}_G(s_i, x) + \text{dist}_G(x, t_i) \leq h$. □

Lemma 2.3 guarantees that when we start maintaining $\mathcal{P}(s_i, t_i)$, it is much smaller than G whp, and so it is beneficial to maintain the distance in $\mathcal{P}(s_i, t_i)$ instead of G. To this end, we analyze the time to maintain the hub-distance, which will be used in all algorithms in this section.

Lemma 2.4 (Maintaining all $\text{dist}_{\mathcal{H}}(s_i, t_i)$)**.** *For any h, we can maintain, for every i, whether $\text{dist}_{\mathcal{H}}(s_i, t_i) \leq h$ in $\widetilde{O}(mbh + kbh)$ total update time.*

Proof Sketch. For each hub $v \in \mathcal{H}$, we maintain the values of $\text{dist}_G(s_i, v)$ and $\text{dist}_G(v, t_i)$ up to h using ES-trees of depth h rooted at each hub. This takes $\widetilde{O}(mbh)$ total update time. Every time $\text{dist}_G(s_i, v)$ or $\text{dist}_G(v, t_i)$ changes, for some hub $v \in \mathcal{H}$, we update the value of $\text{dist}_G(s_i, v) + \text{dist}_G(v, t_i)$ for all $i \in \{1, \ldots, k\}$, incurring $O(k)$ time. Since there are $\widetilde{O}(b)$ hubs, and the value of each of $\text{dist}_G(s_i, v)$ and $\text{dist}_G(v, t_i)$ can change at most h times, we need $\widetilde{O}(kbh)$ time in total. □

How we construct $\mathcal{P}(s_i, t_i)$ and maintain $\text{dist}_{\mathcal{P}}(s_i, t_i)$ varies between algorithms. Here we note one simple way to do it.

Lemma 2.5 (Maintain each $\text{dist}_{\mathcal{P}}(s_i, t_i)$)**.** *(i) We can construct each $\mathcal{P}(s_i, t_i)$ in $\widetilde{O}(m)$ time. (ii) We can maintain each $\text{dist}_{\mathcal{P}}(s_i, t_i)$ up to h in time $\widetilde{O}(hm_{\mathcal{P}}(s_i, t_i))$ where $m_{\mathcal{P}}(s_i, t_i)$ denotes to the number of edges in $\mathcal{P}(s_i, t_i)$ when we construct it.*

Proof Sketch. We can construct $\mathcal{P}(s_i, t_i)$ by computing the distance from s_i and t_i to all nodes and edges, which can be done in $O(m)$ time using BFS-trees. Then $V(\mathcal{P}(s_i, t_i))$ (respectively $E(\mathcal{P}(s_i, t_i))$) is simply the set of nodes x (respectively edges (y, z)) such that $\text{dist}_G(s_i, x) + \text{dist}_G(x, t_i) \leq h$ (respectively $\text{dist}_G(s_i, y) + \text{dist}_G(z, t_i) + 1 \leq h$). Finally, we can maintain $\text{dist}_{\mathcal{P}}(s_i, t_i)$ by maintaining an ES-tree in $\mathcal{P}(s_i, t_i)$, taking $\widetilde{O}(hm_{\mathcal{P}}(s_i, t_i))$ time. □

By Lemma 2.3, whp, our algorithm constructs $\mathcal{P}(s_i, t_i)$ when $|E(\mathcal{P}(s_i, t_i))| \leq m/b$. It follows that we can maintain whether $\text{dist}_G(s_i, t_i) \leq h$, for all $1 \leq i \leq k$, in $\widetilde{O}(mbh + kbh + mk + mhk/b)$ total update time. Note that, previously, the fastest way to maintain whether $\text{dist}_G(s_i, t_i) \leq h$ is to maintain an ES-tree separately for each pair. This takes $\widetilde{O}(mhk)$ time. Our new running time is faster when we set b to be smaller than m and k. This will be how we set b when we use this algorithm later in this section.

2.2 Algorithm for Dense Graphs ($m = \omega(n^{3/2})$)

The following *center graph* is another important notion that will be used in all algorithms.

Definition 2.6 (Center graph $\mathcal{C}(G,c)$ with parameter c)**.** *Let c be a parameter, and C be a set containing the source s, the sink t, and $c\operatorname{poly}\log n$ randomly selected nodes; we call nodes in C* centers*. Define the* center graph*, denoted by $\mathcal{C}(G,c)$, as follows. The Nodes of $\mathcal{C}(G,c)$ are the centers in C. For every pair of centers u and v in C, there is a directed edge (u,v) in $\mathcal{C}(G,c)$ if and only if $\operatorname{dist}_G(u,v) \le n/c$. Note that $\mathcal{C}(G,c)$ is a random graph with $c\operatorname{poly}\log n$ nodes.*

Lemma 2.7 ($\mathcal{C}(G,c)$ preserves s-t reachability)**.** *Whp, s can reach t in G if and only if it can do so in $\mathcal{C}(G,c)$.*

Proof Sketch. Since we can convert any path in $\mathcal{C}(G,c)$ to a path in G, the "if" part is clear. To prove the "only if" part, let P be an s-t path in G. A standard hitting set argument [5, 22] can be used to show that, with high probability, there is a set of centers $c_1, c_2, \ldots, c_k \in V(P)$ such that $c_1 = s$, $c_k = t$, and $\operatorname{dist}_G(c_i, c_{i+1}) \le n/c$ for all i. The last property implies that edge (c_i, c_{i+1}) is contained in $\mathcal{C}(G,c)$ for all i. Thus s can reach t in $\mathcal{C}(G,c)$. □

Lemma 2.7 implies that to maintain s-t reachability in $\mathcal{C}(G,c)$ it is sufficient to maintain the edges of $\mathcal{C}(G,c)$ and to maintain s-t reachability in $\mathcal{C}(G,c)$. To maintain the edges of $\mathcal{C}(G,c)$, we simply have to maintain the distance between all pairs of centers up to n/c. This can be done using the algorithm in Section 2.1 with $k = c^2$ and $h = n/c$, where we make each pair of centers a source-sink pair. Note that, for any centers u and v, $|E(\mathcal{P}(u,v))| \le |V(\mathcal{P}(u,v))|^2$ which, by Lemma 2.3, is at most $(n/b)^2$ whp when we start maintaining $\mathcal{P}(u,v)$. Thus, by Lemmas 2.4 and 2.5, the total time is $\widetilde{O}(mbh + kbh + mk + n^2hk/b^2)$ whp, for any choice of parameters b and c.. To maintain s-t reachability in $\mathcal{C}(G,c)$, note that $\mathcal{C}(G,c)$ undergoes only edge deletions when G undergoes edge deletions; so, we can simply maintain an ES-tree on $\mathcal{C}(G,c)$, which takes $O(|E(\mathcal{C}(G,c))| \cdot |V(\mathcal{C}(G,c))|) = \widetilde{O}(c^3)$ time. By replacing $k = c^2$ and $h = n/c$, the total update time becomes

$$\widetilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } \operatorname{dist}_{\mathcal{H}} \\ \text{(Lemma 2.4)}}} + \underbrace{mc^2}_{\substack{\text{construct } \mathcal{P}(u,v) \\ \text{(Lemma 2.5 (i))}}} + \underbrace{n^3c/b^2}_{\substack{\text{maintain } \operatorname{dist}_{\mathcal{P}}(u,v) \\ \text{(Lemma 2.5 (ii))}}} + \underbrace{c^3}_{\text{maintain } \operatorname{dist}_{\mathcal{C}(G,c)}(s,t)} \Big). \tag{1}$$

By setting[3] $b = n^{8/7}/m^{3/7}$ and $c = (bn)^{1/3} = n^{5/7}/m^{1/7}$, we get the running time of[4]

$$\widetilde{O}\left(m^{5/7}n^{10/7} + n^{20/7}/m^{4/7}\right). \tag{2}$$

This is faster than $O(mn)$ when $m = \omega(n^{3/2})$.

2.3 Algorithm for Sparse Graphs ($m = o(n^{4/3})$)

Maintaining s-t reachability in sparse graphs, especially when $m = O(n)$, needs quite a different approach. Carefully examining the running time of the previous algorithm in Equation (1) reveals that we *cannot* maintain $\operatorname{dist}_{\mathcal{P}}(u,v)$ for *all* pairs of centers at all times: this costs n^3c/b^2 (the fourth term of Equation (1)) which means that we need $b = \omega(n)$ (since we always need $c = \omega(b)$ to keep the first term of Equation (1) to $mbn/c = o(mn)$); this means that we need more hubs than the

[3] 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 $m \le n^2$.

[4] **Detailed calculation for Equation (2):** 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 $bnc = c^4 = n^{20/7}/m^{4/7}$.

number of nodes, which is impossible. The new strategy is to maintain $\text{dist}_{\mathcal{P}}(u, v)$ only for *some* pairs of centers at each time step.

Algorithm. As before, we have roughly b hubs (Definition 2.1) and c centers (Definition 2.6), and maintain $\text{dist}_{\mathcal{H}}(\cdot, \cdot)$ between all centers which takes $\widetilde{O}(mbn/c + bnc)$ time (first two terms of Equation (1)). The algorithm runs in phases. In 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 (unique) 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

$$\forall i,\ \text{dist}_G(c_i, c_{i+1}) \leq n/c \ \text{ and } \ \text{dist}_G(c_i, c_{i+2}) > n/c\,. \tag{3}$$

The first inequality holds because of a standard hitting set argument [5, 22] and the second one holds because, otherwise, we can remove c_{i+1} from the list L without breaking the first inequality. Observe that L induces an s-t path in the center graph $\mathcal{C}(G, c)$. Our intention in this phase is to maintain whether s can still reach t using this path; in other words, whether $\text{dist}_G(c_i, c_{i+1}) \leq n/c$ for all i. (We start a new phase if this is not the case.) We do this using the framework of Section 2.1: For each pair (c_i, c_{i+1}), we know that $\text{dist}_G(c_i, c_{i+1}) \leq n/c$ when $\text{dist}_{\mathcal{H}}(c_i, c_{i+1}) \leq n/c$. Once $\text{dist}_{\mathcal{H}}(c_i, c_{i+1}) > n/c$, we construct $\mathcal{P}(c_i, c_{i+1}) := \mathcal{P}(c_i, c_{i+1}, n/c, G)$. For any deletion of an edge e in this phase, we remove e from all $\mathcal{P}(c_i, c_{i+1})$ that contains it and check whether $\text{dist}_{\mathcal{P}}(c_i, c_{i+1}) \leq h$ using a static BFS algorithm. We start a new phase when $\text{dist}_G(c_i, c_{i+1})$ for some pair of centers (c_i, c_{i+1}) changes from $\text{dist}_G(c_i, c_{i+1}) \leq n/c$ to $\text{dist}_G(c_i, c_{i+1}) > n/c$

Running Time Analysis. First, let us bound the number of phases. As for each pair (c_i, c_{i+1}) the distance $\text{dist}_G(c_i, c_{i+1})$ can only become larger than n/c at most once, there are at most $\widetilde{O}(c^2)$ phases. In the beginning of each phase, we have to construct a BFS-tree in G, taking $O(m)$ time and contributing $\widetilde{O}(mc^2)$ to the total time over all phases. During the phase, we have to construct $\mathcal{P}(c_i, c_{i+1})$ for at most c pairs of center, taking $\widetilde{O}(mc)$ time (by Lemma 2.5) and contributing $\widetilde{O}(mc^3)$ total time. Moreover, after deleting an edge e, we have to update $\text{dist}_{\mathcal{P}}(c_i, c_{i+1})$ for every $\mathcal{P}(c_i, c_{i+1})$ that contains e, by running a BFS algorithm. This takes $O(|E(\mathcal{P}(c_i, c_{i+1})|)$ time for each $\mathcal{P}(c_i, c_{i+1})$, which is $\widetilde{O}(m/b)$ whp, by Lemma 2.3. The following lemma implies that e will be in only a constant number of such $\mathcal{P}(c_i, c_{i+1})$; so, we need $\widetilde{O}(m^2/b)$ time to update $\text{dist}_{\mathcal{P}}(c_i, c_{i+1})$ over all m deletions.

Lemma 2.8. *For any i and $j \geq i + 3$, $E(\mathcal{P}(c_i, c_{i+1}))$ and $E(\mathcal{P}(c_j, c_{j+1}))$ are disjoint.*

Proof. First, we claim that $\text{dist}_G(c_i, c_{j+1}) > 2n/c$. To see this, let G' be the graph in the beginning of the current phase. We know that $\text{dist}_{G'}(c_i, c_{i+4}) = \text{dist}_{G'}(c_i, c_{i+2}) + \text{dist}_{G'}(c_{i+2}, c_{i+4}) > 2n/c$, where the equality holds because in 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 (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', $\text{dist}_{G'}(c_i, c_{j+1}) \geq \text{dist}_{G'}(c_i, c_{i+4}) > 2n/c$. The claim follows since the distance between two nodes never decreases after edge deletions.

Now, suppose for the sake of contradiction that there is a directed edge (u, v) that is in both $E(\mathcal{P}(c_i, c_{i+1}))$ and $E(\mathcal{P}(c_j, c_{j+1}))$. This means that (u, 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. Using the fact that u is on such paths, i.e. $\text{dist}_G(c_i, u) \leq n/c$ and $\text{dist}_G(u, c_{j+1}) \leq n/c$, we have $\text{dist}_G(c_i, c_{j+1}) \leq \text{dist}_G(c_i, u) + \text{dist}_G(u, c_{j+1}) \leq 2n/c$. This contradicts the claim above. □

Thus, the total update time is

$$\widetilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } \text{dist}_{\mathcal{H}} \\ \text{(Lemma 2.4)}}} + \underbrace{mc^2}_{\substack{\text{construct BFS} \\ \text{tree in every phase}}} + \underbrace{mc^3}_{\substack{\text{construct } \mathcal{P}(c_i, c_{i+1}) \\ \text{in every phase}}} + \underbrace{m^2/b}_{\text{update } \text{dist}_{\mathcal{P}}(c_i, c_{i+1})} \Big).$$

By setting $c = (mn)^{1/7}$ and $b = c^4/n = m^{4/7}/n^{3/7}$, we get a running time of [5]

$$\widetilde{O}\left(m^{10/7}n^{3/7}\right). \tag{4}$$

This is better than $O(mn)$ when $m = o(n^{4/3})$.

2.4 Algorithm for the Remaining Graphs

Using the algorithms of Sections 2.2 and 2.3, we can beat the $O(mn)$-time barrier when $m = \omega(n^{3/2})$ and $m = o(n^{4/3}) \approx o(n^{1.34})$. To deal with the in-between cases, we simply optimize some parts of the previous two algorithms. There are many ways to do so, which we explain in later sections. In this section, we will only mention one of them, which leads us to beating the $O(mn)$-time barrier when $m = \omega(n^{17/14}) \approx \omega(n^{1.22})$. Thus, we beat the $O(mn)$-time barrier on all graphs when we combine this with the algorithm for sparse graphs (we combine them in the sense that before the first deletion happens we determine the number of edges in the input graph and use an appropriate algorithm to maintain s-t reachability).

Recursion for Less-Dense Graphs ($m = \omega(n^{10/7})$). Let us make one simple observation on the algorithm for dense graphs (Section 2.2): Recall that, for any pair of centers $u, v \in C$, we want to maintain whether $\text{dist}_{\mathcal{P}}(u, v) \leq h$ so that we can remove edge (u, v) from the center graph $\mathcal{C}(G, c)$ when $\text{dist}_{\mathcal{P}}(u, v) > h$. Observe, however, that we can actually leave this edge in the center graph as long as u can reach v in $\mathcal{P}(u, v)$ – we can still easily guarantee that $\mathcal{C}(G, c)$ preserves s-t reachability (as in Lemma 2.7) with this modification.

This observation allows us to recursively use our new s-t reachability algorithm from Section 2.2 with a total update time of $\widetilde{O}(m^{5/7}n^{10/7} + n^{20/7}/m^{4/7})$ to maintain reachability in each $\mathcal{P}(u, v)$; if $\mathcal{P}(c_i, c_j)$ has n_{ij} nodes and m_{ij} edges, this costs

$$\widetilde{O}\left(c^2 m_{ij}^{5/7} n_{ij}^{10/7} + c^2 n_{ij}^{20/7}/m_{ij}^{4/7}\right) = \widetilde{O}\left(c^2 n_{ij}^{20/7}\right)$$

total update time over all c^2 pairs of centers (the equality holds because $1 \leq m_{ij} \leq n_{ij}^2$). By Lemma 2.3, $n_{ij} \leq n/b$ whp; so, we have the total update time of

$$\widetilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } \text{dist}_{\mathcal{H}} \\ \text{(Lemma 2.4)}}} + \underbrace{mc^2}_{\substack{\text{construct } \mathcal{P}(u,v) \\ \text{(Lemma 2.5)}}} + \underbrace{c^2(n/b)^{20/7}}_{\substack{\text{maintain} \\ \text{dist}_{\mathcal{P}}(u,v)}} + \underbrace{c^3}_{\text{maintain } \text{dist}_{\mathcal{C}(G,c)}(s,t)} \Big).$$

By setting $b = n/m^{7/20}$ and $c = (bn)^{1/3} = n^{2/3}/m^{7/60}$, we get the total update time of [6]

$$\widetilde{O}\left(m^{23/30}n^{4/3} + n^{8/3}/m^{7/15}\right). \tag{5}$$

[5] **Detailed calculation for Equation (4):** 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, $m^2/b = m^2/(m^{4/7}/n^{3/7}) = m^{(14-4)/7}n^{3/7}$.

[6] **Detailed calculation for Equation (5):** First note that $c \leq n^{2/3} \leq m$. So, the term c^3 is dominated by the term mc^2. Moreover, $(n/b)^{20/7} = (m^{7/20})^{20/7} = m$; so, the fourth term is the same as the third term (mc^2). Using $c = (bn)^{1/3}$, we have that the first and the third term are the same. Now, the third term is $mc^2 = m(n^{2/3}/m^{7/60})^2 = m^{(60-14)/60}n^{4/3} = m^{23/30}n^{4/3}$. For the second term, using $bn = c^3$, we have $bnc = c^4 = n^{8/3}/m^{7/15}$.

This is faster than $O(mn)$ when $m = \omega(n^{10/7})$.

Recurse Again ($m = \omega(n^{17/14})$). Again, we can replace the fourth term of Equation (1) by $\widetilde{O}(c^2 m_{ij}^{23/30} n_{ij}^{4/3} + c^2 n_{ij}^{8/3}/m_{ij}^{7/15})$. By Lemma 2.3, $n_{ij} \leq n/b$ and $m_{ij} \leq m/b$ whp; so, we have the total running time of

$$\widetilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } \text{dist}_{\mathcal{H}} \\ \text{(Lemma 2.4)}}} + \underbrace{mc^2}_{\substack{\text{construct } \mathcal{P}(c_i,c_j,n/c,G) \\ \text{(Lemma 2.5)}}} + \underbrace{c^2(m/b)^{23/30}(n/b)^{4/3} + c^2(n/b)^{8/3}/(m/b)^{7/15}}_{\substack{\text{maintain} \\ \text{dist}_{\mathcal{P}(c_i,c_j,n/c,G)}(c_i,c_j)}} + \underbrace{c^3}_{\text{maintain } \text{dist}_{\mathcal{C}(G,c)}(s,t)} \Big).$$

By setting $b = n^{40/63}/m^{1/9}$ and $c = (bn)^{1/3} = n^{103/189}/m^{1/27}$, we get the running time of [7]

$$\widetilde{O}\left(m^{25/27} n^{206/189} + n^{446/189}/m^{8/27}\right). \tag{6}$$

This is faster than $O(mn)$ when $m = \omega(n^{17/14})$. [8]

Putting Everything Together. We now put the algorithm we just presented together with the algorithm for sparse graphs (Section 2.3). We get the total update time, which can be written, for example, as [9]

$$\widetilde{O}\left(\min\left(m^{10/7} n^{3/7}, m^{25/27} n^{206/189} + n^{446/189}/m^{8/27}\right)\right) = \widetilde{O}\left(mn^{132/133}\right). \tag{7}$$

3 Overview of Extensions to Other Problems

Note that, as in Section 2, algorithms we present here might not be the most efficient ones that we obtain, see later sections for those. The total update times that we will present in this section are $\widetilde{O}(mn^{265/266})$ for maintaining single-source reachability, single-source $(1+\epsilon)$-approximate shortest paths, and strongly connected components. We also sketch some ideas for speeding up our algorithms.

3.1 Single-Source Reachability

We use the idea of Bernstein [3] to extend our s-t reachability algorithms to ones for single-source reachability. Let s be the source. We show the following.

[7] **Detailed calculation for Equation (6):** First note that $c \leq n^{103/189} \leq m$. So, the term c^3 is dominated by the term mc^2. Moreover, $(m/b)^{23/30}(n/b)^{4/3} = (m^{10/9}/n^{40/63})^{23/30}(n^{23/63}m^{1/9})^{4/3} = (m^{230/270}/n^{40\cdot 23/63\cdot 30})(n^{23\cdot 4/63\cdot 3} m^{4/9\cdot 3}) = m$. So, the fourth term is the same as the third term (mc^2). Using $c = (bn)^{1/3}$, we have that the first and the third term are the same. Now, the third term is $mc^2 = m(n^{103/189}/m^{1/27})^2 = m^{1-2/27} n^{206/189} = m^{25/27} n^{206/189}$. For the second term, using $bn = c^3$, we have $bnc = c^4 = n^{409/189}/m^{4/27}$. Since $n^{409/189}/m^{4/27} \leq m^{25/27} n^{206/189}$ as long as $m \geq n$, this term can be ignored. For the fifth term we use $c^2 = (bn)^{2/3}$ to get $c^2(n/b)^{8/3}/(m/b)^{7/15} = n^{2/3+8/3} b^{2/3-8/3+7/15}/m^{7/15} = \frac{n^{10/3}}{m^{7/15} b^{23/15}}$. Now use $b = \frac{n^{40/63}}{m^{1/9}}$ to write the fifth term as $\frac{n^{10/3}}{m^{7/15}(n^{40/63}/m^{1/9})^{23/15}} = \frac{n^{10/3-(40/63)(23/15)}}{m^{7/15-(1/9)(23/15)}} = \frac{n^{446/189}}{m^{8/27}}$.

[8] The number 17/14 can be found by solving the equation $x + 1 = \max(25x/27 + 206/189, 446/189 - 8x/27)$.

[9] The number 132/133 can be obtained by finding the maximum value x of the function $f(x) = \min(3/7 + 10x/7 - x, 206/189 + 25x/27 - x)$ in the range $1 \leq x \leq 2$.

Theorem 3.1 (From stR to SSR)**.** *If we can maintain s-t reachability in $T(m,n)$ time, then we can maintain single-source reachability in $\widetilde{O}(cT(m,n)+mn/c)$ for any $1 \le c \le n$. Our data structure is correct whp.*

Proof Sketch. We select a set C of $c \operatorname{poly}\log n$ randomly selected nodes called centers. We will maintain a graph G', obtained by adding to G edges (s,v) for all center $v \in C$ that can be reached from s. To maintain G', we run the s-t reachability algorithm with source s and sink v for each center v. This takes $\widetilde{O}(cT(m,n))$ time. Consider any node x in G'. Obviously, s can reach x in G if and only if it can do so in G'. Moreover, by a standard hitting set argument, $\operatorname{dist}_{G'}(s,x) \le n/c$ whp (since, intuitively, there must be a center nearby x). Thus, we can maintain single-source reachability in G by maintaining an ES-tree in G' of depth n/c. This takes $O(|E(G')|n/c) = O(mn/c)$ time. □

If we have an s-t reachability algorithm with running time, e.g., $\widetilde{O}(mn^{1-\epsilon})$ for some constant $\epsilon > 0$, then we can set $c = n^{\epsilon/2}$ to get $\widetilde{O}(mn^{1-\epsilon/2})$ time for single source reachability, thus breaking the $O(mn)$ time barrier. So, if we use the running time of $\widetilde{O}(n^{132/133})$ in Equation (7), we get a running time of $\widetilde{O}(mn^{265/266})$.

3.2 Single-Source Shortest Paths

We can also use the algorithmic framework introduced above to obtain a $(1+\epsilon)$-approximate SSSP data structure. Again it will be sufficient to develop a data structure for stSP. We then obtain a data structure for SSSP by using a reduction similar to the one of Theorem 3.1. We just sketch the ideas here.

In unweighted graphs, we get the $(1+\epsilon)$-approximation by slightly increasing the number of centers. Instead of $\widetilde{O}(c)$ randomly chosen centers, we use $\widetilde{O}(c/\epsilon)$ randomly chosen centers. This guarantees that on the shortest path from s to t we can find centers $s = c_1, c_2, \ldots, c_k = t$ with the following properties: $(1-\epsilon)n/c \le \operatorname{dist}_G(c_i, c_{i+1}) \le n/c$ for $1 \le i \le k-2$ and $\operatorname{dist}_G(c_{k-1}, c_k) \le n/c$. Using hubs and path union graphs, our algorithm knows that c_{i+1} can be reached from c_i with at most n/c hops. As the distance from c_i to c_{i+1} is at least $(1-\epsilon)n/c$ (except when $i = k-1$), our algorithm replaces paths of length $(1-\epsilon)n/c$ by paths of length n/c. This guarantees a multiplicative error of $1+2\epsilon$ except for the path from c_{k-1} to c_k, which adds an additive error of n/c. This additive error can then be turned into a multiplicative error by additionally maintaining an ES-tree of depth $n/(c\epsilon)$ from the source.

We can extend the same ideas to the weighted case, except that we have to maintain the distances between nodes using Bernstein's algorithm [3] instead of the ES-tree. This algorithm provides a $(1+\epsilon)$-approximation of the distance from a source node to all nodes within h hops in $\widetilde{O}((mh \log W)/\epsilon)$ time. Using suitable modifications of our central concepts, the results for our reachability algorithms carry over to $(1+\epsilon)$-approximate shortest paths.

3.3 Strongly Connected Components

In the strongly connected components problem, we want the operation `query`(k) on a node v to return a number such that the query returns the same number on any two nodes if and only if they are in the same strongly connected component (see Definition 4.2 for detail). Roditty and Zwick [18] presented an algorithm for this problem which takes $O(mn)$ total update time. Their algorithm maintains several $O(mn)$-time algorithms for single-source reachability (i.e. ES-trees) as a subroutine. We analyze their algorithm (with some small modifications) when we replace the

previous $O(mn)$-time algorithm by an $\widetilde{O}(m^\alpha n^\beta)$-time algorithm, as in the following theorem (see Theorem 9.1 for detail).

Theorem 3.2 (From SSR to SCC)**.** *Suppose that single-source reachability can be maintained in $\widetilde{O}(m^\alpha n^\beta + m)$ total update time[10]. If $\alpha \geq 1$ or $\beta \geq 1$, then we can also maintain strongly connected components in $\widetilde{O}(m^\alpha n^\beta + m)$ time.*

This theorem, combined with our $\widetilde{O}(mn^{265/266})$-time algorithm for single-source reachability (Section 3.1), implies that we can maintain strongly connected components with $\widetilde{O}(mn^{265/266})$ total update time. We note that, in proving the above theorem, Roditty and Zwick analyzed a certain recursive function (in particular, $f(m,n)$ in [18, Theorem 2.1]). Due to some technical difficulties, however, we do not bound this function. Instead, we analyze the cost incurred in each step using some charging argument.

Charging Argument (Sketched). For simplicity, let us assume that $\alpha = 1$; so, we have the running time of $\widetilde{O}(mn^\beta)$ for some $0 \leq \beta \leq 1$. The key idea of the Roditty-Zwick algorithm is to maintain reachability from exactly one *random* node in each strongly connected component. This takes $m_0 n_0^\beta$ time for a strongly connected component C_0 having n_0 nodes and m_0 edges. Let s be the source node in C_0 from which we maintain reachability. Suppose now that, after an edge deletion, C_0 breaks into two strongly-connect components, say C_1 and C_2. One of them, say C_i (for some $i \in \{1,2\}$), will contain s, and we have to maintain reachability from a new source in the other component, say C_j. This incurs an additional cost of $\widetilde{O}(m_j n_j^\beta)$ for maintaining reachability from the new source, where m_j and n_j denote the number of edges and nodes in C_j, respectively[11]. In expectation, the total cost is

$$\widetilde{O}\left(\frac{n_1}{n_0} m_2 n_2^\beta + \frac{n_2}{n_0} m_1 n_1^\beta\right).$$

We split this cost to (i) a charge of $(n_1 n_2^\beta)/n_0$ on m_2 edges in C_2 and (ii) a charge of $(n_2 n_1^\beta)/n_0$ on m_1 edges in C_1. In other words, on every edge $e \in C_i$ for $i \in \{1,2\}$, we put a charge of

$$\frac{|V(C_0) \setminus V(C_i)| \cdot |V(C_i)|^\beta}{|V(C_0)|} \leq \frac{|V(C_0) \setminus V(C_i)|}{|V(C_0)|} n^\beta.$$

Now we analyze the total charge on each edge e. Let $C'_1, C'_2, \ldots, C'_k$ be the sequence of strongly connected components that contain e under edge deletions (thus $V(C'_1) \supset V(C'_2) \supset \ldots \supset V(C'_k)$). When an edge deletion causes the strongly connected component that contains e to change from C'_i to C'_{i+1}, we charge the cost of at most

$$\frac{|V(C'_i) \setminus V(C'_{i+1})| \cdot |V(C'_{i+1})|^\beta}{|V(C'_i)|} \leq \frac{|V(C'_i) \setminus V(C'_{i+1})|}{|V(C'_i)|} n^\beta.$$

So, the total charge on e is

$$\begin{aligned}
&\widetilde{O}\left(n^\beta \left(\frac{|V(C'_1) \setminus V(C'_2)|}{|V(C'_1)|} + \frac{|V(C'_2) \setminus V(C'_3)|}{|V(C'_2)|} + \ldots + \frac{|V(C'_{k-1}) \setminus V(C'_k)|}{|V(C'_{k-1})|}\right)\right) \\
&= \widetilde{O}\left(n^\beta \left(\frac{1}{|V(C'_1)|} + \frac{1}{|V(C'_1)|-1} + \frac{1}{|V(C'_1)|-2} + \ldots + \frac{1}{2} + \frac{1}{1}\right)\right) \\
&= \widetilde{O}\left(n^\beta \log |V(C'_1)|\right) \quad = \quad \widetilde{O}\left(n^\beta\right)
\end{aligned}$$

[10] The extra $\widetilde{O}(m)$ term is to guarantee that our running time is never $o(m)$, which is impossible.

[11] We note that Roditty-Zwick algorithm also requires an additional cost of $O(m_0 n_0)$ since it has to run the static algorithm to compute strongly connected components in C_0. By a simple modification, this cost can be avoided.

where the first equality holds because

$$\frac{|V(C'_i) \setminus V(C'_{i+1})|}{|V(C'_i)|} \leq \frac{1}{|V(C'_i)|} + \frac{1}{|V(C'_i)|-1} + \ldots + \frac{1}{|V(C'_{i-1})|+1}.$$

So, we now have that every edge is charged by at most $\widetilde{O}(n^\beta)$, the total update time is thus $\widetilde{O}(mn^\beta)$ as claimed.

3.4 Improvement with Hierarchical Graph Decomposition

In order to further speed up the s-t reachability algorithm we use the following data structure to reduce the time for computing $\mathcal{P}(s_i, t_i)$ in dense graphs. We call it the *approximate reachability data structure (AR data structure).* It is initialized with an unweighted, directed graph G with a source s and a depth h and maintains a subgraph $R(s)$ of G such that $R(s)$ contains all nodes reachable from s with a path of length at most h. It supports two operations: (1) an edge delete operation `delete`(u, v) that removes (u, v) from $R(s)$ and (2) an update operation `remove`(k) that removes at least k nodes of distance larger than $(h+1)\lceil \log n \rceil$ and returns *success* if there are at least k such nodes, and removes an arbitrary number of nodes and returns *no success* otherwise. Each delete operation takes constant time, while the total time for *all* remove operations that return success is $O(n^2)$, and each remove operation that returns unsuccessfully takes time $O(\min(n^2, m \log n))$. The AR data structure is implemented using a hierarchical graph decomposition similar to the ones in [10, 4]. We sketch this data structure below. See Section 7.1 for details.

In our application of the AR data structure for decremental s-t reachability we keep an AR data structure for each center x and make sure that whp every remove operation returns success. Thus each AR data structure incurs a total cost of only $O(n^2)$. It can be used to reduce the total time to construct the graphs $\mathcal{P}(s_i, t_i)$ from $\widetilde{O}(mc^2)$ to $\widetilde{O}(n^2c + (n^2/b^2)c^2)$ as follows. We describe here only the idea of this approach, the details are in Section 7.2. From the start of the algorithm we keep for every center x the graph $R(x)$ in an AR data structure. Whenever an edge (u, v) is deleted, we perform a `delete`(u, v) operation in all AR data structures. We slightly modify the rule of when to start maintaining $\mathcal{P}(x, y)$: we maintain it when there exists no hub v such that both x and y are within $4h \log n$ of v. When we need to construct the path union graph $\mathcal{P}(x, y)$ between x and another center y we first perform a backward breadth-first search from y in $R(x)$ up to depth h. Let $H(x, y)$ be the graph induced by the nodes visited by this BFS. Next we perform a forward BFS from x in $H(x, y)$ up to distance h. Then the graph induced by the nodes visited by this second BFS equals $\mathcal{P}(x, y)$. Let n_H be the number of nodes in $H(x, y)$. Now we call `remove`$(n_H - n/b)$ in the AR data structure of x. This remove operation will return success with high probability as an argument similar to the argument in the proof of Lemma 2.3 shows that whp $H(x, y)$ contains only n/b nodes v with distance from x of at most $(h+1)\lceil \log n \rceil$ when the path union graph $\mathcal{P}(x, y)$ is constructed. If that was not the case then whp a hub v would exist such that there would be a path from x to y through v of length at most $4h \log n$ and this would contradict our condition for when to build a path union graph $\mathcal{P}(x, y)$.

The running time analysis now works as follows: Additionally to the total time of $O(n^2)$ for maintaining the AR data structure of x it takes time linear in the number of edges in $H(x, y)$ to determine $\mathcal{P}(x, y)$. We charge this time in two parts: Let n_r be the number of nodes deleted from $R(x)$ by the `remove`$(n_H - n/b)$ operation. We charge each of these nodes n to account for all of the edges incident to them in $H(x, y)$. This gives a total charge of $O(n^2)$ for center x over the whole algorithm. Whp $n_r \geq n_H - n/b$ and thus whp there are only n/b nodes in $H(x, y)$ that are not removed and they only have n^2/b^2 edges between them. We charge this part of the computation

time for $\mathcal{P}(x, y)$ to the pair of centers (x, y). Thus the total time for building all path union graphs is $O(n^2c + (n^2/b^2)c^2)$.

To implement the AR data structure we fix an order of the out-edges for each node. A delete operation simply removes the edge from its list. A `remove`(k) operation uses a hierarchical graph decomposition of $R(x)$ into graphs $R_i(x)$ for $1 \leq i \leq \log n$ such that $R_i(x)$ is a subgraph of $R_{i+1}(x)$ and it has $O(2^i n)$ edges. Specifically, every node has as out-edges in $R_i(x)$ its first up to 2^i out-edges of $R(x)$. If a node has degree larger than 2^i in $R(x)$ it is blue in $R_i(x)$. Now starting from $i = 2$ the a `remove`(k) operation builds $R_i(x)$, performs a BFS in $R_i(x)$ with s and all its blue nodes as sources (i.e., level 0 nodes) and checks whether a blue node of $R_{i-1}(x)$ is at distance larger than $h + 1$ from all sources. If so it removes all nodes with distance from the sources in $R_i(x)$ larger than h from $R(x)$. If at least k nodes were removed, it stops, otherwise it increments i and repeats. Obviously no node with distance at most h is ever removed and one can also show that every node that is not removed has distance at most $(h + 1)\lceil \log n \rceil$ from s. To analyze the running time note that if a successful `remove`(k) operation removes the k-th node in $R_i(x)$ then there must exist a blue node from $R_{i-1}(x)$, i.e., a node with out-degree at least 2^{i-1} at distance more than $h+1$ from the sources of $R_i(x)$ and thus all its out-neighbors in $R_i(x)$ must be removed as well. Hence $\Omega(2^i)$ nodes are removed and the time to build and check all graphs up to $R_i(x)$ is $O(2^i n)$. We charge a cost of $O(n)$ to each removed node to account for this running time, giving a total time of $O(n^2)$ for all update operations that return success. Together with the algorithmic ideas of Sections 2.1 and 2.2 we get the following $O(\log n)$-approximation for stSP.

Theorem 3.3. *There is an $O(\log n)$-approximate stSP data structure for unweighted directed graphs undergoing edge deletions with constant query time and a total update time of $\widetilde{O}(m^{1/2}n^{7/4})$.*

Part II
Full Details

In the following we provide all technical details of our algorithms. We are trying to state our results as general as possible. Therefore we also consider weighted graphs for some of our problems.

4 Preliminaries

4.1 Problem Description

We are given a directed graph G that might be weighted or unweighted. In the weighted case we consider positive integer edge weights and denote the maximum edge weight by W. We denote the weight of an edge (u, v) in G by $w_G(u, v)$. In unweighted graphs we think of every edge weight to be equal to 1. The graph undergoes a sequence of *updates*, which might be edge deletions or, for weighted graphs, edge weight increases. This is called the *decremental setting*. We denote by n the number of nodes of G and by m the number of edges of G before any updates. We denote by Δ_G the number of updates in G, i.e., the number of edge deletions and edge weight increases. Note that for unweighted graphs we have $\Delta_G \leq m$ and for weighted graphs we have $\Delta_G \leq mW$.

For every path π we distinguish its *length* and its *weight*. The length of a path is the number of edges it contains. To avoid confusion we sometimes also say that a path of length h has h *hops*. We say that a node v is *reachable* from u (in G) if there is a path from u to v in G. The *distance* $\text{dist}_G(x, y)$ of a node x to a node y 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 $\text{dist}_G(x, y) = \infty$.

Our goal is to design efficient data structures for the following problems.

Definition 4.1 (Single-source reachability)**.** *A* decremental single-source reachability (SSR) data structure *for a directed graph G undergoing edge deletions and a source node s maintains all 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 (Strongly connected components)**.** *A* decremental strongly connected components (SCC) data structure *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 4.3 (γ-approximate stSP)**.** *A* decremental γ-approximate single-source single-sink shortest path (stSP) data structure *for a weighted directed graph G undergoing edge deletions and edge weight increases, a source node s, and a sink node t maintains a distance estimate $\delta(s, t)$ such that $\text{dist}_G(s, t) \leq \delta(s, t) \leq \gamma \cdot \text{dist}_G(s, t)$. It supports the following operations:*

- *Delete(u, v): Delete the edge (u, v) from G.*
- *Increase(u, v, x): Increase the weight of the edge (u, v) to x.*
- *Query(): Return the γ-approximate distance estimate $\delta(s, t)$.*

Definition 4.4 (γ-approximate SSSP)**.** *A* decremental γ-approximate single-source shortest paths (SSSP) data structure *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 $\text{dist}_G(s, v) \leq \delta(s, v) \leq \gamma \cdot \text{dist}_G(s, v)$. It supports the following operations:*

- *Delete(u, v): Delete the edge (u, v) from G.*
- *Increase(u, v, x): Increase the weight of the edge (u, v) to x.*
- *Query(v): Return the γ-approximate distance estimate $\delta(s, v)$.*

The approximation factors we obtain will be of the form $\gamma = O(\log n)$ or $\gamma = 1 + \epsilon$ for some ϵ such that $0 < \epsilon \leq 1$. All the data structures we design will have constant query time and we will compare them by their *total update time* over all deletions (and edge weight increases). We use $\widetilde{O}$-notation to hide $\log n$ and $\log \log W$ factors. We assume that arithmetic operations on integers can be performed in constant time. We will obtain data structures with a total update time of the form $\widetilde{O}(m^\alpha n^\beta)$ (possibly with additional $1/\epsilon$- and $\log W$-factors). Our data structures are randomized and will be correct with high probability (whp). The total update times we state are in expectation.

Our overall goal is to obtain data structures with a total update time of $o(mn)$. We will not achieve this with a single algorithm. For each of the problems listed above we will obtain several algorithms which break the $O(mn)$ bound for different values of m. We use a meta-algorithm that, before any updates in the graph, determines the number of edges in the graph and then uses the

data structure which has the best bound on the total update time for the given number of edges. The running time of this meta-algorithm is simply the minimum of the individual running times and will be $o(mn)$.

Note that the total update time of approximate stSP and SSSP data structures for weighted graphs will be $\Omega(\Delta_G)$, the number of edge deletions and edge weight increases in G. This is unavoidable as the data structure has to read every update in the graph. In unweighted graphs the dependence on Δ_G does not have to be stated explicitly because there we have $\Delta_G \leq m$ and the total update time is $\Omega(m)$ for reading the initial graph anyway.

4.2 Definitions and Basic Properties

The first concept that is needed by our algorithms are several modified definitions of the distance between nodes, which can be maintained efficiently under edge deletions and edge weight increases.

Definition 4.5. *For every integer $h \geq 1$ and all nodes x and y in a directed graph G, the* h-hops distance $\text{dist}^h_G(x,y)$ *is the minimum weight of all paths of length at most h from x to y in G.*

Note that in an unweighted graph G we have $\text{dist}^h_G(x,y) = \text{dist}_G(x,y)$ if $\text{dist}_G(x,y) \leq h$ and ∞ otherwise.

Lemma 4.6 (Even-Shiloach tree [6, 8, 13])**.** *There is a data structure, called* Even-Shiloach tree *(short: ES-tree), that, given an unweighted directed graph G undergoing edge deletions, a source node s, and a depth h, can maintain $\text{dist}^h_G(s,v)$ for every node v in total time $O(mh)$. By reversing the edges in G it can also maintain $\text{dist}^h_G(v,s)$ for every node v in the same time.*

Definition 4.7. *Let $h \geq 1$ and $0 \leq i \leq \lceil \log_{1+\epsilon}(nW) \rceil$ be integers and let x and y be nodes in a directed graph G. We define* rounded h-hops distance of range i $\widetilde{\text{dist}}^{h,i}_G(x,y)$ *to be the distance from x to y when every edge weight is rounded up to the nearest multiple of $\alpha = \epsilon(1+\epsilon)^i/h$. This means that $\widetilde{\text{dist}}^{h,i}_G(x,y) := \text{dist}_{G'}(x,y)$ where G' is the graph having the same nodes and edges as G and a weight function w' given by $w'(u,v) = \lceil w(u,v)/\alpha \rceil \cdot \alpha$ for every edge (u,v). The* rounded h-hops distance $\widetilde{\text{dist}}^{h}_G(x,y)$ *is defined by $\widetilde{\text{dist}}^{h}_G(x,y) = \min_i \widetilde{\text{dist}}^{h,i}_G(x,y)$.*

These rounded hops distances are useful because they provide $(1+\epsilon)$-approximate distance estimates.

Lemma 4.8 (Approximation guarantee of rounded hops distance [2, 3, 16])**.** *For all integers $h \geq 1$ and $0 \leq i \leq \lceil \log_{1+\epsilon}(nW) \rceil$ and all nodes x and y in a directed graph G, the following holds:*

- $\widetilde{\text{dist}}^{h,i}_G(x,y) \geq \text{dist}_G(x,y)$ *and* $\widetilde{\text{dist}}^{h}_G(x,y) \geq \text{dist}_G(x,y)$,
- *if* $\text{dist}^h_G(x,y) \geq (1+\epsilon)^i$, *then* $\widetilde{\text{dist}}^{h,i}_G(x,y) \leq (1+\epsilon)\,\text{dist}^h_G(x,y)$,
- $\widetilde{\text{dist}}^{h}_G(x,y) \leq (1+\epsilon)\,\text{dist}^h_G(x,y)$.

The main advantage of rounded distances is that in weighted graphs we know how to maintain them more efficiently than exact distances.

Lemma 4.9 (Algorithms for rounded hops distance [2, 3])**.** *Given integers $h \geq 1$ and $0 \leq i \leq \lceil \log_{1+\epsilon}(nW) \rceil$ and source node s, the following algorithms exist:*

- *Compute* $\widetilde{\text{dist}}_G^{h,i}(v,s)$ *and* $\widetilde{\text{dist}}_G^{h,i}(s,v)$ *for every node* v *in a (static) weighted directed graph* G *in time* $\widetilde{O}(m)$.

- *Maintain* $\widetilde{\text{dist}}_G^{h,i}(s,v)$ *and* $\widetilde{\text{dist}}_G^{h,i}(v,s)$ *for every node* v *in a weighted directed graph* G *undergoing edge deletions and edge weight increases in total time* $O(mh/\epsilon + \Delta_G)$*, where* Δ_G *is the number of updates in* G.

- *Maintain* $\widetilde{\text{dist}}_G^{h}(s,v)$ *and* $\widetilde{\text{dist}}_G^{h}(v,s)$ *in a weighted directed graph* G *undergoing edge deletions and edge weight increases in total time* $\widetilde{O}((mh\log W)/\epsilon + \Delta_G)$*, where* Δ_G *is the number of updates in* G.

Note that in the following we will sometimes use the rounded distance although the argument would also go through with any $(1+\epsilon)$-approximate distance estimate. This may first seem quite technical but consistently using the rounded distance actually simplifies our arguments a bit.

The second concept that we need is the path union graph. Intuitively, the path union graph of an unweighted graph G is a subgraph of G containing all paths from a given node x to a given node y that have at most h edges (for some given h). We define it as follows.

Definition 4.10. *For all nodes* x *and* y *in an unweighted directed graph* G *and every integer* $h \geq 1$*, we define* $\mathcal{P}(x,y,h,G)$ *to be the subgraph of* G *induced by the nodes* v *of* G *for which* $\text{dist}_G(x,v) + \text{dist}_G(v,y) \leq h$.

It is easy to see that the definition we gave is equivalent to the property we initially desired for path union graphs.

Proposition 4.11. *The path union graph* $\mathcal{P}(x,y,h,G)$ *is the subgraph of* G *induced by the nodes* v *of* G *that lie on a path from* x *to* y *with at most* h *edges.*

As the path union graph contains all paths from x to y of length at most h, it in particular contains the shortest of these paths. Therefore we have $\text{dist}_G^h(x,y) = \text{dist}^h_{\mathcal{P}(x,y,h,G)}(x,y)$. In our algorithms we will be able to bound the number of edges in the path union graphs we use, which makes them very useful. It is also easy to see that the path union graph is, in a weak sense, preserved under deletions in the original graph.

Proposition 4.12. *Let* $\mathcal{Q}$ *be the path union graph* $\mathcal{P}(x,y,h,G)$ *of an unweighted directed graph* G*. Let* G' *be the result of deleting edges from* G *and let* $\mathcal{Q}'$ *be the result of deleting the same edges from* $\mathcal{Q}$ *(if they are contained in* $\mathcal{Q}$*). Then* $\mathcal{Q}'$ *contains the path union* $\mathcal{P}(x,y,h,G')$.

We also want to extend the concept of path union graphs to weighted graphs. We would like the path union graph to contain all paths from x to y with at most h edges and weight at most X (for a given upper bound X). However, we will use a slightly different definition that is more suitable for our context in which we cannot guarantee exact distances anyway.

Definition 4.13. *For all nodes* x *and* y *in a weighted directed graph* G *and all integers* $h \geq 1$ *and* $0 \leq i \leq \lceil \log_{1+\epsilon}(nW) \rceil$*, we define* $\mathcal{P}(x,y,h,i,G)$ *to be the subgraph of* G *induced by the nodes* v *of* G *for which* $\widetilde{\text{dist}}_G^{h,i}(x,v) + \widetilde{\text{dist}}_G^{h,i}(v,y) \leq (1+\epsilon)^{i+2}$.

Lemma 4.14. *If* $\text{dist}_G^h(x,y) \geq (1+\epsilon)^i$*, then the path union graph* $\mathcal{P}(x,y,h,i,G)$ *contains every path from* x *to* y *in* G *that has at most* h *edges and weight at most* $(1+\epsilon)^{i+1}$.

Proof. Consider a path π from x to y in G that has at most h edges and weight at most $(1+\epsilon)^{i+1}$. We argue that every node on π is contained in $\mathcal{P}(x,y,h,i,G)$. Let v be a node on π and let π_1 and π_2 denote the subpaths of π from x to v and from v to y, respectively. Set $\alpha = \epsilon(1+\epsilon)^i/h$ and let G' be the graph that has the same nodes and edges as G and the weight of every edge (u,v) in G' is set to $w_{G'}(u,v) = \lceil w_G(u,v)/\alpha \rceil \cdot \alpha$. Remember that for all nodes x' and y' we have $\widetilde{\text{dist}}_G^{h,i}(x',y') = \text{dist}_{G'}(x',y')$ by Definition 4.7.

Since π is a path in G with at most h edges we have $w_G(\pi) \geq \text{dist}_G^h(x,y)$. From the assumption $\text{dist}_G^h(x,y) \geq (1+\epsilon)^i$ it therefore follows that $w_G(\pi) \geq (1+\epsilon)^i$. Note further that the rounding guarantees that $w_{G'}(u,v) \leq w_G(u,v) + \alpha$ for every edge (u,v) in G. As π has at most h edges, we get

$$\begin{aligned} w_{G'}(\pi) = \sum_{(u,v)\in\pi} w_{G'}(u,v) &\leq \sum_{(u,v)\in\pi} (w_G(u,v) + \alpha) \\ &= \sum_{(u,v)\in\pi} w_G(u,v) + \sum_{(u,v)\in\pi} \alpha \\ &\leq w_G(\pi) + h\alpha \\ &= w_G(\pi) + \epsilon(1+\epsilon)^i \\ &\leq w_G(\pi) + \epsilon w_G(\pi) = (1+\epsilon) w_G(\pi)\,. \end{aligned}$$

We also clearly have $\widetilde{\text{dist}}_G^{h,i}(x,v) = \text{dist}_{G'}(x,v) \leq w_{G'}(\pi_1)$ and $\widetilde{\text{dist}}_G^{h,i}(v,y) = \text{dist}_{G'}(v,y) \leq w_{G'}(\pi_2)$. Thus, we get

$$\widetilde{\text{dist}}_G^{h,i}(x,v) + \widetilde{\text{dist}}_G^{h,i}(v,y) \leq w_{G'}(\pi_1) + w_{G'}(\pi_2) = w_{G'}(\pi) \leq (1+\epsilon) w_G(\pi) \leq (1+\epsilon)(1+\epsilon)^{i+1} = (1+\epsilon)^{i+2}\,.$$

This means that the node v is contained in $\mathcal{P}(x,y,h,i,G)$ by the definition of path union graphs.

Since v was an arbitrary node of the path π, we conclude that all nodes of π are contained in $\mathcal{P}(x,y,h,i,G)$. As all nodes of π are contained in $\mathcal{P}(x,y,h,i,G)$, also the edges connecting them are contained in $\mathcal{P}(x,y,h,i,G)$. Thus, $\mathcal{P}(x,y,h,i,G)$ contains the path π. □

Lemma 4.15. *The path union graph $\mathcal{P}(x,y,h,i,G)$ can be computed in time $\widetilde{O}(m)$.*

Proof. By Lemma 4.9 we can compute $\widetilde{\text{dist}}_G^{h,i}(x,v)$ and $\widetilde{\text{dist}}_G^{h,i}(v,y)$ for every node v in time $\widetilde{O}(m)$. Checking the defining inequality for every node takes time $O(n)$, which is dominated by $\widetilde{O}(m)$. □

Our algorithms heavily use randomization. It is well-known, and exploited by many other algorithms for dynamic (approximate) 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 whp. To the best of our knowledge, the first use of this technique in graph algorithms goes back to Ullman and Yannakakis [22].

Lemma 4.16. *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 $p = (a \ln kt)/q$, where a is a constant. 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

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

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

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

□

We now sketch one example of how we intend to use Lemma 4.16 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 every deletion): if $\text{dist}_G(x, y) \geq q$, then there is a shortest path from x to y that contains a node in U. To apply Lemma 4.16 we do the following. 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 every deletion) and every pair of nodes x and y such that $\text{dist}_G(x, y) \geq q$, 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 4.16.

Note that in general the sampling probability we need depends on Δ_G, the number of edge deletions and edge weight increases. However, in weighted graphs, Δ_G can only be bounded by $O(mW)$. Thus, we would have to sample $\Omega(\log(mW))$ nodes in expectation. We will now show that we can assume without loss of generality that Δ_G is only $O(m \log_{1+\epsilon} W)$, which reduces the number of nodes that have to be sampled. This assumption is also useful for a second reason. As mentioned above, the total update time of approximate decremental stSP and SSSP data structures for weighted graphs are $\Omega(\Delta_G)$. In our algorithms we will sometimes internally use k such decremental data structures (for some suitable choice of k) and the running time of $\Omega(\Delta_G)$ then has to be multiplied by k. This multiplicative factor of k causes, however, no problem when $\Delta_G = O(m \log_{1+\epsilon} W)$ as the term $km \log_{1+\epsilon} W$ is usually dominated by other running time aspects of our algorithms.

Lemma 4.17. *Given a weighted directed graph G undergoing edge deletions and edge weight increases, we can, in constant time per update in G, maintain a weighted directed graph G' undergoing edge deletions and edge weight increases such that*

- *the number of updates in G' is at most $\lceil \log_{1+\epsilon} W \rceil$ per edge (which implies $\Delta_{G'} = O(m \log_{1+\epsilon} W)$) and*
- $\text{dist}_G(x, y) \leq \text{dist}_{G'}(x, y) \leq (1+\epsilon)\,\text{dist}_G(x, y)$ *for all nodes x and y.*

Proof. We define G' to be the graph where each edge weight of G is rounded up to the nearest power of $(1+\epsilon)$. For every edge (u, v), we have $(1+\epsilon)^i \leq w(u, v) < (1+\epsilon)^{i+1}$ for some i. By setting $w'(u, v) = \lfloor (1+\epsilon)^{i+1} \rfloor$ we get $w(u, v) \leq w'(u, v) \leq (1+\epsilon) w(u, v)$. This means that the

weight of every path P in G is $(1+\epsilon)$-approximated by the weight of P in G'. As this is also true for a shortest path, we get $\text{dist}_G(x,y) \leq \text{dist}_{G'}(x,y) \leq (1+\epsilon)\,\text{dist}_G(x,y)$ for all nodes x and y. With every edge weight increase of an edge (u,v) in G we can in constant time compute the weight $w'(u,v)$ and check whether it has increased. The number of different edge weights in G is $\lceil \log_{1+\epsilon} W \rceil$ and therefore the total number of edge deletions and edge weight increases in G' is $O(m \log_{1+\epsilon} W)$. Thus, G' is the desired graph. □

5 $(1+\epsilon)$-Approximate stSP for Dense Weighted Graphs

In the following we provide an algorithm for maintaining $(1+\epsilon)$-approximate stSP that is tailored to dense graphs.

Theorem 5.1. *There is a $(1+\epsilon)$-approximate stSP data structure for weighted directed graphs undergoing edge deletions and edge weight increases with constant query time and a total update time of $\widetilde{O}((m^{5/7}n^{10/7}\log W)/\epsilon^2 + \Delta_G)$.*

5.1 Main Algorithm

Our algorithm is guided by the idea of maintaining the distance between s and t in a certain smaller graph. We call this graph *center graph* as it contains a set of distinguished nodes, called *centers*, and edges between them that correspond to their distances in the original graph. In the following we define an approximate version of this concept.

Definition 5.2. *Given a graph G, a set of centers C, a hop count h, and an approximation parameter ϵ, a $(1+\epsilon)$-approximate center graph $\mathcal{C}(C,h,G)$ is a graph whose set of nodes is C and whose edges satisfy the following conditions. For every pair of centers x and y there is a directed edge (x,y) in $\mathcal{C}(C,h,G)$ if and only if there is a path from x to y using at most h hops. The weight $w_{\mathcal{C}(C,h,G)}(x,y)$ of such an edge (x,y) fulfills the inequality $\text{dist}_G(x,y) \leq w_{\mathcal{C}(C,h,G)}(x,y) \leq (1+\epsilon)\,\text{dist}^h_G(x,y)$.*

It can be shown, by fairly standard techniques, that the center graph approximates distances in the original graph if we randomly choose a sufficiently large number of centers.

Lemma 5.3. *Let C be a set of centers containing a set of nodes sampled uniformly at random with probability $(a \ln(n\Delta_G))/h$ for a large enough constant a. Then a $(1+\epsilon)$-approximate center graph $\mathcal{C}(C,h,G)$ provides $(1+\epsilon)$-approximate distances between centers whp, i.e., $\text{dist}_G(x,y) \leq \text{dist}_{\mathcal{C}(C,h,G)}(x,y) \leq (1+\epsilon)\,\text{dist}_G(x,y)$ for all pairs of centers $x,y \in C$.*

Proof. We first prove the first inequality $\text{dist}_G(x,y) \leq \text{dist}_{\mathcal{C}(C,h,G)}(x,y)$. If $\text{dist}_{\mathcal{C}(C,h,G)}(x,y) = \infty$, then the inequality is trivially true. If $\text{dist}_{\mathcal{C}(C,h,G)}(x,y) < \infty$, we prove the inequality by induction on $\text{dist}_{\mathcal{C}(C,h,G)}(x,y)$. The base case is $\text{dist}_{\mathcal{C}(C,h,G)}(x,y) = 0$, for which the inequality is trivially true. For the induction step, let π be a shortest path from x to y in $\mathcal{C}(C,h,G)$ and let (x,z) be the first edge on this shortest path. Note that $\text{dist}_{\mathcal{C}(C,h,G)}(z,y) \leq \text{dist}_{\mathcal{C}(C,h,G)}(x,y) - 1$ because $w_{\mathcal{C}(C,h,G)}(x,z) \geq \text{dist}_G(x,z) \geq 1$ as the edge weights in G are non-negative integers. Therefore we may apply the induction hypothesis and get that $\text{dist}_G(z,y) \leq \text{dist}_{\mathcal{C}(C,h,G)}(z,y)$. We now have

$$\text{dist}_G(x,y) \leq \text{dist}_G(x,z) + \text{dist}_G(z,y) \leq w_{\mathcal{C}(C,h,G)}(x,z) + \text{dist}_{\mathcal{C}(C,h,G)}(z,y) = \text{dist}_{\mathcal{C}(C,h,G)}(x,y)$$

as desired.

We now prove the second inequality $\text{dist}_{\mathcal{C}(C,h,G)}(x,y) \leq (1+\epsilon)\,\text{dist}_G(x,y)$. It is trivially true if $\text{dist}_G(x,y) = \infty$. If $\text{dist}_G(x,y) < \infty$, we prove the inequality by induction on $\text{dist}_G(x,y)$. The base case is $\text{dist}_G(x,y) = 0$, for which the inequality is trivially true. For the induction step, let π be a shortest path from x to y in G. If π contains at most h edges, then we have $\text{dist}_G(x,y) = \text{dist}^h_G(x,y)$. Furthermore we know that $\mathcal{C}(C,h,G)$ contains an edge (x,y) of weight $w_{\mathcal{C}(C,h,G)}(x,y) \leq (1+\epsilon)\,\text{dist}^h_G(x,y)$. As $\text{dist}_{\mathcal{C}(C,h,G)}(x,y) \leq w_{\mathcal{C}(C,h,G)}(x,y)$, the desired inequality follows.

Consider now the case that π has more than h edges. Then we know by Lemma 4.16 that whp there is some center $z \neq x$ within the first h nodes of π. Note that $\mathcal{C}(C,h,G)$ contains an edge (x,z) of weight $w_{\mathcal{C}(C,h,G)}(x,z) \leq (1+\epsilon)\,\text{dist}^h_G(x,z)$. Furthermore, we have $\text{dist}_G(z,y) \leq \text{dist}_G(x,y) - 1$ because the edge weights in G are non-negative integers. Therefore we may apply the induction hypothesis, which gives $\text{dist}_{\mathcal{C}(C,h,G)}(z,y) \leq (1+\epsilon)\,\text{dist}_G(z,y)$. In total, we get

$$\begin{aligned}\text{dist}_{\mathcal{C}(C,h,G)}(x,y) &\leq \text{dist}_{\mathcal{C}(C,h,G)}(x,z) + \text{dist}_{\mathcal{C}(C,h,G)}(z,y)\\ &\leq w_{\mathcal{C}(C,h,G)}(x,z) + \text{dist}_{\mathcal{C}(C,h,G)}(z,y)\\ &\leq (1+\epsilon)\,\text{dist}_G(x,z) + (1+\epsilon)\,\text{dist}_G(z,y) = (1+\epsilon)\,\text{dist}_G(x,y)\,.\end{aligned}$$ □

Assuming that we can maintain the edges of an approximate center graph, our algorithm now is straightforward. At the initialization we sample a set of nodes as described in Lemma 5.3. The sampled nodes, together with the source node s and the sink node t will be the set of centers used by the algorithm. We then maintain the edges of an approximate center graph and in this center graph we maintain the approximate distance between s and t.

Lemma 5.4. *Let C be a set of centers consisting of s, t, and a set of nodes sampled uniformly at random with probability $(a \ln(n\Delta_G))/h$ for a large enough constant a. Given a data structure for maintaining the edges of a $(1+\epsilon)$-approximate center graph $\mathcal{C}(C,h,G)$, there is a $(1+\epsilon)$-approximate* stSP *data structure with constant query time and a total update time of $\widetilde{O}((n^3/h^3 \log_{1+\epsilon} W)/\epsilon + \Delta_{\mathcal{C}(C,h,G)})$, where $\Delta_{\mathcal{C}(C,h,G)}$ is the number of updates in $\mathcal{C}(C,h,G)$.*

Proof. Setting $c = n/h$, the number of centers $|C|$ is $\widetilde{O}(c)$ in expectation. Therefore the approximate center graph has $\widetilde{O}(c)$ nodes and $\widetilde{O}(c^2)$ edges in expectation. Thus, maintaining $\widetilde{\text{dist}}^{|C|}_{\mathcal{C}(C,h,G)}(s,t)$ takes time $\widetilde{O}((c^3 \log_{1+\epsilon} W)/\epsilon + \Delta_G)$ by Lemma 4.9.

By Lemma 5.3 we know that $\text{dist}_G(s,t) \leq \text{dist}_{\mathcal{C}(C,h,G)}(s,t) \leq (1+\epsilon)\,\text{dist}_G(s,t)$. Furthermore, as the center graph has $|C|$ nodes, we have $\text{dist}^{|C|}_{\mathcal{C}(C,h,G)}(s,t) = \text{dist}_{\mathcal{C}(C,h,G)}(s,t)$. Now observe the following:

$$\begin{aligned}\text{dist}_G(s,t) \leq \text{dist}_{\mathcal{C}(C,h,G)}(s,t) &\leq \widetilde{\text{dist}}^{|C|}_{\mathcal{C}(C,h,G)}(s,t)\\ &\leq (1+\epsilon)\,\text{dist}^{|C|}_{\mathcal{C}(C,h,G)}(s,t)\\ &= (1+\epsilon)\,\text{dist}_{\mathcal{C}(C,h,G)}(s,t)\\ &\leq (1+\epsilon)^2\,\text{dist}_G(s,t) \leq (1+3\epsilon)\,\text{dist}_G(s,t)\,.\end{aligned}$$

By running the algorithm with $\epsilon' := \epsilon/3$, we can use $\delta(s,t) := \widetilde{\text{dist}}^{|C|}_{\mathcal{C}(C,h,G)}(s,t)$ as a $(1+\epsilon)$-approximate estimate of $\text{dist}_G(s,t)$. □

The main difficulty in our algorithm is how to actually maintain the approximate center graph, i.e., the distance estimates between centers. Doing this in time $O(mn)$ is easy using standard

techniques and we show in the following how to do it *even more efficiently*. In the rest of this section we provide an auxiliary algorithm for maintaining the distance estimates and analyze the overall running time.

5.2 Auxiliary Algorithm

Our auxiliary algorithm maintains, for a set of centers, the pairwise distance between all centers restricted to a given number of hops.

Theorem 5.5. *Given a weighted directed graph undergoing edge deletions and edge weight increases, an approximation parameter $0 < \epsilon \leq 1$, a hop count $h \geq 1$, and a set of centers of size c, there is a data structure that maintains, for every pair of centers x and y, a distance estimate $\delta(x,y)$ such that $\operatorname{dist}_G(x,y) \leq \delta(x,y) \leq (1+3\epsilon)\operatorname{dist}^h_G(x,y)$. Its total update time is $\widetilde{O}((c^2m + hc^{2/3}n^{2/3}m^{2/3})(\log W)/\epsilon^2)$.*

To achieve this goal we will actually design the a data structure that maintains, for every pair of centers x and y, an index i such that $\operatorname{dist}^h_G(x,y) \geq (1+\epsilon)^i$ and $\operatorname{dist}_G(x,y) \leq (1+\epsilon)^{i+2}$. Observe that

$$\operatorname{dist}_G(x,y) \leq (1+\epsilon)^{i+2} = (1+\epsilon)^2(1+\epsilon)^i \leq (1+\epsilon)^2 \operatorname{dist}^h_G(x,y) \leq (1+3\epsilon)\operatorname{dist}^h_G(x,y)\,.$$

Thus, $\delta(x,y) := (1+\epsilon)^{i+2}$ is the desired distance estimate.

In the following we provide the auxiliary algorithm needed to maintain this data structure and show its correctness and running time guarantee. Initially, we sample each node with probability $(ab\ln(n\Delta_G))/n$ (for a large enough constant a) to obtain a set $B = \{b_1, \ldots, b_{|B|}\}$ of nodes called *hubs*. We fix an arbitrary order on the hubs. For every hub $v \in B$ we use the data structure of Lemma 4.9 to maintain $\widetilde{\operatorname{dist}}^h_G(\cdot, v)$ and $\widetilde{\operatorname{dist}}^h_G(v, \cdot)$, which gives $(1+\epsilon)$-approximate distance estimates of the distances to and from v up to h hops, respectively.

We now show how to maintain an index $i(x,y)$ for every pair of centers x and y such that $\operatorname{dist}^h_G(x,y) \geq (1+\epsilon)^{i(x,y)}$ and $\operatorname{dist}_G(x,y) \leq (1+\epsilon)^{i(x,y)+2}$. Initially we use $i(x,y) = 0$. At the initialization and after every update in the graph, we do the following to ensure that $i(x,y)$ is the correct index. As long as there is a hub v such that $\widetilde{\operatorname{dist}}^h_G(x,v) + \widetilde{\operatorname{dist}}^h_G(v,y) \leq (1+\epsilon)^{i(x,y)+2}$ we leave $i(x,y)$ unchanged. If there is no hub v (anymore) such that $\widetilde{\operatorname{dist}}^h_G(x,v) + \widetilde{\operatorname{dist}}^h_G(v,y) \leq (1+\epsilon)^{i(x,y)+2}$, we first initialize a new graph $\mathcal{Q}(x,y)$ which initially has the same nodes and edges as the path union graph $\mathcal{P}(x,y,h,i(x,y),G)$ (see Definition 4.13). We then use the data structure of Lemma 4.9 to maintain $\widetilde{\operatorname{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x,y)$ from now on. Whenever $\widetilde{\operatorname{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x,y)$ exceeds $(1+\epsilon)^{i(x,y)+2}$, we increase $i(x,y)$ by 1 and start again with finding a suitable hub. Note that, as soon as $\mathcal{Q}(x,y)$ has been created, we maintain $\mathcal{Q}(x,y)$ in the sense that we perform all graph updates of G also in $\mathcal{Q}(x,y)$.

We use a second index $j(x,y)$ that tells us which hub $b_{j(x,y)}$ currently ensures that $\widetilde{\operatorname{dist}}^h_G(x, b_{j(x,y)}) + \widetilde{\operatorname{dist}}^h_G(b_{j(x,y)}, y) \leq (1+\epsilon)^{i(x,y)+2}$. As soon as this inequality does not hold anymore for $b_{j(x,y)}$, we keep increasing $j(x,y)$ until we find a new hub $b_{j(x,y)}$ for which it holds. If we reach $j(x,y) = |B|+1$ we know that we are in the case that there is no suitable hub anymore and we start constructing the path union graph. Whenever we increase $i(x,y)$ we set $j(x,y)$ back to 1. The pseudocode of the algorithm we just described is given in Algorithm 1.

In the following we fix a pair of centers x and y. We will simply write i and j instead of $i(x,y)$ and $j(x,y)$, respectively. We first prove the correctness of the algorithm.

Lemma 5.6 (Correctness)**.** *When the algorithm has finished updating, we have $\operatorname{dist}^h_G(x,y) \geq (1+\epsilon)^i$ and $\operatorname{dist}_G(x,y) \leq (1+\epsilon)^{i+2}$ for the current index i.*

Algorithm 1: Auxiliary algorithm for maintaining approximate distance of centers

Determine set of hubs $B = \{b_1, \ldots, b_{|B|}\}$ and set of centers C by random sampling of nodes
For every hub $v \in B$ maintain $\widetilde{\text{dist}}^{h}_{G}(x, v)$ and $\widetilde{\text{dist}}^{h}_{G}(v, x)$ for every center x (see Lemma 4.9)
Set $i(x, y) = 0$ and $j(x, y) = 1$ for every pair of centers x and y ac call `update(`x, y`)`
After every deletion or edge weight increase, report update to internal data structures and call `update(`x, y`)` for every pair of centers x and y

`update(`x, y`)`
  if $j(x, y) < |B| + 1$ **then**
    while $j(x, y) \leq |B|$ **and** $\widetilde{\text{dist}}^{h}_{G}(x, b_{j(x,y)}) + \widetilde{\text{dist}}^{h}_{G}(b_{j(x,y)}, y) > (1 + \epsilon)^{i(x,y)+2}$ **do**
      $j(x, y) \leftarrow j(x, y) + 1$
    if $j(x, y) = |B| + 1$ **then**
      Initialize $\mathcal{Q}(x, y)$ as the path union graph $\mathcal{P}(x, y, h, i(x, y), G)$
      Start maintaining $\mathcal{Q}(x, y)$ by making the same future edge deletions and edge weight increases as in G
      Start maintaining $\widetilde{\text{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x, y)$ (see Lemma 4.9)
  else
    if $\widetilde{\text{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x, y) > (1 + \epsilon)^{i(x,y)+2}$ **then**
      Stop maintaining $\mathcal{Q}(x, y)$ and $\widetilde{\text{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x, y)$
      if $i(x, y) < \log_{1+\epsilon}(nW)$ **then**
        $i(x, y) \leftarrow i(x, y) + 1$
        $j(x, y) \leftarrow 1$
        `update(`x, y`)`
      else
        $i(x, y) \leftarrow \infty$

Proof. We first show that, every time the algorithm has finished updating i and j, we have $\text{dist}^h_G(x,y) \geq (1+\epsilon)^i$ and $\text{dist}_G(x,y) \leq (1+\epsilon)^{i+2}$, for the current value of i in the algorithm. Initially, i is set to 0 and it is clear that $\text{dist}^h_G(x,y) \geq (1+\epsilon)^0$. We now show that every time the algorithm increases i in Line 17, the value $\text{dist}^h_G(x,y)$ has increased to the next multiple of $1+\epsilon$. In order to show this we need the fact that $\mathcal{Q}(x,y)$, even though it might not follow the definition of a path union graph anymore, still contains all necessary paths to give the correct answer.

Claim 5.7. *For the current value of i, the graph $\mathcal{Q}(x,y)$ contains every path from x to y in G that has at most h edges and weight at most $(1+\epsilon)^{i+1}$.*

Proof. The claim is true when we initialize $\mathcal{Q}(x,y)$ to be $\mathcal{P}(x,y,h,i,G)$ by Lemma 4.14. Now consider a path π from x to y in G that has at most h edges and has weight at most $(1+\epsilon)^{i+1}$. When we initialized $\mathcal{Q}(x,y)$, the weight of π was not more than its current weight. Therefore π was contained in $\mathcal{Q}(x,y)$ (again by Lemma 4.14). As $\mathcal{Q}(x,y)$ undergoes exactly the same deletions and weight increases as G, we get that π is contained in $\mathcal{Q}(x,y)$. □

Claim 5.8. *If $\text{dist}^h_G(x,y) \geq (1+\epsilon)^i$ and $\widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y) \geq (1+\epsilon)^{i+2}$, then $\text{dist}^h_G(x,y) \geq (1+\epsilon)^{i+1}$.*

Proof. Suppose for the sake of contradiction that $\text{dist}^h_G(x,y) < (1+\epsilon)^{i+1}$. Thus, the shortest path from x to y in G with at most h hops has weight at most $(1+\epsilon)^{i+1}$. By Claim 5.7 this path is contained in $\mathcal{Q}(x,y)$ and thus $\text{dist}^h_G(x,y) = \text{dist}^h_{\mathcal{Q}(x,y)}(x,y)$. Since $\text{dist}^h_G(x,y) \geq (1+\epsilon)^i$, we have $\widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y) \leq (1+\epsilon)\,\text{dist}^h_{\mathcal{Q}(x,y)}(x,y)$ by Lemma 4.8. Putting everything together, we get

$$(1+\epsilon)^{i+2} \leq \widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y) \leq (1+\epsilon)\,\text{dist}^h_{\mathcal{Q}(x,y)}(x,y) = (1+\epsilon)\,\text{dist}^h_G(x,y) < (1+\epsilon)^{i+2}$$

which gives a contradiction. □

It follows that, when the algorithm has finished updating, we have $\text{dist}^h_G(x,y) \geq (1+\epsilon)^i$. We now show that then also the second bound holds, i.e., that $\text{dist}_G(x,y) \leq (1+\epsilon)^{i+2}$.

Claim 5.9. *When the algorithm has finished updating, we have $\text{dist}_G(x,y) \leq (1+\epsilon)^{i+2}$.*

Proof. If $j \leq |B|$, then we know by the rules of the algorithm that $\widetilde{\text{dist}}^h_G(x,b_j) + \widetilde{\text{dist}}^h_G(b_j,y) \leq (1+\epsilon)^{i+2}$. By the triangle inequality we have $\text{dist}_G(x,y) \leq \text{dist}_G(x,b_j) + \text{dist}_G(b_j,x)$ and by Lemma 4.8 we have $\text{dist}_G(x,b_j) \leq \widetilde{\text{dist}}^h_G(x,b_j)$ and $\text{dist}_G(b_j,y) \leq \widetilde{\text{dist}}^h_G(b_j,y)$. Thus, we get

$$\text{dist}_G(x,y) \leq \text{dist}_G(x,b_j) + \text{dist}_G(b_j,x) \leq \widetilde{\text{dist}}^h_G(x,b_j) + \widetilde{\text{dist}}^h_G(b_j,y) \leq (1+\epsilon)^{i+2}\,.$$

If $j = |B|+1$, then we know by the rules of the algorithm that $\widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y) \leq (1+\epsilon)^{i+2}$. As $\mathcal{Q}(x,y)$ is a subgraph of G, we have $\text{dist}_G(x,y) \leq \text{dist}_{\mathcal{Q}(x,y)}(x,y)$ and by Lemma 4.8 we know that $\text{dist}_{\mathcal{Q}(x,y)}(x,y) \leq \widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y)$. Thus, we get

$$\text{dist}_G(x,y) \leq \text{dist}_{\mathcal{Q}(x,y)}(x,y) \leq \widetilde{\text{dist}}^{h,i}_{\mathcal{Q}(x,y)}(x,y) \leq (1+\epsilon)^{i+2}\,.$$ □

This finishes the correctness proof. □

We now analyze the running time of Algorithm 1. For this purpose we bound the size of the graph $\mathcal{Q}(x,y)$. *The sparsity of $\mathcal{Q}(x,y)$ is the key to the efficiency of our algorithm.* We only have to bound the size of $\mathcal{Q}(x,y)$ when we initialize $\mathcal{Q}(x,y)$ to be $\mathcal{P}(x,y,h,i,G)$ because the size of $\mathcal{Q}(x,y)$ never increases during the algorithm.

Lemma 5.10. *After setting j to $|B|+1$, $\mathcal{P}(x,y,h,i,G)$ has at most n/b nodes with high probability.*

Proof. If $j = |B|+1$, then we have $\widetilde{\text{dist}}^{h}_{G}(x,v)+\widetilde{\text{dist}}^{h}_{G}(v,y) > (1+\epsilon)^{i+2}$ for every hub $v \in B$. Suppose that $\mathcal{P}(x,y,h,i,G)$ has more than n/b nodes. Then, by Lemma 4.16 and the random sampling of the nodes in B, $\mathcal{P}(x,y,h,i,G)$ contains at least one hub $v \in B$ with high probability. By the definition of $\mathcal{P}(x,y,h,i,G)$ (Definition 4.13), this means that $\widetilde{\text{dist}}^{h,i}_{G}(x,v) + \widetilde{\text{dist}}^{h,i}_{G}(v,y) \leq (1+\epsilon)^{i+2}$. We also have $\widetilde{\text{dist}}^{h}_{G}(x,v) \leq \widetilde{\text{dist}}^{h,i}_{G}(x,v)$ and $\widetilde{\text{dist}}^{h}_{G}(v,y) \leq \widetilde{\text{dist}}^{h,i}_{G}(v,y)$ by definition. Putting everything together, this gives

$$\widetilde{\text{dist}}^{h}_{G}(x,v) + \widetilde{\text{dist}}^{h}_{G}(v,y) \leq \widetilde{\text{dist}}^{h,i}_{G}(x,v) + \widetilde{\text{dist}}^{h,i}_{G}(v,y) \leq (1+\epsilon)^{i+2}$$

which contradicts the fact that $\widetilde{\text{dist}}^{h}_{G}(x,v) + \widetilde{\text{dist}}^{h}_{G}(v,y) > (1+\epsilon)^{i+2}$. □

We now bound the running time of the auxiliary algorithm as follows.

Lemma 5.11. *The total update time of the auxiliary algorithm is $\widetilde{O}((c^2m+hc^{2/3}n^{2/3}m^{2/3})(\log W)/\epsilon^2)$.*

Proof. Recall that Δ_G denotes the number of edge deletions and edge weight increases in G and we may assume that $\Delta_G = O(m \log_{1+\epsilon} W)$ by Lemma 4.17. Note that the number of hubs is $\widetilde{O}(b)$ in expectation. We analyze the costs of the algorithm as follows:

- For every hub v, we maintain $\widetilde{\text{dist}}^{h}_{G}(x,v)$ and $\widetilde{\text{dist}}^{h}_{G}(v,y)$ for all nodes x and y. As this takes time $\widetilde{O}((mh \log_{1+\epsilon} W)/\epsilon + \Delta_G)$ for every hub by Lemma 4.9, these costs are bounded by $\widetilde{O}((bmh \log_{1+\epsilon} W)/\epsilon + b\Delta_G)$.
- We have to construct the path union graph $\mathcal{P}(x,y,h,i(x,y),G)$ at most once for every pair of centers x and y and every index $i(x,y)$. As constructing a single path union graph takes time $\widetilde{O}(m)$ by Lemma 4.9, these costs are bounded by $\widetilde{O}(c^2 m \log_{1+\epsilon} W)$.
- In the worst case, we have to maintain $\widetilde{\text{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x,y)$ for every pair of centers x and y and every index $i(x,y)$. By Lemma 5.10 each graph $\mathcal{Q}(x,y)$ considered by our algorithm has at most n/b nodes with high probability and thus at most n^2/b^2 edges. Maintaining $\widetilde{\text{dist}}^{h,i(x,y)}_{\mathcal{Q}(x,y)}(x,y)$ takes time $\widetilde{O}((n^2h)/(b^2\epsilon) + \Delta_{\mathcal{Q}(x,y)})$ by Lemma 4.9, where $\Delta_{\mathcal{Q}(x,y)}$ is the number of updates in $\mathcal{Q}(x,y)$. Note that $\sum_{i(x,y)} \Delta_{\mathcal{Q}(x,y)} \leq \Delta_G$ as the algorithm only considers one graph $\mathcal{Q}(x,y)$ at a time. Thus, these costs are bounded by $\widetilde{O}((c^2n^2h \log_{1+\epsilon} W)/(b^2\epsilon) + c^2\Delta_G)$.
- The remaining costs of the auxiliary algorithm can be bounded as follows. The algorithm incurs a cost of $O(1)$ for every update in the graph and for every update of the indices $i(x,y)$ or $j(x,y)$ of any pair of centers x and y. As there are $\widetilde{O}(b)$ different values of $i(x,y)$ and $\widetilde{O}(\log_{1+\epsilon} W)$ different values of $j(x,y)$, these costs are bounded by $\widetilde{O}(bc^2 \log_{1+\epsilon} W + \Delta_G)$.

Note that the term bc^2 is dominated by c^2m. By balancing the terms bmh and c^2hn^2/b^2 we obtain $b = c^{2/3}n^{2/3}/m^{1/3}$. With this choice of parameters, we get a running time of $\widetilde{O}((c^2m + hc^{2/3}n^{2/3}m^{2/3})(\log W)/\epsilon^2)$. Here we use the facts that by Lemma 4.17 we may assume that $\Delta_G = O(m \log_{1+\epsilon} W)$ and that $1/\log(1+\epsilon) = O(1/\epsilon)$. □

This finishes the proof of Theorem 5.5. Now the algorithm of Theorem 5.1 simply follows from Theorem 5.5 and Lemma 5.4.

Proof of Theorem 5.1. We set $h = n^{2/7}m^{1/7}$ and $c = n/h$. Let C be a set of centers of size $\widetilde{O}(c)$ consisting of the nodes s and t and a set of nodes sampled uniformly at random with probability $(a \ln(n\Delta_G))/h$ for a large enough constant a. We maintain an approximate center graph $\mathcal{C}(C, h, G)$ by maintaining approximate distances between the centers in time $\widetilde{O}((c^2m + hc^{2/3}n^{2/3}m^{2/3})(\log W)/\epsilon^2)$ using Theorem 5.5. By our choices of h and c this is $\widetilde{O}((m^{5/7}n^{10/7}\log W)/\epsilon^2)$. In the center graph we maintain the approximate distance between s and t (see Lemma 5.4). The total update time of this step is $\widetilde{O}((c^3 \log_{1+\epsilon} W)/\epsilon + \Delta_{\mathcal{C}(C,h,G)})$, where $\Delta_{\mathcal{C}(C,h,G)}$ is the number of edge deletions and edge weight increases of the approximate center graph $\mathcal{C}(C, h, G)$. As $c^3 \leq c^2m$, the first term is dominated by $\widetilde{O}((c^2m \log W)/\epsilon^2)$. As $\mathcal{C}(C, h, G)$ has $\widetilde{O}(c^2)$ edges, each update in G changes at most $\widetilde{O}(c^2)$ edges in G. We may assume that the number of updates in G is $O(m \log_{1+\epsilon} W)$ by Lemma 4.17 and therefore the second term is also dominated by $\widetilde{O}((c^2m \log W)/\epsilon^2)$. □

When we formulated Lemma 5.4 we focused on $(1+\epsilon)$-approximate stSP, but in fact our result can be generalized a bit. Instead of maintaining the approximate distance between only one source and one sink, we can as well maintain all pairwise approximate distances between a set of k nodes.

Corollary 5.12. *Given a weighted directed graph undergoing edge deletions and edge weight increases, an approximation parameter $0 < \epsilon \leq 1$, and set of nodes S of size k, we can, for all nodes x and y in S, maintain a distance estimate $\delta(x, y)$ such that $\operatorname{dist}(x, y) \leq \delta(x, y) \leq (1+\epsilon)\operatorname{dist}(x, y)$. This data structure has constant query time and a total update time of $\widetilde{O}((k^2m + m^{5/7}n^{10/7}\log W)/\epsilon^2)$.*

Proof sketch. Set $h = n^{2/7}m^{1/7}$ and $c = k + n/h$. Let C be a set of centers of size $\widetilde{O}(c)$ consisting of the nodes in S and a set of nodes sampled uniformly at random with probability $(a \ln(n\Delta_G))/h$ for a large enough constant a. We maintain an approximate center graph $\mathcal{C}(C, h, G)$ in time $\widetilde{O}((c^2m + hc^{2/3}n^{2/3}m^{2/3})(\log W)/\epsilon^2)$ using Theorem 5.5. In the center graph we maintain $(1+\epsilon)$-approximate distances between *all* nodes using Bernstein's decremental approximate all-pairs shortest paths data structure [3]. This takes time $\widetilde{O}((c^3 \log W/\epsilon + \Delta_{\mathcal{C}(C,h,G)})$, where $\Delta_{\mathcal{C}(C,h,G)}$ is the number of edge deletions and edge weight increases of the approximate center graph $\mathcal{C}(C, h, G)$. Since $c^3 \leq c^2m$ and since we may assume that the number of updates in G is $O(m \log_{1+\epsilon} W)$ by Lemma 4.17, both terms in this running time are dominated by $\widetilde{O}((c^2m \log W)/\epsilon^2)$.

By our choice of h we have $hc^{2/3}n^{2/3}m^{2/3} = c^{2/3}n^{20/21}m^{17/21}$. Note that $c^{2/3}n^{20/21}m^{17/21} \leq c^2m$ if and only if $c \geq n^{5/7}/m^{1/7} = n/h$. Thus, if $k \geq n/h$, then c is $O(n/h)$ and c^2m is $O(c^{2/3}n^{20/21}m^{17/21}) = O(m^{5/7}n^{10/7})$. If $k \geq n/h$, then $c \geq n/h$ and $c^{2/3}n^{20/21}m^{17/21} \leq c^2m \leq k^2m$. Thus, the total update time is $\widetilde{O}((k^2m + m^{5/7}n^{10/7}\log W)/\epsilon^2)$ as desired. □

5.3 Algorithm for Graphs of Medium Density

We now explain how to break the "$O(mn)$-barrier" in graphs of medium density. In the algorithm of Theorem 5.1 for dense graphs we have used a "rounded-weights shortest paths tree" to maintain the approximate distance from s to t in the path union graph $\mathcal{Q}(x, y)$. But in fact, we could use any $(1+\epsilon)$-approximate stSP data structure running on $\mathcal{Q}(x, y)$ for this task. Our idea is to use the algorithm from Theorem 5.1 for that. In fact we only have to modify the running time analysis from above to consider the new running time of the internal stSP data structure. Thus, we will only sketch the argument in the following.

Corollary 5.13. *Assume that there is a $(1+\epsilon)$-approximate stSP data structure with constant query time and a total update time of $\widetilde{O}(m^\alpha n^\beta (\log W)^\gamma/\epsilon^\delta + \Delta_G)$ for weighted directed graphs*

undergoing edge deletions and edge weight increases and every $0 < \epsilon \leq 1$*. Then, given a set of nodes* S *of size* k*, we can, for all nodes* x *and* y *in* S*, maintain a distance estimate* $\delta(x,y)$ *such that* $\operatorname{dist}(x,y) \leq \delta(x,y) \leq (1+\epsilon)\operatorname{dist}(x,y)$*. This data structure has constant query time and a total update time of*

- $\widetilde{O}(m^{\alpha'}n^{\beta'}(\log W)^{\gamma+1}/\epsilon^{\delta+1} + \Delta_G)$*, where* $\alpha' = 1 - 2/(6\alpha+3\beta)$ *and* $\beta' = (8\alpha+4\beta)/(6\alpha+3\beta)$ *or*
- $\widetilde{O}(m^{\alpha'}n^{\beta'}(\log W)^{\gamma+1}/\epsilon^{\delta+1} + \Delta_G)$*, where* $\alpha' = 2(\alpha-1)/(3(\alpha+\beta)) + 1$ *and* $\beta' = 2\beta/(3(\alpha+\beta)) + 2/3$*.*

Proof sketch. We first sketch how to get the first of these two running times. We simply have to redo the analysis of Lemma 5.11. Assume that there is a $(1+\epsilon)$-approximate stSP data structure with constant query time and a total update time of $\widetilde{O}(m^{\alpha}n^{\beta}(\log W)^{\gamma}/\epsilon^{\delta} + \Delta_G)$. We have argued that, every graph $\mathcal{Q}(x,y)$ of some center x to some center y has at most n/b nodes and thus at most n^2/b^2 edges whp. Thus, for every graph $\mathcal{Q}(x,y)$ of some center x to some center y, we can maintain a $(1+\epsilon)$-approximate estimate of the distance from x to y in time $\widetilde{O}((n^2/b^2)^{\alpha}(n/b)^{\beta}(\log W)^{\gamma}/\epsilon^{\delta} + \Delta_{\mathcal{Q}(x,y)})$. By balancing the terms bmn/c, c^2m, and $c^2n^{2\alpha+\beta}/b^{2\alpha+\beta}$, we obtain $b = n^{3(4\alpha+2\beta)/(6\alpha+3\beta)-1}/m^{1/(6\alpha+3\beta)}$ and $c = n^{(4\alpha+2\beta)/(6\alpha+3\beta)}/m^{1/(6\alpha+3\beta)}$. Using the same arguments as in the proof of Lemma 5.11, we obtain a running time of $\widetilde{O}(m^{\alpha'}n^{\beta'}(\log W)^{\gamma+1}/\epsilon^{\delta+1} + \Delta_G)$ for the auxiliary algorithm, where $\alpha' = 1 - 2/(6\alpha+3\beta)$ and $\beta' = (8\alpha+4\beta)/(6\alpha+3\beta)$. Note that the running time needed for maintaining the distance estimate in the center graph from Lemma 5.4 is $\widetilde{O}(c^3 \log_{1+\epsilon} W/\epsilon + \Delta_G)$ and is dominated by $\widetilde{O}(c^2 m \log_{1+\epsilon} W/\epsilon + \Delta_G)$. Thus, the running time of the auxiliary algorithm gives the total running time.

To get the second running time, we use the same idea, but this time we do not only sample the hubs from the nodes but additionally sample each edge with probability $ab\ln(n\Delta_G)/m$ (for a large enough constant a) and use the tail u of each sampled edge (u,v) as a hub. Then we can show that the path union graphs we are building have at most n/b nodes and at most m/b edges whp by Lemma 4.16. Assume that there is a $(1+\epsilon)$-approximate stSP data structure with constant query time and a total update time of $\widetilde{O}(m^{\alpha}n^{\beta}(\log W)^{\gamma}/\epsilon^{\delta} + \Delta_G)$. Then, for every graph $\mathcal{Q}(x,y)$ of some center x to some center y, we can maintain a $(1+\epsilon)$-approximate estimate of the distance from x to y in time $\widetilde{O}((m/b)^{\alpha}(n/b)^{\beta}(\log W)^{\gamma}/\epsilon^{\delta} + \Delta_{\mathcal{Q}(x,y)})$. By balancing the terms bmn/c, c^2m, and $c^2m^{\alpha}n^{\beta}/(b^{\alpha}b^{\beta})$, we obtain $b = m^{(\alpha-1)/(\alpha+\beta)}n^{\beta/(\alpha+\beta)}$ and $c = m^{(\alpha-1)/(3(\alpha+\beta))}n^{\beta/(3(\alpha+\beta))+1/3}$. It follows that there is a $(1+\epsilon)$-approximate stSP data structure with constant query time and a total update time of $\widetilde{O}(m^{\alpha'}n^{\beta'}(\log W)^{\gamma+1}/\epsilon^{\delta+1} + \Delta_G)$, where $\alpha' = 2(\alpha-1)/(3(\alpha+\beta)) + 1$ and $\beta' = 2\beta/(3(\alpha+\beta)) + 2/3$. □

Using the first reduction of Corollary 5.13 and the parameters of Theorem 5.1, i.e., $\alpha = 5/7$, $\beta = 10/7$, $\gamma = 1$, and $\delta = 2$, we obtain $\alpha' = 23/30$ and $\beta' = 4/3$. Using these parameters with the second reduction of Corollary 5.13, we obtain $\alpha' = 25/27$ and $\beta' = 206/189$. This allows us to state the following running times.

Corollary 5.14. *Given a weighted directed graph undergoing edge deletions and edge weight increases, an approximation parameter* $0 < \epsilon \leq 1$*, and set of nodes* S *of size* k*, we can, for all nodes* x *and* y *in* S*, maintain a distance estimate* $\delta(x,y)$ *such that* $\operatorname{dist}(x,y) \leq \delta(x,y) \leq (1+\epsilon)\operatorname{dist}(x,y)$*. This data structure has constant query time and a total update time of* $\widetilde{O}(m^{23/30}n^{4/3}(\log W)^2/\epsilon^3 + \Delta_G)$ *or* $\widetilde{O}(m^{25/27}n^{206/189}(\log W)^3/\epsilon^4 + \Delta_G)$*.*

6 $(1+\epsilon)$-Approximate stSP for Sparse Weighted Graphs

In the following we provide a $(1+\epsilon)$-approximate algorithm for stSP that is tailored to sparse graphs.

Theorem 6.1. *For every $0 < \epsilon \leq 1$, there is a $(1+\epsilon)$-approximate stSP data structure for weighted directed graphs undergoing edge deletions and edge weight increases with constant query time and a total update time of $\widetilde{O}(m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$.*

Note that this algorithm breaks the "$O(mn)$-barrier" as long as $m = o(n^{1.5})$.

6.1 Algorithm Description

In some sense, the algorithm of Theorem 5.1 becomes inefficient for sparse graphs because maintaining the distance estimates for the center graph becomes inefficient as we maintain distance estimates between *all* pairs of centers. The algorithm we describe in the following exploits the fact that not all pairs of centers are always equally relevant. It has a more "lazy" approach and tries to compute only the information that is really necessary.

The algorithm has two parameters h and b, which we will set in an optimal way later on. It also uses the number $c = n/(\epsilon h)$. At the initialization the algorithm determines a set $B = \{b_1, \ldots, b_{|B|}\}$ of hubs and a set C of centers in the same way as in the dense setting (see Section 5.2). We fix an arbitrary order on the hubs. We sample each edge with probability $(ab \ln(n\Delta_G))/m$ for a large enough constant a. The tail u of each sampled edge (u, v) is used as a hub. We also sample each node with probability $(ac \ln(n\Delta_G))/n$ for a large enough constant a. The sampled nodes are used as the centers. Besides these sampled nodes, also the source s and the sink t are used as centers.

The algorithm works in phases. At the beginning of every phase we compute a shortest path π from s to t in G using, for example, Dijkstra's algorithm. During the whole phase $\delta(s,t) = \text{dist}_G(s,t)$, the distance from s to t at the beginning of the phase, is the distance estimate returned by the algorithm for the whole phase. We then identify the set of centers $A = \{c_1, c_2, \ldots, c_{|A|}\}$ on this shortest path, which we call *active* centers for the current phase. We number the centers according to their order on the path, i.e., $c_1 = s$ and $c_{|A|} = t$. For every $1 \leq k < |A|$, we call the center c_{k+1} the *successor* of the center c_k. Note that, for every $1 \leq k < |A|$, the number of edges from c_k to its successor c_{k+1} on the path π is at most h whp by Lemma 4.16. Furthermore, for every active center c_k with successor c_{k+1} we compute the largest index $i(c_k, c_{k+1})$ such that $\text{dist}^h_G(x,y) \geq (1+\epsilon)^{i(c_k,c_{k+1})}$.

The main idea of the algorithm is as follows (see Algorithm 2 for the pseudocode): As soon as the bounded hops distance from an active center c_k to its successor c_{k+1} has increased to more than $(1+\epsilon)^{i(c_k,c_{k+1})+1}$, the algorithm cannot guarantee the desired approximation anymore. It will then simply start a new phase. This approach will still be efficient because we can limit the number of phases by $\widetilde{O}(c^2 \log_{1+\epsilon} W)$.

After every update in G (edge deletion or edge weight increase), the algorithm proceeds as follows for every active center c_k with successor c_{k+1}. First, it checks whether there is still a hub v such that $\widetilde{\text{dist}}^h_G(c_k, v) + \widetilde{\text{dist}}^h_G(v, c_{k+1}) \leq (1+\epsilon)^{i(c_k,c_{k+1})+2}$. For this purpose we maintain the index $j(c_k, c_{k+1})$ of the hub that currently fulfills this inequality, i.e., we check them one after the other in a fixed order. If there is such a hub, then the algorithm does not do any more work for c_k. If not, the algorithm initializes a graph $\mathcal{Q}(c_k, c_{k+1})$ as the path union graph $\mathcal{P}(c_k, c_{k+1}, h, i, G)$. From now on $\mathcal{Q}(c_k, c_{k+1})$ will undergo the same updates as G does. The algorithm will also maintain $\text{dist}_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1})$ by simply computing a shortest path from c_k to c_{k+1} after every update in $\mathcal{Q}(c_k, c_{k+1})$ and if $\text{dist}_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1}) > (1+\epsilon)^{i(c_k,c_{k+1})+2}$ it will start a new phase.

Note that, for every $1 \leq k < |A|$ the graph $\mathcal{Q}(c_k, c_{k+1})$ was initialized as a path union graph, but after several updates in G it might not fulfill the conditions of a path union graph anymore. However, it will always contain $\mathcal{P}(c_k, c_{k+1}, h, i, G)$ as a subgraph. We cannot afford to enforce that $\mathcal{Q}(c_k, c_{k+1})$ equals $\mathcal{P}(c_k, c_{k+1}, h, i, G)$ after each update, but we will enforce it at the beginning of each phase. At this point we recompute the path union graph for every active center c_k with successor c_{k+1} if $\mathcal{Q}(c_k, c_{k+1})$ has already been constructed in a previous phase. We do this by setting $\mathcal{Q}(c_k, c_{k+1})$ to $\mathcal{P}(c_k, c_{k+1}, h, i, \mathcal{Q}(c_k, c_{k+1}))$, i.e., by computing the path union graph in the graph $\mathcal{Q}(c_k, c_{k+1})$. It can easily be verified that the result of this operation is the same as recomputing the path union graph in G from scratch.

Lemma 6.2. *Let $\mathcal{Q}(x, y)$ be a graph that contains all paths from x to y in G of length at most h and weight at most $(1+\epsilon)^{i+1}$ (and possibly additional nodes and edges). Then $\mathcal{P}(x, y, h, i, \mathcal{Q}(x, y))$ is equal to $\mathcal{P}(x, y, h, i, G))$.*

6.2 Correctness Proof

The correctness of the algorithm is straightforward as the algorithm itself checks whether it can still provide the desired approximation guarantee.

Lemma 6.3 (Correctness)**.** $\delta(s,t) \leq \text{dist}_G(s,t) \leq (1+3\epsilon)\delta(s,t)$

Proof. Let G' be the version of the graph at the beginning of the current phase. We first show that $\text{dist}_G(c_k, c_{k+1}) \leq (1+\epsilon)^2 \,\text{dist}_{G'}(c_k, c_{k+1})$ for all $1 \leq k < |A|$. Once we have shown this, the claim easily follows by the following inequalities:

$$\begin{aligned}
\text{dist}_G(s,t) &\leq \sum_{1 \leq k < |A|} \text{dist}_G(c_k, c_{k+1}) \\
&\leq \sum_{1 \leq k < |A|} (1+\epsilon)^2 \,\text{dist}_{G'}(c_k, c_{k+1}) \\
&= (1+\epsilon)^2 \sum_{1 \leq k < |A|} \text{dist}_{G'}(c_k, c_{k+1}) \\
&= (1+\epsilon)^2 \delta(s,t) \leq (1+3\epsilon)\delta(s,t)\,.
\end{aligned}$$

Consider an active center c_k and its successor c_{k+1} for some $1 \leq k < |A|$. Remember that $\text{dist}_{G'}(c_k, c_{k+1}) \geq (1+\epsilon)^{i(c_k, c_{k+1})}$. Consider first the case that $j(c_k, c_{k+1}) \leq |B|$. Then there is a hub v such that $\widetilde{\text{dist}}^h_G(c_k, v) + \widetilde{\text{dist}}^h_G(v, c_{k+1}) \leq (1+\epsilon)^{i(c_k, c_{k+1})+2}$. We now get the desired bound as follows:

$$\begin{aligned}
\text{dist}_G(c_k, c_{k+1}) &\leq \text{dist}_G(c_k, v) + \text{dist}_G(v, c_{k+1}) \\
&\leq \widetilde{\text{dist}}^h_G(c_k, v) + \widetilde{\text{dist}}^h_G(v, c_{k+1}) \\
&\leq (1+\epsilon)^{i(c_k, c_{k+1})+2} \leq (1+\epsilon)^2 \,\text{dist}_{G'}(c_k, c_{k+1})\,.
\end{aligned}$$

Consider now the case that $j(c_k, c_{k+1}) = |B|+1$. The algorithm guarantees that $\text{dist}_{\mathcal{Q}(c_k, c_{k+1})}(c_k, c_{k+1}) \leq (1+\epsilon)^{i(c_k, c_{k+1})+2}$. As $\mathcal{Q}(c_k, c_{k+1})$ is a subgraph of G, we have $\text{dist}_G(c_k, c_{k+1}) \leq \text{dist}_{\mathcal{Q}(c_k, c_{k+1})}(c_k, c_{k+1})$. Thus, we get

$$\text{dist}_G(c_k, c_{k+1}) \leq \text{dist}_{\mathcal{Q}(c_k, c_{k+1})}(c_k, c_{k+1}) \leq (1+\epsilon)^{i(c_k, c_{k+1})+2} \leq (1+\epsilon)^2 \,\text{dist}_{G'}(c_k, c_{k+1})\,. \qquad \square$$

This approximation can be turned into the usual $(1+\epsilon)$-approximation by running our algorithm with the parameter $\epsilon' = \epsilon/3$.

Algorithm 2: Algorithm for sparse graphs

Determine set of hubs $B = \{b_1, \ldots, b_{|B|}\}$ and set of centers C by random sampling of edges and nodes
For every hub $v \in B$ maintain $\widetilde{\text{dist}}^h_G(x, v)$ and $\widetilde{\text{dist}}^h_G(v, x)$ for every center x (see Lemma 4.9)
Set $i(x, y) = 0$ and $j(x, y) = 1$ for every pair of centers x and y and call `update()`
After every deletion or edge weight increase, report update to internal data structures and call `update()`
`initialize() // start first phase`

`initialize() // first initialization and beginning of a new phase`
    Compute a shortest path π from s to t in G
    Set $\delta(s, t) = \text{dist}_G(s, t)$
    Determine set of active centers $A = \{c_1, \ldots, c_{|A|}\}$ on π
    foreach $1 \leq k < |A|$ **do** `// iterate over active centers`
        $i \leftarrow \lceil \log_{1+\epsilon} \text{dist}_G(c_k, c_{k+1}) \rceil$
        if $i = i(c_k, c_{k+1})$ **then**
            if $j(c_k, c_{k+1}) = |B| + 1$ **then**
                `// Update path union graph`
                Set $\mathcal{Q}(c_k, c_{k+1})$ to be the path union graph $\mathcal{P}(c_k, c_{k+1}, h, \mathcal{Q}(c_k, c_{k+1}))$
        else
            $i(c_k, c_{k+1}) \leftarrow i$
            $j(c_k, c_{k+1}) \leftarrow 1$
            `update()`

`update()`
    foreach $1 \leq k < |A|$ **do** `// iterate over active centers`
        while $j(c_k, c_{k+1}) \leq |B|$ **and**
        $\widetilde{\text{dist}}^h_G(c_k, j(c_k, c_{k+1})) + \widetilde{\text{dist}}^h_G(j(c_k, c_{k+1}), c_{k+1}) > (1 + \epsilon)^{i(c_k, c_{k+1})+2}$ **do**
            $j(c_k, c_{k+1}) \leftarrow j(c_k, c_{k+1}) + 1$
            if $j(c_k, c_{k+1}) = |B| + 1$ **then**
                Initialize $\mathcal{Q}(c_k, c_{k+1})$ as the path union graph $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$
                Start maintaining $\mathcal{Q}(c_k, c_{k+1})$ by making the same future edge deletions and edge weight increases as in G
        if $j(c_k, c_{k+1}) = |B| + 1$ **then**
            Compute a shortest path from c_k to c_{k+1} in $\mathcal{Q}(c_k, c_{k+1})$
            if $\text{dist}_{\mathcal{Q}(c_k, c_{k+1})}(c_k, c_{k+1}) > (1 + \epsilon)^{i(c_k, c_{k+1})+2}$ **then**
                `initialize() // start new phase`

6.3 Running Time Analysis

Before we obtain the desired running time we have to show three crucial properties of our algorithm. First, we bound the number of phases of the algorithm. Second, we show that the path union graphs of consecutive active centers do not have too much overlap. Third, we show that the path union graphs constructed by the algorithm are relatively sparse. Note that the first two properties are specifically needed for this algorithm tailored to sparse graphs, whereas the third property is crucial in all our algorithms.

Lemma 6.4. *The number of phases of the algorithm is* $\widetilde{O}(c^2 \log_{1+\epsilon} W)$.

Proof. Note that for all centers x and y the index $i(x, y)$ is non-decreasing as distances in G never decrease under edge deletions and edge weight increases. Every time the algorithm starts a new phase, there are two active centers c_k and c_{k+1} (for some $1 \leq k < |A|$) such that $\text{dist}_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1}) > (1+\epsilon)^{i(c_k,c_{k+1})+2}$. Assume by contradiction that $\text{dist}^h_G \leq (1+\epsilon)^{i(c_k,c_{k+1})+1}$. Since $\mathcal{Q}(c_k, c_{k+1})$ was initialized as a path union graph, it contains all paths from c_k to c_{k+1} of length at most h and weight at most $(1+\epsilon)^{i(c_k,c_{k+1})+1}$. Since $\text{dist}^h_G \leq (1+\epsilon)^{i(c_k,c_{k+1})+1}$, we therefore have $\text{dist}^h_G(c_k, c_{k+1}) = \text{dist}^h_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1})$. We arrive at the following contradictory statement:

$$\begin{aligned}(1+\epsilon)^{i(c_k,c_{k+1})+2} &< \text{dist}_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1}) \\ &\leq \text{dist}^h_{\mathcal{Q}(c_k,c_{k+1})}(c_k, c_{k+1}) \\ &= \text{dist}^h_G(c_k, c_{k+1}) \leq (1+\epsilon)^{i(c_k,c_{k+1})+1}.\end{aligned}$$

It follows that $\text{dist}^h_G > (1+\epsilon)^{i(c_k,c_{k+1})+1}$. Note that at the beginning of the current phase we had $\text{dist}^h_G \leq (1+\epsilon)^{i(c_k,c_{k+1})+1}$, where G' is the version of the graph at the beginning of the phase. As G only undergoes edge deletions, this distance will never decrease anymore. Therefore this kind of increase can happen at most $\lceil \log_{1+\epsilon} W \rceil$ times for every pair of centers, because the number of possible indices is $\lceil \log_{1+\epsilon} W \rceil$. Thus, the number of phases is limited by $\widetilde{O}(c^2 \log_{1+\epsilon} W)$. □

Lemma 6.5. *For every node v there are at most $O(\log_{1+\epsilon} W)$ indices k with $1 \leq k < |A|$ such that v is contained in the graph $\mathcal{Q}(c_k, c_{k+1})$.*

Proof. Note that for every $1 \leq k < |A|$ the graph $\mathcal{Q}(c_k, c_{k+1})$ is a subgraph of the path union graph $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$ where G is the version of the graph at the beginning of the current phase. Thus, it is sufficient to show that for every node v there are at most $O(\log_{1+\epsilon} W)$ indices k such that v is contained in the graph $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$.

Fix some index i such that $0 \leq i \leq \lfloor \log_{1+\epsilon} W \rfloor$. We show that every node v is contained in a constant number of path union graphs $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$ such that $i(c_k, c_{k+1}) = i$. As there are at most $\lceil \log_{1+\epsilon} W \rceil$ such indices, this will imply the lemma.

In particular, suppose that v is contained in nine path union graphs $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$ such that $i(c_k, c_{k+1}) = i$. Let $k_1, \ldots, k_9$ be the corresponding indices and assume without loss of generality that $k_1 < k_2 \ldots < k_9$. For all $k \in \{k_1, \ldots, k_9\}$, we have $i(c_k, c_{k+1}) = i$ and thus $\text{dist}^h_G(c_k, c_{k+1}) \geq (1+\epsilon)^i$. Note that an active center is connected to its successor by a shortest path of length at most h. Therefore we get $\text{dist}_G(c_k, c_{k+1}) = \text{dist}^h_G(c_k, c_{k+1}) \geq (1+\epsilon)^i$ for all $k \in \{k_1, \ldots, k_9\}$ and it follows that

$$\text{dist}_G(c_{k_1}, c_{k_9+1}) \geq 9(1+\epsilon)^i \tag{8}$$

We now derive an upper bound on $\text{dist}_G(c_{k_1}, c_{k_9+1})$ contradicting this lower bound. Since v is contained in the path union graph $\mathcal{P}(c_{k_1}, c_{k_1+1}, h, i, G)$ we know by the definition of path union

graphs that $\widetilde{\text{dist}}_G^{h,i}(c_{k_1}, v) + \widetilde{\text{dist}}_G^{h,i}(v, c_{k_1+1}) \leq (1+\epsilon)^{i+2}$ and thus $\widetilde{\text{dist}}_G^{h,i}(c_{k_1}, v) \leq (1+\epsilon)^{i+2}$. A similar argument shows that $\widetilde{\text{dist}}_G^{h,i}(v, c_{k_9}) \leq (1+\epsilon)^{i+2}$. We therefore have

$$\text{dist}_G(c_{k_1}, c_{k_9+1}) \leq \text{dist}_G(c_{k_1}, v) + \text{dist}_G(v, c_{k_9+1}) \leq \widetilde{\text{dist}}_G^{h,i}(c_{k_1}, v) + \widetilde{\text{dist}}_G^{h,i}(v, c_{k_9+1}) \leq 2(1+\epsilon)^{i+2} \tag{9}$$

By combining Inequalities (8) and (9) we get $9 \leq 2(1+\epsilon)^2$, which is a contradictory statement because $\epsilon \leq 1$. Thus, v is contained in at most eight path union graphs $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G)$ such that $i(c_k, c_{k+1}) = i$. □

Lemma 6.6 (Sparsity of path union graph)**.** *For every $1 \leq k < |A|$, the graph $\mathcal{Q}(c_k, c_{k+1})$ has at most m/b edges whp, if it exists.*

Proof. Let G' be the version of the graph in which the algorithm sets $j(c_k, c_{k+1})$ to $|B|+1$. Note that $\mathcal{Q}(c_k, c_{k+1})$ is a subgraph of the path union graph $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G')$. Therefore it is sufficient to show that $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G')$ has at most m/b edges.

Suppose that $\mathcal{P}(c_k, c_{k+1}, h, i(c_k, c_{k+1}), G')$ has more than m/b edges. Then one of these edges, say (u, v), has been sampled at the initialization of the algorithm whp by Lemma 4.16. Therefore the node u is a hub contained in the path union graph. By the definition path union graphs (see Definition 4.13) we have

$$\widetilde{\text{dist}}_G^{h}(x, u) + \widetilde{\text{dist}}_G^{h}(u, y) \leq \widetilde{\text{dist}}_G^{h,i(c_k,c_{k+1})}(x, u) + \widetilde{\text{dist}}_G^{h,i(c_k,c_{k+1})}(u, y) \leq (1+\epsilon)^{i(c_k,c_{k+1})+2}.$$

But our algorithm only sets $j(c_k, c_{k+1})$ to $|B|+1$ when it has ruled out the existence of such a hub. (Note that once the inequality holds it will always hold as distances never decrease under edge deletions.) This is a contradiction and proves that $\mathcal{Q}(c_k, c_{k+1})$ has at most m/b edges. □

Using the lemmas above, we can now bound the time needed for computing the path union graphs. We amortize this cost over all phases.

Lemma 6.7. *Computing the path union graphs takes time $\widetilde{O}(c^2 m \log_{1+\epsilon} W)$ in total.*

Proof. Computing the path union graph in a graph G takes time $O(m)$, where m is the number of edges of m. Our algorithm does the following, for every pair of centers x and y. The first time it computes $\mathcal{Q}(x, y)$ as the result of computing the path union graph in G. Every time it recomputes $\mathcal{Q}(x, y)$ at the beginning of a phase it does this by computing a path union graph in $\mathcal{Q}(x, y)$. Computing the path union graph in $\mathcal{Q}(x, y)$ takes time proportional to the number of edges in $\mathcal{Q}(x, y)$. Note that a path union graph computed in $\mathcal{Q}(x, y)$ is a subgraph of $\mathcal{Q}(x, y)$. Thus, we can view the recomputation of the path union graph as a process that removes edges from $\mathcal{Q}(x, y)$. In particular the time needed for computing the path union graph in $\mathcal{Q}(x, y)$ is proportional to the sum of the number of edges removed from $\mathcal{Q}(x, y)$ and the sum of the edges remaining in $\mathcal{Q}(x, y)$. We pay the cost of the computation by charging these two types of edges. Every edge is removed from $\mathcal{Q}(x, y)$ at most once. Therefore the total charge for all removed edges is $O(m)$ for every pair of centers and thus $\widetilde{O}(c^2 m)$ in total.

It remains to bound the charge on the remaining edges. Note that the remaining edges are exactly those that are contained in the path union graph we are computing. By Lemma 6.5 every node, and thus also every edge, is contained in at most $O(\log_{1+\epsilon} W)$ path union graphs of consecutive active centers. Thus the charge on the remaining edges is $O(m \log_{1+\epsilon} W)$ in every phase. As the number of phases is $\widetilde{O}(c^2)$ by Lemma 6.4, the total charge for the remaining edges over all phases is $O(c^2 m \log_{1+\epsilon} W)$. □

We now provide the overall running time analysis and explain how to set the parameters h and b to get the desired running time.

Lemma 6.8 (Running time)**.** *The total update time of Algorithm 2 is* $\widetilde{O}(m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$.

Proof. The number of hubs and centers are $\widetilde{O}(b)$ and $\widetilde{O}(c)$ in expectation, respectively. Remember that by Lemma 4.17 we may assume that $\Delta_G = \widetilde{O}(m \log_{1+\epsilon} W)$. We bound the individual running times as follows.

- Computing all path union graphs takes time $\widetilde{O}(c^2 m \log_{1+\epsilon} W)$ by Lemma 6.7.
- For every hub v, we maintain $\widetilde{\text{dist}}^h_G(x, v)$ and $\widetilde{\text{dist}}^h_G(v, y)$ for all nodes x and y. As this takes time $\widetilde{O}((mh \log_{1+\epsilon} W)/\epsilon + \Delta_G)$ for every hub by Lemma 4.9, these costs are bounded by $\widetilde{O}((bmh \log_{1+\epsilon} W)/\epsilon + b\Delta_G) = \widetilde{O}((bmn \log_{1+\epsilon} W)/(\epsilon c) + bm \log_{1+\epsilon} W) = \widetilde{O}((bmn \log_{1+\epsilon} W)/(\epsilon c))$.
- At the beginning of every phase, we compute a shortest path from s to t using, for example, Dijkstra's algorithm which takes time $\widetilde{O}(m)$. As the number of phases is $\widetilde{O}(c^2 \log_{1+\epsilon} W)$ by Lemma 6.4, these costs are bounded by $\widetilde{O}(c^2 m \log_{1+\epsilon} W)$.
- After every deletion of an edge (u, v) we compute, for every active center c_k with successor c_{k+1} such that (u, v) is contained in $\mathcal{Q}(c_k, c_{k+1})$, a shortest path from c_k to c_{k+1} in $\mathcal{Q}(c_k, c_{k+1})$. As $\mathcal{Q}(x, y)$ has at most m/b edges whp by Lemma 6.6 and (u, v) is contained in $O(\log_{1+\epsilon} W)$ graphs $\mathcal{Q}(c_k, c_{k+1})$ by Lemma 6.5, this takes time $O((m/b) \log_{1+\epsilon} W)$ after every update. Thus, the total cost of this step is $O((m/b)\Delta_G \log_{1+\epsilon} W)$, where Δ_G is the number of updates in G. Using $\Delta_G = \widetilde{O}(m \log_{1+\epsilon} W)$, we get a total cost of $O((m^2/b)(\log_{1+\epsilon} W)^2)$
- The remaining costs of the auxiliary algorithm can be bounded as follows. The algorithm incurs a cost of $\widetilde{O}(1)$ for every update in the graph and for every update of the indices $i(x, y)$ or $j(x, y)$ of any pair of centers x and y. As there are $\widetilde{O}(b)$ different values of $i(x, y)$ and $\widetilde{O}(\log_{1+\epsilon} W)$ different values of $j(x, y)$, these costs are bounded by $\widetilde{O}(bc^2 \log_{1+\epsilon} W + \Delta_G)$.

Note that the term bc^2 is dominated by the term $c^2 m$. Furthermore the term mn/c is dominated by bmn/c. By balancing the terms $c^2 m$, bmn/c, and m^2/b, we obtain $b = m^{3/5}/n^{2/5}$ and $c = m^{1/5}n^{1/5}$. Thus, we obtain a running time of $\widetilde{O}(m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$. Here we use the fact that $1/\log(1+\epsilon) = O(1/\epsilon)$. □

If we wanted to maintain $(1+\epsilon)$-approximate shortest paths for k source/sink pairs, the naive running time would be $\widetilde{O}(km^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$. Upon inspecting the running time analysis above, we see that this task can be carried out a little bit more efficiently.

Corollary 6.9. *Given a weighted directed graph undergoing edge deletions and edge weight increases, an approximation parameter* $0 < \epsilon \leq 1$*, and* k *source/sink pairs* $(s_i, t_i)_{1 \leq i \leq k}$*, we can, for all* $1 \leq i \leq k$*, maintain a distance estimate* $\delta(s_i, t_i)$ *such that* $\text{dist}_G(s_i, t_i) \leq \delta(s_i, t_i) \leq (1+\epsilon)\,\text{dist}_G(s_i, t_i)$*. This data structure has a total update time of* $\widetilde{O}(k^{3/5}m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$ *and constant query time.*

Proof. We simply run k instances of the algorithm that "share" a common set of hubs. Thus, the contributions to the running time analyzed in the proof of Lemma 6.8 all have to be multiplied by k, except for the term bmn/c we get for maintaining $\widetilde{\text{dist}}^h_G(x, v)$ and $\widetilde{\text{dist}}^h_G(v, y)$ for every hub v and all nodes x and y. By balancing the terms $kc^2 m$, bmn/c, and km^2/b, we obtain $b = k^{2/5}m^{3/5}/n^{2/5}$ and $c = m^{1/5}n^{1/5}/k^{1/5}$. Thus, we get a running time of $\widetilde{O}(k^{3/5}m^{7/5}n^{2/5}(\log W)^2/\epsilon^2)$. □

7 $O(\log n)$-Approximate stSP for Dense Unweighted Graphs

In the following we give an $O(\log n)$-approximate stSP data structure for *unweighted* graphs. This algorithm is more efficient than the more general $(1+\epsilon)$-approximation we presented in Section 5.

Theorem 7.1. *There is an $O(\log n)$-approximate stSP data structure for unweighted directed graphs undergoing edge deletions with constant query time and a total update time of $\widetilde{O}(m^{1/2}n^{7/4})$.*

We will use the same algorithmic framework as in Section 5 where we efficiently maintain a center graph using hubs and path union graphs. The new aspect presented in the following is a faster way of constructing the path union graph for all centers. In the previous approach this took $\widetilde{O}(c^2m)$ time, where $\widetilde{O}(c)$ is the number of centers. For the new algorithm this will only take $\widetilde{O}(cn^2 + c^2n^2/b^2)$ time, where the second term will be dominated by other running time aspects of the overall algorithm.

Our main tool is a novel "approximate" reachability data structure that maintains a subgraph $R(s)$ of G, which is an "out-dated" version of the subgraph induced by all nodes reachable from s. If we know that k nodes can be removed from this graph, i.e. are not reachable from s anymore, the data structure updates $R(s)$ by removing at least k nodes efficiently. In our main algorithm we will use probabilistic arguments to determine k without knowing explicitly which nodes are to be removed.

7.1 Approximate Reachability

Our first goal is to design a data structure for "approximate" reachability from a source s up to depth h in a directed graph G undergoing edge deletions. The data structure maintains a subgraph $R(s)$ of G. As an invariant we demand that $R(s)$ contains the subgraph of G induced by all nodes v such that $\text{dist}_G(s,v) \leq h$, but it might possibly contain more nodes and edges. The data structure provides an operation `remove`(k), where k is a positive integer, which removes at least k nodes from $R(s)$ and returns the nodes that have been removed. However, this operation only works as desired if it is guaranteed that $R(s)$ contains at least k nodes v such that $\text{dist}_G(s,v) > (h+1)\lceil \log n \rceil$.

Definition 7.2. *An approximate reachability data structure (short: AR data structure) on an unweighted directed graph G, a source node s, and a depth h, maintains a subgraph $R(s)$ of G such that $R(s)$ contains the subgraph of G induced by all nodes v with $\text{dist}_G(s,v) \leq h$. It provides the following operations:*

- `delete`(u,v)*: Delete the edge (u,v) from G.*
- `remove`(k)*: If there are at least k nodes v in $R(s)$ such that $\text{dist}_G(s,v) > (h+1)\lceil \log n \rceil$, remove at least k nodes and their incident edges from $R(s)$. Otherwise, remove an arbitrary number of nodes from $R(s)$. Return the removed nodes.*

Theorem 7.3 (Efficient AR data structure)**.** *For every unweighted directed graph G with source node s and every integer $h \geq 1$, there is an AR data structure that with a total running time of $O(n^2 + q\min(m\log n, n^2))$ over all operations, where q is the number of* `remove`(k) *operations such that the number of nodes v in $R(s)$ with $\text{dist}_G(s,v) > (h+1)\lceil \log n \rceil$ is less than k.*

In the following we will prove Theorem 7.3. We will in fact only have to deal with the operation `remove`(1). The operation `remove`(k) can simply be implemented by calling `remove`(1) repeatedly until at least k nodes have been removed from $R(s)$.

We now show how to implement the data structure. In the following we will fix the source node s and simply write R instead of $R(s)$. We initialize R with G and after every deletion of an edge (u, v) from G we also delete (u, v) from R. We consider an arbitrary but fixed order of the (current) edges of R. For every $0 \leq i \leq \lceil \log n \rceil$ we let R_i denote the subgraph of R in which every node has only its first 2^i outgoing edges. Thus, the out-degree of every node in R_i is at most 2^i and every node v with $\text{outdeg}_R(v) \leq 2^i$ has all of its outgoing edges in R_i. We denote by $B_i = \{v \in V \mid \text{outdeg}(v) > 2^i\}$ the set of nodes that do not have all of their outgoing edges in R_i. Note that $B_i \subseteq B_{i-1}$ as we have fixed an order of the edges. Furthermore we consider $S_i = B_i \cup \{s\}$, the set of sources in R_i. We denote the distance of a node v from S_i in R_i by $\text{dist}_{R_i}(S_i, v) = \min_{v' \in S_i} \text{dist}_{R_i}(v', v)$. When `remove(1)` is called we run Algorithm 3, which searches for the smallest index i such that there is a node $u \in B_{i-1}$ with $\text{dist}_{R_i}(S_i, u) > h + 1$. It then removes all nodes v such that $\text{dist}_{R_i}(S_i, v) > h$ from v.

Algorithm 3: Algorithm for `remove`(1)-operation

```
1  remove(1)
2      D ← ∅
3      i ← 0
4      while D = ∅ and i ≤ ⌈log n⌉ do
5          Compute BFS-tree from S_i in R_i and corresponding distances dist_{R_i}(S_i, ·)
6          if there is a node u ∈ B_{i-1} such that dist_{R_i}(S_i, u) > h + 1 then
7              add all nodes v such that dist_{R_i}(S_i, v) > h to D
8          i ← i + 1
9      remove nodes in D and their incident edges from R
10     return D
```

We now argue about the correctness and the running time of Algorithm 3.

Lemma 7.4 (Correctness)**.** *At any time, for every node v if $\text{dist}_G(s, v) \leq h$, then v is contained in R.*

Proof. Let R be the version of R *before* the `remove`(1)-operation. By induction we know that R contains all nodes v such that $\text{dist}_G(s, v) \leq h$ and the edges between them. This implies that $\text{dist}_R(s, v) = \text{dist}_G(s, v)$ for every node v such that $\text{dist}_G(s, v) \leq h$. Thus, it is sufficient to argue that we do not remove any node v such that $\text{dist}_R(s, v) \leq h$.

Suppose some node v is removed at level i. Then we know that $\text{dist}_{R_i}(S_i, v) > h$ by the rules of the algorithm. Assume that $\text{dist}_R(s, v) \leq h$ and consider a shortest path π from s to v in R. We now argue that all edges on this path are contained in R_i, which will imply $\text{dist}_{R_i}(s, v) \leq h$, contradicting the fact that $\text{dist}_{R_i}(S_i, v) > h$. Let v' be the last node of π that misses its outgoing path edge in R_i. Then it must be the case that $v' \in B_i$. As the subpath from v to v' is contained in R_i, we have $\text{dist}_{R_i}(v', v) \leq h$ and therefore also $\text{dist}_{R_i}(S_i, v) \leq h$, which contradicts the fact that $\text{dist}_{R_i}(S_i, v) > h$. □

Lemma 7.5. *For every $1 \leq i \leq \lceil \log n \rceil$, if there is no node $u \in B_{i-1}$ such that $\text{dist}_{R_i}(S_i, u) > h+1$, then, for every node v, $\text{dist}_{R_i}(S_i, v) \leq \text{dist}_{R_{i-1}}(S_{i-1}, v)+h+1$, i.e., the distance of v from the source nodes has increased by at most $h + 1$ from G_{i-1} to G_i.*

Proof. First, observe that $\text{dist}_{R_i}(S_i, v) \leq h + 1$ for every node $v \in S_{i-1}$. This is clearly true for every node $v \in S_i$ as then $\text{dist}_{R_i}(S_i, v) = 0$. Furthermore, every node $v \in S_{i-1} \setminus S_i$ is contained in B_{i-1} and by the assumption of the lemma we then have $\text{dist}_{R_i}(S_i, v) \leq h + 1$.

Now consider an arbitrary node $v \notin S_{i-1}$. Let $v' \in S_{i-1}$ be the node closest to v in R_{i-1}, i.e., $\text{dist}_{R_{i-1}}(S_{i-1}, v) = \text{dist}_{R_{i-1}}(v', v)$. As argued above, we have $\text{dist}_{R_i}(S_i, v') \leq h + 1$. Since all edges of R_{i-1} are also contained in R_i we have $\text{dist}_{R_i}(v', v) \leq \text{dist}_{R_{i-1}}(v', v)$. By applying the triangle inequality we get:

$$\begin{aligned}\text{dist}_{R_i}(S_i, v) &\leq \text{dist}_{R_i}(v', v) + \text{dist}_{R_i}(S_i, v') \\ &\leq \text{dist}_{R_{i-1}}(v', v) + h + 1 \\ &= \text{dist}_{R_{i-1}}(S_{i-1}, v) + h + 1\,.\end{aligned}$$

□

Lemma 7.6 (Running time)**.** *The worst-case time for each* `remove`(1)*-operation is* $O(\min(n^2, m \log n)$*. The total time over all* `remove`(1)*-operations for which there is a node* v *in* R *such that* $\text{dist}_G(s, v) > (h+1)\lceil \log n \rceil$ *is* $O(n^2)$*.*

Proof. The first part of the lemma follows easily from the fact that computing a BFS-tree in R_i takes time $O(\min(n2^i, m))$ for $0 \leq i \leq \lceil \log n \rceil$.

We now argue that `remove`(1) removes at least one node from R if R contains a node v with $\text{dist}_G(s, v) > (h+1)\lceil \log n \rceil$. If the algorithm would never remove any node from R, i.e., $\text{dist}_{R_i}(S_i, v) > h + 1$ for all $0 \leq i \leq \lceil \log n \rceil$ and every node v, then we would know by Lemma 7.5 that

$$\text{dist}_G(s, v) \leq \text{dist}_R(s, v) = \text{dist}_{R_{\lceil \log n \rceil}}(s, v) \leq (h+1)\lceil \log n \rceil$$

for every node v, contradicting the assumption of the lemma.

Let i be the level where `remove`(1) added nodes to D (as at least one node is removed, such a level exists). Then D contains every node v such that $\text{dist}_{R_i}(S_i, v) > h$ and we know that there is some node $u \in B_{i-1}$ such that $\text{dist}_{R_i}(S_i, u) > h+1$. The graph R_i has at most $n2^i$ edges, and thus computing a BFS-tree in R_i takes time $O(n2^i)$. Therefore, the total work performed in level i and all previous levels is $O(n2^{i+1}) = O(n2^i)$. Since $u \in B_i$, the out-degree of u in R_i is at least 2^{i-1} and for all neighbors v with an edge (u, v) we have $\text{dist}_{R_i}(S_i, v) > h$, i.e., all of these neighbors are removed from R. Thus, at least 2^{i-1} nodes are removed from R. We charge the running time of $O(n2^i)$ to the 2^{i-1} nodes we have removed from R. Thus, we charge $O(n)$ to each node removed from R. Over all `remove`-operations at most n nodes are removed from R. Therefore the total time for all `remove`-operations is $O(n^2)$. Computing the initial BFS-tree takes time $O(m)$, which is dominated by $O(n^2)$. □

Note that the depth parameter h does not show up in the running time.

7.2 Efficient Construction of Path Union Graphs

Before we give the rest of the algorithm, which mainly uses the same arguments as before, we show how to initialize the path union graphs more efficiently, which is the key idea in this section. At this point we only have to know a few things about the main algorithm. The main algorithm has a set of centers of size $\widetilde{O}(c)$ and might construct some path union graphs $\mathcal{P}(x, y, h, G)$ for some pairs of centers x and y, where h is a parameter of the algorithm. It constructs the path union graph $\mathcal{P}(x, y, h, G)$ for the centers x and y only when there is no hub v anymore such that $\text{dist}_G(x, v) + \text{dist}_G(v, y) \leq 4h \log n$. The hubs are a set of nodes sampled from the initial graph with probability $(ab \ln(n\Delta_G))/n$ for a large enough constant a and a parameter b.

For every center x, we use the approximate reachability data structure of Theorem 7.3 to maintain the graph $R(x)$. The algorithm for computing the path union graph of two centers x and y is as follows:

1. Perform a backward breadth-first search from y in $R(x)$ up to h hops, which visits every node v such that $\text{dist}_{R(x)}(v, y) \leq h$. Set $\delta(v, y) := \text{dist}_{R(x)}(v, y)$ for every visited node v and let $H(x, y)$ be the subgraph of $R(x)$ consisting of all visited nodes and edges.

2. Perform a forward breadth-first search from x in $H(x, y)$ up to h hops, which visits every node v such that $\text{dist}_{H(x,y)}(x, v) \leq h$. Set $\delta(x, v) := \text{dist}_{H(x,y)}(x, v)$ for every visited node v and let $Q(x, y)$ be the set of visited nodes.

3. Set $\mathcal{Q}(x, y)$ to be the subgraph of G induced by the nodes in $Q(x, y)$

4. Tell the approximate reachability data structure of x to remove $|H(x, y)| - n/b$ nodes from $R(x)$, where $|H(x, y)|$ is the number of nodes in $H(x, y)$.

We claim that the graph $\mathcal{Q}(x, y)$ computed by this algorithm is actually the path union graph $\mathcal{P}(x, y, h, G)$. This is everything we need to show for the correctness of the algorithm.

Lemma 7.7 (Correctness: Path union graph)**.** *The graph $\mathcal{Q}(x, y)$ computed by the algorithm is the path union graph $\mathcal{P}(x, y, h, G)$.*

Proof. For every node v we claim that $v \in \mathcal{P}(x, y, h, G)$ if and only if $\delta(x, v)+\delta(v, y) \leq h$. Remember that, by the definition of the path union graph, v is in $\mathcal{P}(x, y, h, G)$ if and only if $\text{dist}_G(x, v) + \text{dist}_G(v, y) \leq h$. Thus we need to show that $\text{dist}_G(x, v) + \text{dist}_G(v, y) \leq h$ if and only if $\delta(x, v) + \delta(v, y) \leq h$. We first prove the "only if"-direction, which is simpler. Note that $\delta(x, v)$ and $\delta(v, y)$ are h-hops distances in certain subgraphs of G and thus they do not under-estimate the real distance. Thus, we have $\text{dist}_G(x, v) + \text{dist}_G(v, y) \leq \delta(x, v) + \delta(v, y) \leq h$

Now we prove the "if"-direction. Let v be a node in $\mathcal{P}(x, y, h, G)$, i.e., $\text{dist}_G(x, v)+\text{dist}_G(v, y) \leq h$. Thus, v is on a path π of length at most h from x to y in G. Therefore we have, for every node v' on π, $\text{dist}_G(x, v') \leq h$ which implies that v' is in $R(x)$, as $R(x)$ contains all such nodes by Theorem 7.3. Thus, π is a consecutive path of nodes in $R(x)$ and is therefore contained in $R(x)$. As π ends in y and has length at most h, all nodes and edges on π will be visited by the backward breadth-first search from y which means that π is contained in $H(x, y)$. Furthermore, the distance of v to y determined by this search satisfies $\delta(v, y) = \text{dist}_G(v, y)$. As π is contained in $H(x, y)$ and has length at most h, v will be visited by the forward search of x and we also have $\delta(x, v) = \text{dist}_G(x, v)$. By combining the two inequalities we get $\delta(x, v)+\delta(v, y) = \text{dist}_G(x, v)+\text{dist}_G(v, y) \leq h$ as desired. □

Before we provide the running time analysis, we show that we only call the remove operation of the approximate reachability data structure when a sufficient number of nodes can be removed.

Lemma 7.8 (Precondition of remove operation)**.** *There are $|H(x, y)| - n/b$ nodes v such that $\text{dist}_G(s, v) > (h + 1)\lceil \log n \rceil$ whp.*

Proof. We show that the number of nodes v in $H(x, y)$ such that $\text{dist}_G(x, v) \leq (h + 1)\lceil \log n \rceil$ is at most n/b whp. Assume by contradiction that there are more than n/b such nodes. Then one of them, say v, is a hub whp by Lemma 4.16. Note that v, as every other node in $H(x, y)$, satisfies $\text{dist}_G(v, y) \leq h$. This means that $\text{dist}_G(x, v)+\text{dist}_G(v, y) \leq h+(h+1)\lceil \log n \rceil \leq 4h \log n$. But, since the main algorithm is constructing the path union graph from x to y, there cannot be such a hub anymore. This proves that $H(x, y)$ contains at most n/b nodes such that $\text{dist}_G(s, v) \leq (h+1)\lceil \log n \rceil$ whp. Thus, the number of nodes in $H(x, y)$ such that $\text{dist}_G(s, v) > (h + 1)\lceil \log n \rceil$ is at least $|H(x, y)| - n/b$ whp. □

In the running time analysis we will amortize over the construction of all path union graphs $\mathcal{P}(x, y, h, G)$ for a fixed center x.

Lemma 7.9 (Running time)**.** *The total time needed for constructing all path union graphs is* $\widetilde{O}(cn^2 + c^2n^2/b^2)$ *in expectation.*

Proof. Maintaining the graph $R(x)$ with the approximate reachability data structures takes time $O(n^2)$ whp for every center x by Theorem 7.3 and Lemma 7.8. By setting the constant in the sampling probability of the nodes large enough, we can maintain it in time $O(n^2)$ in expectation. Thus, the total time of this task for all centers is $\widetilde{O}(cn^2)$ as there are $\widetilde{O}(c)$ centers. Fixing some center x, we now bound the total cost of all backward breadth-first searches we perform for some center y in constructing $\mathcal{Q}(x, y)$. Note that the time needed for the forward searches will then already be included. After each of these breadth-first searches, the approximate reachability data structure removes at least $|H(x, y)| - n/b$ nodes from $R(x)$. We simply bound the cost of visiting nodes that are not removed and the edges between them by $O(n^2/b^2)$, which adds the term $\widetilde{O}(c^2n^2/b^2)$ to the running time. For all nodes that are removed from $R(x)$, also their incoming and outgoing edges are removed. Thus, these edges will never be visited again when we construct $\mathcal{Q}(x, y')$ for some other center $y' \neq y$. As each of these edges is only visited once over all updates in the graph, this adds a cost of $O(m)$ for the center x and thus $\widetilde{O}(cm)$ for all centers, which is dominated by the cost $\widetilde{O}(cn^2)$. □

7.3 Main Algorithm

The basic algorithmic framework is similar to the algorithm of Theorem 5.1. Our algorithm has two parameters h and b, which we will set in an optimal way later on. It also sets $c = 2n/h$. At the initialization the algorithm determines a set of hubs and a set of centers. We sample each node with probability $(ab \ln(n\Delta_G))/n$ for a large enough constant a and use the sampled nodes as hubs. We also sample each node with probability $(ac \log n)/n$ for a large enough constant a and use the sampled nodes as centers. Besides these sampled nodes, also the source s and the sink t are used as centers.

Using an auxiliary algorithm (see Algorithm 4) we will always know whether the distance between two centers is small. The auxiliary algorithm maintains $\text{dist}_G^{4h \log n}(x, v)$ and $\text{dist}_G^{4h \log n}(v, y)$ for every hub v. (Note that the additional factor of $4 \log n$ comes from the fact that we want to use AR data structures for the centers). As we consider unweighted graphs, we can do this by maintaining an ES-tree up to depth $4h \log n$ for every hub (see Lemma 4.6). The auxiliary algorithm always checks whether there is a hub v such that $\text{dist}_G^{4h \log n}(x, v) + \text{dist}_G^{4h \log n}(v, y) \leq 4h \log n$. As long as such a hub exists, we know that $\text{dist}_G(x, y) \leq 4h \log n$. If no such hub exists anymore, the auxiliary algorithm will construct the path union graph $\mathcal{P}(x, y, h, G)$. For this step we use the new algorithm we described in Section 7.2 using an AR data structure for every center. As soon as we have constructed this path union graph, we store it in a graph $\mathcal{Q}(x, y, h, G)$. The graph $\mathcal{Q}(x, y, h, G)$ undergoes the same edge deletions as G. Using an ES-tree we maintain $\text{dist}^h_{\mathcal{Q}(x,y)}(x, y)$. As $\mathcal{Q}(x, y)$ contains all paths from x to y of length at most h, we have $\text{dist}^h_{\mathcal{Q}(x,y)}(x, y) = \text{dist}^h_G(x, y)$.

Using this auxiliary algorithm, we can maintain the following unweighted graph $\mathcal{C}$: The nodes of $\mathcal{C}$ are the centers and there is an edge (x, y) in $\mathcal{C}$ if and only if one of the following two conditions is fulfilled.

1. There is a hub v such that $\text{dist}^h_G(x, v) + \text{dist}^h_G(v, y) \leq (4 \log n)h$
2. $\text{dist}_G(x, y) \leq h$

Note that we can easily maintain the edges of $\mathcal{C}$. The first condition is equivalent to $j(x, y) \leq |B|$. If $j(x, y) = |B| + 1$, the second condition is equivalent to $\text{dist}^h_{\mathcal{Q}(x,y)}(x, y) \leq h$.

Algorithm 4: Auxiliary algorithm for maintaining approximate distance of centers

Determine set of hubs $B = \{b_1, \ldots, b_{|B|}\}$ and set of centers C by random sampling
For every hub $v \in B\}$ maintain $\text{dist}^h_G(x, v)$ and $\text{dist}^h_G(v, y)$ using ES-trees
Set $j(x, y) = 1$ for all pairs of centers x and y
After every deletion in G call `update()`

```
update()
    foreach pair of centers x and y do
        if j(x,y) < |B| + 1 then
            while j(x,y) ≤ |B| + 1 and dist^h_G(x, b_{j(x,y)}) + dist^h_G(b_{j(x,y)}, y) > (4 log n)h do
                j(x,y) ← j(x,y) + 1
            if j(x,y) = |B| + 1 then
                Initialize Q(x,y) as the path union graph P(x,y,h,G) using the algorithm of
                Section 7.2
                Start maintaining Q(x,y) by making the same future deletions as in G
                Start maintaining dist^h_{Q(x,y)}(x,y) using ES-tree
```

Each edge (x, y) in $\mathcal{C}$ represents a path from x to y of length at most $(4 \log n)h$. We can show that $\mathcal{C}$ provides a relatively good approximation of the distances in G if we multiply the distance in $\mathcal{C}$ by $4h \log n$.

Lemma 7.10. *For all centers x and y we have* $\text{dist}_G(x, y) \leq (4h \log n)\, \text{dist}_{\mathcal{C}}(x, y) \leq (8 \log n)\, \text{dist}_G(x, y) + 4h \log n$.

Proof. The inequality $\text{dist}_G(x, y) \leq (4h \log n)\, \text{dist}_{\mathcal{C}}(x, y)$ follows from the fact that an edge (x', y') in $\mathcal{C}$ can only exist for centers x' and y' if $\text{dist}_G(x', y') \leq (4 \log n)h$. We now show that

$$(4h \log n)\, \text{dist}_{\mathcal{C}}(x, y) \leq (8 \log n)\, \text{dist}_G(x, y) + 4h \log n\,.$$

Consider a shortest path π from x to y in G and divide it into k subpaths $\pi_1, \ldots, \pi_k$ such that, for $1 \leq i \leq k-1$, π_i has length exactly h and π_k has length at most h. Now consider the *last* $h/2$ nodes of every subpath π_i for $1 \leq i \leq k-1$. By Lemma 4.16 and the random choice of the centers all these sets of nodes contain some center c_i whp. Note that x and y are also included in the set of centers. We set $c_0 = x$ and $c_k = y$. Since $\text{dist}_G(c_i, c_{i+1}) \leq h$ for all $0 \leq i \leq k-1$, the graph $\mathcal{C}$ contains an edge (c_i, c_{i+1}) for all $0 \leq i \leq k-1$. Furthermore, $\text{dist}_G(c_i, c_{i+1}) \geq h/2$ for all $0 \leq i \leq k-2$. Thus, we have replaced each subpath π_i (for $1 \leq i \leq k-1$), which has length at least $h/2$, by an edge in $\mathcal{C}$. It follows that $\text{dist}_{\mathcal{C}}(c_0, c_{k-1}) \leq \text{dist}_G(c_0, c_{k-1})/(h/2)$ and thus $(4h \log n)\, \text{dist}_{\mathcal{C}}(c_0, c_{k-1}) \leq (8 \log n)\, \text{dist}_G(c_0, c_{k-1})$ (by multiplying both sides of the inequality with $4h \log n$). Since $\text{dist}_G(c_{k-1}, t) \leq h$, $\mathcal{C}$ also contains the edge (c_{k-1}, t). Therefore we have $(4h \log n)\, \text{dist}_{\mathcal{C}}(x, y) = (4h \log n)\, \text{dist}_{\mathcal{C}}(c_0, c_k) \leq (8 \log n)\, \text{dist}_G(x, y) + 4h \log n$. □

Note that $\mathcal{C}$ only undergoes deletions but no insertions. Thus, we can maintain the distance $\text{dist}_{\mathcal{C}}(s, x)$, for every center $x \in C$, in the unweighted graph $\mathcal{C}$ by using an ES-tree from s (with full depth $\widetilde{O}(c)$) (see Lemma 4.6). If $\text{dist}_G(s, x) \geq h$, we may apply the previous lemma and get

$$\begin{aligned}(4h \log n)\, \text{dist}_{\mathcal{C}}(s, x) &\leq (8 \log n)\, \text{dist}_G(s, x) + 4h \log n \\ &\leq (8 \log n)\, \text{dist}_G(s, x) + (4 \log n)\, \text{dist}_G(s, x) \\ &= (12 \log n)\, \text{dist}_G(s, x)\,.\end{aligned}$$

To deal with the case $\text{dist}_G(s,x) < h$, we simply maintain $\text{dist}^h_G(s,x)$ *exactly*, for every center $x \in C$, by using an ES-tree from s up to depth h in G. This is not too inefficient because the depth is relatively small. To fulfill the desired approximation guarantee, the algorithm then simply returns $\delta(s,x) := \min(\text{dist}_{\mathcal{C}}(s,x), \text{dist}^h_G(s,x))$.

Lemma 7.11. *The algorithm described above runs in time* $\widetilde{O}(m^{1/2}n^{7/4})$.

Proof. The number of hubs and centers are $\widetilde{O}(b)$ and $\widetilde{O}(c)$ in expectation, respectively. Maintaining the ES-trees up to depth $O(h \log n)$ for the hubs takes time $\widetilde{O}(bmh) = \widetilde{O}(bmn/c)$ by Lemma 4.6. Construction the path union graphs between all centers takes time $\widetilde{O}(cn^2 + c^2n^2/b^2)$ by Lemma 7.9. Maintaining the ES-trees up to depth h in the path union graphs takes time $\widetilde{O}(c^2(n^2/b^2)h) = \widetilde{O}(n^3c/b^2)$ by Lemma 4.6 because each path union graph has at most n/b nodes whp. Maintaining the ES-tree from s up to depth c in $\mathcal{C}$ (which has at most $\widetilde{O}(c^2)$ edges) takes time $\widetilde{O}(c^3)$. Maintaining the ES-tree from s up to depth h in G takes time $O(mh) = O(mn/c)$. Note that $mn/c \leq bmn/c$. Furthermore, $c^3 \leq cn^2$ and $c^2n^2/b^2 \leq n^3c/b^2$ since $c \leq n$. By balancing the terms bmn/c, cn^2, and n^3c/b^2 we arrive at the parameter choice $b = n^{1/2}$ and $c = m^{1/2}/n^{1/4}$, which gives the running time stated above. □

Similar to Corollary 5.12 we can use our algorithm to solve a problem that is slightly more general than approximate stSP.

Corollary 7.12. *Given an unweighted directed graph undergoing edge deletions, an approximation parameter* $0 < \epsilon \leq 1$*, a source node* s*, and a set of sinks* S *of size* k*, we can, for every node* $x \in S$*, maintain a distance estimate* $\delta(s,x)$ *such that* $\text{dist}(s,x) \leq \delta(s,x) \leq O((\log n)\,\text{dist}(s,x))$*. This data structure has constant query time and a total update time of* $\widetilde{O}(k^2n + k^{2/3}m^{1/6}n^{23/12} + m^{1/2}n^{7/4})$.

Proof sketch. We set $c = k + n/h$. Let C be a set of centers of size $\widetilde{O}(c)$ consisting of the nodes in S and a set of nodes sampled uniformly at random with probability $(a \ln(n\Delta_G))/h$ for a large enough constant a. Run the algorithm described above with this set of centers. The same running time analysis a above reveals that the dominant terms in the running time are bmh, kn^2, and k^2hn^2/b^2. We balance bmh and k^2hn^2/b^2 by setting $b = c^{2/3}n^{2/3}/m^{1/3}$. Thus, the total update time is

$$\begin{aligned}\widetilde{O}(cn^2 + c^{2/3}hm^{2/3}n^{2/3}) &= \widetilde{O}((k+n/h)n^2 + (k+n/h)^{2/3}hm^{2/3}n^{2/3}) \\ &= \widetilde{O}(kn^2 + n^3/h + k^{2/3}hm^{2/3}n^{2/3} + h^{1/3}m^{2/3}n^{4/3})\,.\end{aligned}$$

We balance n^3/h and $h^{1/3}m^{2/3}n^{4/3}$ by setting $h = n^{5/4}/m^{1/2}$, which gives a running time of

$$\widetilde{O}(kn^2 + k^{2/3}m^{1/6}n^{23/12} + m^{1/2}n^{7/4})\,.$$

□

Using the same proof as in Corollary 5.14, we can get faster algorithms for graphs of medium density.

Corollary 7.13. *Given an unweighted directed graph undergoing edge deletions and set of nodes* S *of size* k*, we can, for all nodes* x *and* y *in* S*, maintain a distance estimate* $\delta(x,y)$ *such that* $\text{dist}(x,y) \leq \delta(x,y) \leq O((\log n)\,\text{dist}(s,x))$*. This data structure has constant query time and a total update time of* $\widetilde{O}(m^{25/33}n^{4/3})$ *or* $\widetilde{O}(m^{191/207}n^{226/207})$

Proof Sketch. Recall from Corollary 5.13 that if we have an algorithm with $\widetilde{O}(m^{\alpha}n^{\beta})$ total update time, then we can get an algorithm with $\widetilde{O}(m^{\alpha'}n^{\beta'})$ where the pair (α', β') is either (i) $\alpha' = 1 - 2/(6\alpha + 3\beta)$ and $\beta' = (8\alpha + 4\beta)/(6\alpha + 3\beta)$ or (ii) $\alpha' = (2(\alpha - 1)/(3(\alpha + \beta)) + 1$ and $\beta' = 2\beta/(3(\alpha + \beta)) + 2/3$. We use the first formula with $\alpha = 1/2$ and $\beta = 7/4$ (from Theorem 7.1), then we get $\alpha' = 25/33$ and $\beta' = 4/3$, thus the first running time. Then we use the second formula with $\alpha = 25/33$ and $\beta = 4/3$ to $\alpha' = 191/207$ and $\beta' = 226/207$, thus the second running time. □

8 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 single-source single-sink shortest path data structure, 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 [3].

Theorem 8.1. *Assume we have the following data structure: given a weighted directed graph undergoing edge deletions and edge weight increases, an approximation parameter $0 < \epsilon \leq 1$, a source node s, and a set of sinks S of size k, we can, for every node $x \in S$, maintain a distance estimate $\delta(s, x)$ such that $\operatorname{dist}(s, x) \leq \delta(s, x) \leq O((\log n)\operatorname{dist}(s, x))$ with constant query time and a total update time of $T(k, m, n)$. Then there exists a $(1 + \epsilon)$-approximate SSSP data structure with constant query time and an expected total update time of $O(T(c \log n, m, n) + mn \log(nW)/(\epsilon c))$ for any $c \leq n$. This SSSP data structure is correct with high probability.*

Proof. Let G denote the graph undergoing edge deletions and edge weight increases. At the initialization we randomly sample each node in G with probability $p = (ac \ln(n\Delta_G))/n$ (for a large enough constant a). We call the sampled nodes sinks. We use the data structure of the assumption to maintain a distance estimate $\delta(s, x)$ for every sink x such that $\operatorname{dist}_G(s, x) \leq \delta(s, x) \leq (1 + \epsilon)\operatorname{dist}_G(s, x)$. Additionally, we maintain a graph G' which consists of the graph G augmented by the following edges: for every sink x we add an edge (s, x) of weight $w'(s, x) = \delta(s, x)$ (these edges are the *shortcut edges*). We maintain G' by updating the weights of these edges every time a distance estimate $\delta(s, x)$ changes its value. On G' we use Bernstein's $(1 + \epsilon)$-approximate SSSP data structure [3] with source s and hop count $h = n/c$ (see Lemma 4.9). This data structure maintains a distance estimate $\delta'(s, v)$ such that $\operatorname{dist}_{G'}(s, v) \leq \delta'(s, v) \leq (1 + \epsilon)\operatorname{dist}^h_{G'}(s, v)$.

First, observe that $\operatorname{dist}_G(s, v) = \operatorname{dist}_{G'}(s, v)$ as the new shortcut edges never under-estimate the true distance in G. We now claim that $\operatorname{dist}^h_{G'}(s, v) \leq (1 + \epsilon)\operatorname{dist}_G(s, v)$. If $\operatorname{dist}_G(s, v) = \infty$, then the claim is trivially true. Otherwise let π be a 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 $\operatorname{dist}^h_{G'}(s, v) = \operatorname{dist}_G(s, v)$. If π has more then h edges, then we know that the set of nodes consisting of the last $h - 1$ edges of π contains a sink x whp by Lemma 4.16 (note that $p = (a \ln(n\Delta_G))/h$). Thus, the graph G' contains a shortcut edge (s, x) with weight $w'(s, x) = \delta(s, x)$. Now let π' be the path from s to v that starts with this edge (s, x) and the follows the path π from x to v. Clearly, the path π' has at most h hops and is contained in G'. As the weight of π' is $w'(s, x) + \operatorname{dist}_G(x, v)$ we get

$$\begin{aligned}\operatorname{dist}^h_{G'}(s, v) = w'(s, v) + \operatorname{dist}_G(x, v) &= \delta(s, x) + \operatorname{dist}_G(x, v)\\ &\leq (1 + \epsilon)\operatorname{dist}_G(s, x) + \operatorname{dist}_G(x, v) \leq (1 + \epsilon)\operatorname{dist}_G(s, v)\,.\end{aligned}$$

Putting everything together, we get that the distance estimate $\delta'(s, v)$ fulfills

$$\operatorname{dist}_G(s, v) = \operatorname{dist}_{G'}(s, v) \leq \delta'(s, v) \leq (1 + \epsilon)\operatorname{dist}^h_{G'}(s, v) \leq (1 + \epsilon)^2\operatorname{dist}_G(s, v) \leq (1 + 3\epsilon)\operatorname{dist}^h_{G'}(s, v)$$

By using $\epsilon' = \epsilon/3$ we obtain a $(1+\epsilon)$-approximation of distances from s in G.

Finally, we argue about the running time. Our running time has two parts. (1) Bernstein's $(1+\epsilon)$-approximate SSSP data structure has constant query time and a total update time of $O(mh \log(nW)/\epsilon)$ (see Lemma 4.9). Thus, our data structure also has constant query time and by our choice of $h = n/c$, our total update time contains the term $mn \log(nW)/(\epsilon c)$. (2) Furthermore, we have we have $O(c \log n)$ sinks in expectation. Thus, the expected total update time of the data structure from our assumption is $T(c \log n, m, n)$. □

Using this reduction, we can now summarize our results for decremental approximate shortest paths as follows.

Theorem 8.2 (Summary of Results)**.** *We obtain the following data structures (remember that in weighted directed graphs Δ_G is the number of edge deletions and edge weight increases):*

(1) $(1+\epsilon)$*-approximate SSSP data structure in weighted directed graphs with total update time*

$$\widetilde{O}(m^{5/4}n^{5/8}(\log W)^2/\epsilon^2 + \Delta_G)$$

(Follows from Corollary 6.9 using $c = n^{3/8}/m^{1/4}$. This is $o(mn(\log W)^2/\epsilon^2 + \Delta_G)$ if $m = o(n^{3/2})$.)

(2) $(1+\epsilon)$*-approximate SSSP in weighted directed graphs with total update time*

$$\widetilde{O}(m^{25/27}n^{206/189}(\log W)^3/\epsilon^4 + \Delta_G)$$

(Follows from Corollary 5.14 using $c = n^{103/189}/m^{1/27}$. This is $o(mn(\log W)^2/\epsilon^3 + \Delta_G)$ if $m = \omega(n^{17/14})$. This improves (1) when $m = \omega(n^{257/182})$, e.g. when $m \geq n^{1.42}$.)

(3) $(1+\epsilon)$*-approximate SSSP in weighted directed graphs with total update time*

$$\widetilde{O}(m^{23/30}n^{4/3}(\log W)^2/\epsilon^3 + \Delta_G)$$

(Follows from Corollary 5.14 using $c = n^{5/9}/m^{2/45}$. This is $o(mn(\log W)^2/\epsilon^3 + \Delta_G)$ if $m = \omega(n^{30/21})$, e.g. $m \geq n^{1.43}$. This improves (2) when $m = \omega(n^{460/301})$, e.g. when $m \geq n^{1.53}$.)

(4) $(1+\epsilon)$*-approximate SSSP in weighted directed graphs with total update time*

$$\widetilde{O}((m^{5/7}n^{10/7}\log W)/\epsilon^2 + \Delta_G)$$

(Follows from Corollary 5.12 using $c = n^{5/7}/m^{1/7}$. This is $o(mn \log W/\epsilon^2 + \Delta_G)$ if $m = \omega(n^{3/2})$. This improves (3) when $m = \omega(n^{20/11})$, e.g. when $m \geq n^{1.82}$.)

(5) $O(\log n)$*-approximate SSSP in unweighted directed graphs with total update time*

$$\widetilde{O}(m^{191/207}n^{226/207})$$

(Follows from Corollary 7.13 using $c = n^{113/207}/m^{8/207}$. This is $o(mn)$ if $m = \omega(n^{19/16})$, e.g. when $m \geq n^{1.19}$. This improves (1) when $m = \omega(n^{283/202})$, e.g. when $m \geq n^{1.41}$.)

(6) $O(\log n)$*-approximate SSSP in unweighted directed graphs with total update time*

$$\widetilde{O}(m^{25/33}n^{4/3})$$

(Follows from Corollary 7.13 using $c = n^{16/27}/m^{2/27}$. This is $o(mn)$ if $m = \omega(n^{11/8})$, e.g. when $m \geq n^{1.38}$. This improves (5) when $m = \omega(n^{275/188})$, e.g. when $m \geq n^{1.47}$.)

(7) $O(\log n)$-approximate SSSP in unweighted directed graphs with total update time

$$\widetilde{O}(m^{1/2}n^{7/4})$$

(Follows from Corollary 7.12 using $c = m^{1/2}/n^{1/4}$. This is $o(mn)$ if $m = \omega(n^{3/2})$. This improves (6) when $m = \omega(n^{55/34})$, e.g. when $m \geq n^{1.62}$.)

All these data structures have constant query time.

9 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 [18], 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 the paper, we will here impose the following technical condition on the single-source reachability data structure: when we update the data structure 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 data structures we have presented so far fulfill this condition.[12]

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 SSR data structure, 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 data structures of w).

We now explain the behavior of the algorithm when a component C decomposes into the new components $C_1, \ldots, C_k$. 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 SSR data structures. There is one notable exception: 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 data structures 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 $C_1, \ldots, C_k$. 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 $C_1, \ldots, C_{j-1}, C_{j+1}, \ldots C_k$.

[12] Remember that our algorithms *explicitly* store an estimate of the distance from the source to v for every node v. In the case of SSR, this distance estimate is finite if the node is reachable from the source and ∞ otherwise. After every deletion we simply have to return all nodes for which the distance estimate changes to ∞. This does not increase the update time as we have to access the distance estimates of these nodes anyway.

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 $C_1, \ldots, C_{j-1}, C_{j+1}, \ldots C_k$. Note that the sets A and B are returned by the update-procedure of the SSR data structures 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.

We also consider a second variant of the algorithm where the representative of a connected component C is chosen by a random edge. To be precise, we pick an edge (u, v) among all outgoing edges of nodes in C uniformly at random. The representative is the tail u of the chosen edge.

Analysis. The correctness of the algorithm explained above is immediate. For the running time we will argue that, up to a log factor, it is the same as the running time of the SSR data structure. When the running time of SSR is of the form $\widetilde{O}(m^\alpha n^\beta)$ (as in the previous sections), 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 SSR data structure. 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 9.1 (From SSR to SCC)**.** *If there is a decremental SSR data structure with constant query time and a total update time of (1) $O(nt(m, n) + m)$ or (2) $O(mt(m, n))$ such that $t(m, n)$ is non-decreasing in m and n[13], then there exists a decremental SCC data structure with constant query time and an expected total update time of (1) $O(nt(m, n) \log n + m \log n)$ or (2) $O(mt(m, n) \log n)$, respectively.*

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 in the initial graph (before any deletions) of the nodes in C_i. 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 $O(n_j t(m_j, n_j))$ for initializing and maintaining the SSR data structure of the new representative of C_j. Second, we have to pay for the cost of 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 data structure of w outputs A and the outgoing SSR data structure of w outputs B. Thus, the cost of computing $A \cup B$ can be charged

[13]The technical assumption that $t(m, n)$ is non-decreasing in m and n is natural as usually the running time of an algorithm does not improve with increasing problem size.

to the reachability data structures 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 [20], we can compute the strongly connected components of G' in the same running time.

Let us now proceed with analyzing case (1) where the running time of the SSR data structure is $O(nt(m,n)+m)$. Here the algorithm uses random nodes as representatives. By the random choice of 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

$$\sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} (n_j t(m_j, n_j) + m'_j) = \sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} n_j t(m_j, n_j) + \sum_{i=1}^{k} \frac{n_i}{n_0} \sum_{j \neq i} m'_j \,.$$

We analyze each of these terms individually.

Consider first the cost of $O(\sum_{i=1}^{k} n_i/n_0 \sum_{j \neq i} n_j t(m_j, n_j))$. For every pair i, j such that $i \neq j$ we have to pay a cost of $O(n_i n_j (t(m_j, n_j) + n)/n_0)$. 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 proportinal to

$$\sum_{j \neq i} \frac{n_i n_j (t(m_j, n_j) + n)}{n_0} \leq n_i (t(m,n) + n) \frac{\sum_{j \neq i} n_j}{n_0} \leq n_i (t(m,n) + n) \,.$$

We share this cost equally among the nodes in C_i and thus charge a cost of $O(t(m,n)+n)$ 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 of the algorithm is $O(nt(m,n) \log n + n^2 \log n)$.

Consider now the cost of $O(\sum_{i=1}^{k} n_i/n_0 \sum_{j \neq i} m'_j)$. 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 can argue that in this way every edge is charged only $O(\log n)$. 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 i-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 $O(\log n)$. Thus, we charge $O(\log n)$ to every edge of the initial graph, which gives a running time bound of $O(m \log n)$.

We now consider case (2) where the total update time of the SSR data structure is $O(mt(m,n))$. Here we use the variant of the algorithm where the representatives are chosen using random edges.

Therefore the expected cost of the decomposition of C_0 is

$$\sum_{i=1}^{k} \frac{m'_i}{m'_0} \sum_{j \neq i} (m_j t(m_j, n_j) + m'_j)$$

which means that for every pair i, j such that $i \neq j$ we have to pay $O(m'_i m'_j t(m_j, n_j)/m'_0)$ (note that $t(m_j, n_j) = \Omega(1)$). If $m'_i \leq m'_j$, we charge this cost to C_i, otherwise we charge it to C_j (again the smaller component is charged). For the component to which we charge we know that the sum of the degrees of its nodes is at most $m'_0/2$. For a fixed component i the total charge is of order

$$\sum_{j \neq i} \frac{m'_i m'_j t(m_j, n_j)}{m'_0} \leq m'_i t(m, n) \frac{\sum_{j \neq i} m'_j}{m'_0} \leq m'_i t(m, n)\,.$$

Thus, we can charge $O(t(m, n))$ to every outgoing edge of a node in C_i. As every edge is charged at most $\log m$ times, the total update time of the algorithm is $O(mt(m, n) \log n)$.

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 [20]. Furthermore, each component starts maintaining one SSR data structure and we have to pay for the total update time of these data structures. In case (1) this time is proportional to $\sum_{i=1}^{k}(n_i t(m_i, n_i) + m_i) \leq t(m, n) \sum_{i=1}^{k} n_i + \sum_{i=1}^{k} m_i \leq nt(m, n) + m$. Similarly, the initialization time of case (2) is bounded by $\sum_{i=1}^{k} m_i t(m_i, n_i) \leq t(m, n) \sum_{i=1}^{k} m_i \leq mt(m, n)$. In both cases the running time is within the stated upper bounds. □

Using the reductions above, it follows immediately that our decremental SSR data structures listed in Theorem 8.2 lead to improved decremental SCC data structures.

Corollary 9.2 (Summary of Results)**.** *There are data structures for maintaining strongly connected components in directed graphs undergoing edge deletions with constant query time and the following update times:*

- $\widetilde{O}(m^{5/4} n^{5/8})$
- $\widetilde{O}(m^{191/207} n^{226/207})$
- $\widetilde{O}(m^{25/33} n^{4/3})$
- $\widetilde{O}(m^{1/2} n^{7/4})$.

References

[1] A. Abboud and V. Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. *CoRR*, abs/1402.0054, 2014. 4

[2] A. Bernstein. Fully Dynamic $(2 + \epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time. In *FOCS*, pages 693–702, 2009. 16

[3] A. Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. In *STOC*, pages 725–734, 2013. 1, 10, 11, 16, 26, 41

[4] K. Chatterjee and M. Henzinger. An $O(n^2)$ time algorithm for alternating Büchi games. In *SODA*, pages 1386–1399, 2012. 13

[5] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. CRC Press, 2005. 6, 7, 8

[6] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981. 1, 5, 16

[7] M. Henzinger, S. Krinninger, and D. Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *SODA*, 2014. 1

[8] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *FOCS*, pages 664–672, 1995. 1, 16

[9] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *STOC*, pages 519–527, 1995. 1

[10] M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999. Announced at SODA, 1996. 13

[11] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. Announced at STOC, 1998. 1

[12] B. M. Kapron, V. King, and B. Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *SODA*, pages 1131–1142, 2013. 1

[13] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *STOC*, pages 81–91, 1999. 1, 16

[14] V. King. Fully dynamic transitive closure. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*. Springer, 2008. 1

[15] J. Lacki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Transactions on Algorithms*, 9(3):27, 2013. 1

[16] D. Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. Manuscript, 2013. 16

[17] L. Roditty. Decremental maintenance of strongly connected components. In *SODA*, pages 1143–1150, 2013. 1

[18] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, 37(5):1455–1471, 2008. Announced at FOCS, 2002. 1, 11, 12, 43

[19] L. Roditty and U. Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011. Announced at ESA, 2004. 4

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. Announced at SWA (FOCS), 1971. 45, 46

[21] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *STOC*, pages 343–350, 2000. 1

[22] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991. 6, 7, 8, 18

[23] V. Vassilevska Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *FOCS*, pages 645–654, 2010. 4