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 with a total update time of $O(\min(m^{7/6}n^{2/3+o(1)}, m^{3/4}n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$ for SSR and $(1+\epsilon)$-approximate SSSP if the edge weights are integers from 1 to $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c. We also extend our algorithm to achieve roughly the same running time for Strongly Connected Components (SCC), improving the algorithm of Roditty and Zwick [FOCS 2002]. Our algorithm is most efficient for sparse and dense graphs. When $m = \Theta(n)$ its running time is $O(n^{1+5/6+o(1)})$ and when $m = \Theta(n^2)$ its running time is $O(n^{2+3/4+o(1)})$. For SSR we also obtain an algorithm that is faster for dense graphs and has a total update time of $O(m^{2/3}n^{4/3+o(1)} + m^{3/7}n^{12/7+o(1)})$ which is $O(n^{2+2/3})$ when $m = \Theta(n^2)$. All our algorithms have constant query time in the worst case and are correct with high probability against an oblivious adversary.

Contents

1   Introduction

Dynamic graph algorithms are data structures that maintain a property of a dynamically changing graph, supporting both update and query operations on the graph. In *undirected* graphs fundamental properties such as the connected, 2-edge connected, and 2-vertex connected components as well as a minimum spanning forest can be maintained *very quickly*, i.e., in polylogarithmic time per operation [HK99, HLT01, Tho00, KKM13], 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 [HKN14a, HKN14b].

These problems when considered on *directed* graphs, however, become much harder. Consider, for example, a counterpart of the connectivity problem where we want to know whether there is a directed path from a node u to a node v, i.e., whether u can *reach* v. In fact, consider a very special case where we want to maintain whether a *fixed* node s can reach any node v under *edge deletions only*. This problem is called *single-source reachability* (SSR) in the *decremental* setting. It is one of the simplest, oldest, yet most useful dynamic graph problems. It was used as a subroutine for solving other dynamic graph problems such as transitive closure [HK95, Kin99, BHS07, RZ08] and strongly connected components (SCC) [RZ08]. Furthermore it is a special case of decremental (approximate) single-source shortest paths (SSSP) [ES81, BR11, HKN14b], which in turn was used as a subroutine for solving dynamic all-pairs shortest paths [Kin99, BHS07, BHS03, RZ08, RZ11, Ber13, HKN13, ACT14]. Yet, no algorithm with sublinear update time was known for this problem. This was also the case for decremental *s-t reachability* (stR), where we want to maintain whether the node s can reach the node t, and *s-t shortest path* (stSP), where we want to maintain the distance from s to t.

Related Work.   To the best of our knowledge, the first non-trivial algorithm was published in 1971 as part of Dinitz's celebrated maximum flow algorithm; this algorithm takes $O(mn)$ total update time to solve decremental stSP and stR (see [Din06] for details). In other words, it takes 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. Independently from Dinitz, Even and Shiloach [ES81] developed an algorithm with the same total update time for SSR and SSSP[1]. This algorithm remains the fastest prior to our work. For directed *acyclic* graphs, Italiano [Ita88] gave a decremental algorithm with a total update time of $O(m)$. For the *incremental* version of the problem, where we only allow insertions of edges, a total update time of $O(m)$ is sufficient in general directed graphs [Ita86]. For the *fully dynamic* version of the problem, where both insertions and deletions of edges are allowed, Sankowski obtained an algorithm with a worst-case running time of $O(n^{1.575})$ *per update*, resulting in a total update time of $O(\Delta n^{1.575})$, where Δ is the number of updates.[2]

King [Kin99] showed how to extend the algorithm of Even and Shiloach to *weighted* graphs, giving the first decremental single-source shortest path algorithm with total update

---

[1]It was actually published for undirected graphs and it was observed by Henzinger and King [HK95] that it can be easily adapted to work for directed graphs.

[2]Sankowski's worst-case update time for the fully dynamic single-source single-sink reachability (stR) problem is $O(n^{1.495})$.

time $O(mnW)$, where W is the maximum edge weight (and all edge weights are positive integers)[3]. Using weight rounding [RT87, Coh98, Zwi02, Ber09, Mad10, Ber13] this can be turned into a $(1 + \epsilon)$-approximate single-source shortest paths algorithm with total update time $\tilde{O}(mn \log W)$, where the $\tilde{O}(\cdot)$ notation hides factors polylogarithmic in n. The situation is similar for decremental strongly connected components: The fastest decremental SCC algorithms take total update time $O(mn)$ ([RZ08, Lac13, Rod13]). 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 [Kin08, Lac13, RZ08].

Our Results. We improve the previous $O(mn)$-time algorithms for decremental SSR, approximate SSSP, and SCC in directed graphs. We also give algorithms for stR and approximate stSP. We summarize our results in the following theorem. In Figure 1 we compare the running times of our new algorithms with the previous solution by Even and Shiloach for different densities of the graph and Table 1 summarizes our results.

Theorem 1.1. *There exist decremental algorithms for reachability and shortest path problems in directed graphs with the following total update times:*

- *SSR and SCC:* $\tilde{O}(\min(m^{7/6}n^{2/3}, m^{3/4}n^{5/4+o(1)}, m^{2/3}n^{4/3+o(1)} + m^{3/7}n^{12/7+o(1)}))$
  $= O(mn^{9/10+o(1)})$.

- *stR:* $\tilde{O}(\min(m^{5/4}n^{1/2}, m^{2/3}n^{4/3+o(1)}) = O(mn^{6/7+o(1)})$.

- $(1 + \epsilon)$-*approximate SSSP:* $O(\min(m^{7/6}n^{2/3+o(1)}, m^{3/4}n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$.

- $(1 + \epsilon)$-*approximate stSP:* $O(\min(m^{5/4}n^{1/2+o(1)}, m^{2/3}n^{4/3+o(1)}) = O(mn^{6/7+o(1)})$.

The algorithms are correct with high probability against an oblivious adversary and have constant query time. The total update time of the SCC algorithms is only in expectation. Our algorithms can maintain $(1 + \epsilon)$-approximate shortest paths in graphs with positive integer edge weights from 1 to W if $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for any constant c.

Discussion. Our main result are dynamic algorithms with constant query time and sublinear update time for single source reachability and $(1 + \epsilon)$-approximate single source shortest paths in directed graphs undergoing edge deletions. There is some evidence that it is hard to generalize our results in the following ways.

- *(All pairs vs. single source)* The naive algorithm for computing all-pairs reachability (also called transitive closure) in a directed graph takes time $O(mn)$ even in the *static setting*. To date no combinatorial algorithm (not relying on fast matrix multiplication) is known that gives a polynomial improvement over this running time. Thus, unless there is a major breakthrough for static transitive closure, we cannot hope for a combinatorial algorithm for decremental all-pairs reachability with a total update time of $o(mn)$ and small query time. Further, it was recently shown [HKN+15] that no algorithm (not even a non-combinatorial one) can solve decremental all-pairs

---

[3]The total update time is actually $O(md)$, where d is the maximum distance, which could be $\Theta(nW)$.
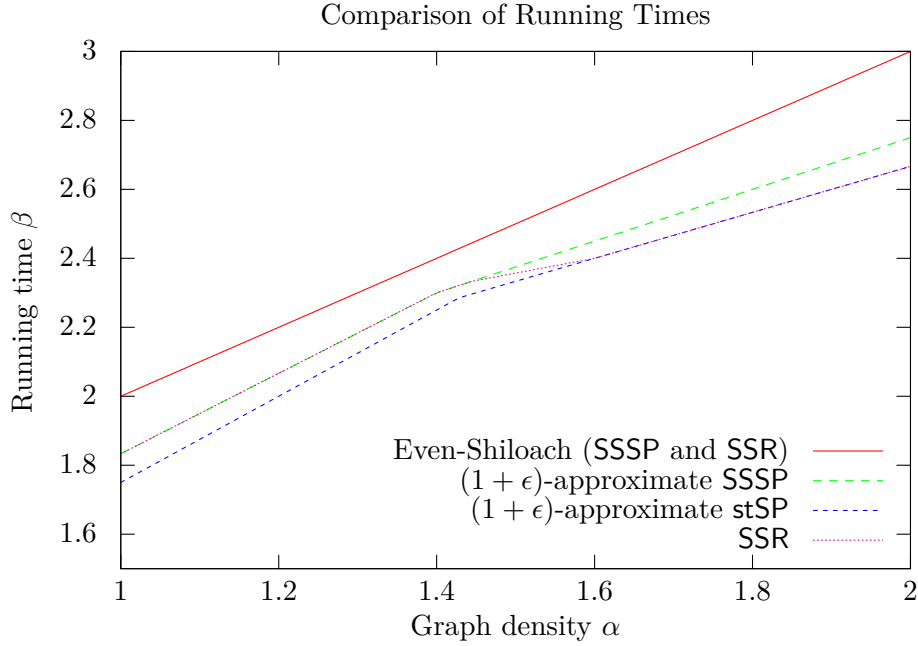
Figure 1: Running times of our decremental reachability and approximate shortest paths algorithms as a function of the density of the initial graph in comparison to the Even-Shiloach algorithm with total update time $O(mn)$. A point (α, β) in this diagram means that for a graph with $m = \Theta(n^\alpha)$ the algorithm has a running time of $O(n^{\beta+o(1)})$.

Problems	Running Time	$o(mn)$ when	Fastest when
SSR	$\tilde{O}(m^{7/6}n^{2/3})$	$m = o(n^2)$	$n \le m \le n^{7/5}$
	$O(m^{3/4}n^{5/4+o(1)})$	$m = \omega(n)$	$n^{7/5} \le m \le n^{13/9}$
	$O(m^{2/3}n^{4/3+o(1)} + m^{3/7}n^{12/7+o(1)})$	$m = \omega(n^{5/4})$	$n^{13/9} \le m \le n^2$
stSP	$\tilde{O}(m^{5/4}n^{1/2})$	$m = o(n^2)$	$m \le n^{10/7}$
	$O(m^{2/3}n^{4/3+o(1)})$	$m = \omega(n)$	$m \ge n^{10/7}$
$(1+\epsilon)$-approximate SSSP	$O(m^{7/6}n^{2/3+o(1)})$	$m = o(n^2)$	$m \le n^{7/5}$
	$O(m^{3/4}n^{5/4+o(1)})$	$m = \omega(n)$	$m \ge n^{7/5}$
$(1+\epsilon)$-approximate stSP	$O(m^{5/4}n^{1/2+o(1)})$	$m = o(n^2)$	$m \le n^{10/7}$
	$O(m^{2/3}n^{4/3+o(1)})$	$m = \omega(n)$	$m \ge n^{10/7}$

Table 1: Summary of our algorithms. The running time in the second column is the total update time. For $(1+\epsilon)$-approximate SSSP and $(1+\epsilon)$-approximate stSP we assume that, for some constant c, $\epsilon \ge 1/\log^c n$ and the largest edge weight is at most $2^{\log^c n}$. 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.

reachability faster, based on an assumption called Online Matrix-Vector Multiplication (OMv).

- *(Worst-case 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 [AVW14] 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 [VWW10], 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 both $o(m)$ falsifies the strong exponential time hypothesis. It might therefore not be possible to deamortize our algorithms.

- *(Approximate vs. exact)* Our SSSP algorithms only provide approximate solutions. It has been observed by Roditty and Zwick [RZ11] that any exact combinatorial SSSP algorithm handling edge deletions that has a total update time of $o(mn)$ and small query time implies a faster combinatorial for Boolean matrix multiplication. After the preliminary version [HKN14c] of this work appeared, Henzinger et al. [HKN$^+$15] showed that $O(mn)$ is essentially the best possible total update time for maintaining exact distances under the assumption that there is no "truly subcubic" algorithm for a problem called online Boolean matrix-vector multiplication. Both of these hardness results apply even for unweighted undirected graphs. Thus, approximation might be necessary to break the $O(mn)$ barrier.

Organization. We first introduce the notation and basic concepts shared by all our algorithms in Section 2. To provide some intuition for our approach we sketch two basic *s-t* reachability algorithms in Section 3. We give the reductions for extending our results to SSSP (and SSR) and SCC in Sections 4.1 and 4.2, respectively. In Section 5 we give a hierarchical *s-t* reachability algorithm, which in turn gives algorithms for SSR and SCC. In Section 6 we extend this idea to $(1 + \epsilon)$-approximate stSP and SSSP. Finally, in Section 7 we give an SSR algorithm with improved total update time for dense graphs. We conclude the paper by discussing open problems.

2 Preliminaries

2.1 Notation

We are given a directed graph G that might be weighted or unweighted. 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 $\Delta(G)$ the number of updates in G, i.e., the number of edge deletions and edge weight increases. We say that a node v is *reachable* from u (or u can reach v) if there is a path from u to v in G. The *distance* $d_G(x, y)$ of a node x to a node y in G is the weight of the shortest path, i.e., the minimum-weight path, from x to y in G. If there is no path from x to y in G we set $d_G(x, y) = \infty$.

Let $G = (V, E)$ be a weighted directed graph, where V is the set of nodes of G and E is the set of edges of G. We denote by n the number of nodes of G and by m the number of

edges of *G before the first update*. We denote the *weight* of an edge (u, v) in G by $w_G(u, v)$. In the weighted case we consider positive integer edge weights and denote the maximum allowed edge weight by W. We assume that $W \leq 2^{\log^c n}$ for some constant c. In unweighted graphs we think of every edge weight as equal to 1. Note that for unweighted graphs we have $\Delta(G) \leq m$ and for weighted graphs we have $\Delta(G) \leq mW$.

For every path $\pi = \langle v_0, v_2, \ldots, v_k \rangle$ we denote its weight in G by $w(\pi, G) = \sum_{0 \leq i \leq k-1} w_G(v_i, v_{i+1})$ and its size in terms of number of edges (also called *hops*) by $|\pi| = k$. We say that the *length* of a path is its weight, but, to avoid ambiguity, we reserve this notion for unweighted graphs where the length of a path is equal to its number of edges. For every integer $h \geq 1$ and all nodes x and y in a directed graph G, the *h-hops distance* $d_G^h(x, y)$ is the minimum weight of all paths from x to y in G consisting of at most h edges. Note that $d_G(u, v) = d_G^n(u, v)$.

For every graph $G = (V, E)$ and every subset of nodes $U \subseteq V$, we define $E[U] = E \cap U^2$ and denote by $G[U] = (U, E[U])$ the *subgraph of G induced by U*. We set the weight of every edge in $G[U]$ equal to its weight in G. For sets of nodes $U \subseteq V$ and $U' \subseteq V$ we define $E[U, U'] = E \cap (U \times U')$, i.e., $E[U, U']$ is the set of edges $(u, v) \in E$ such that $u \in U$ and $v \in U'$.

Our algorithms allow problem-specific update and query operations. The algorithms we design will have constant query time and we will compare them by their *total update time* over all deletions and, in case of weighted graphs, edge weight increases. They are randomized and correct with high probability against an oblivious adversary [BDBK+94] who chooses its sequence of updates and queries before the algorithm starts.[4] We say that an event happens *with high probability (whp)* if it happens with probability at least $1 - 1/n^c$, for some constant c. We assume that arithmetic operations on integers can be performed in constant time. Note that the total update time of decremental algorithms for weighted graphs will be $\Omega(\Delta(G))$, the number of edge deletions and edge weight increases in G. This is unavoidable as the algorithm has to read every update in the graph. In unweighted graphs the dependence on $\Delta(G)$ 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. We use $\tilde{O}$-notation to hide logarithmic factors, i.e., we write $\tilde{O}(t(m, n, W))$ as an abbreviation for $O(t(m, n, W) \log^c n)$ when c is a constant. Similarly, we use the notation $\hat{O}(t(m, n))$ as an abbreviation for $O(t(m, n) \cdot n^{o(1)})$. Whenever one of our algorithms returns an α-approximate result, the approximation factor will be of the form $\alpha = 1 + \epsilon$ where $0 < \epsilon \leq 1$. In this paper we assume that $\epsilon \geq 1/\log^c n$ for some constant c.

2.2 Problem Descriptions

Our goal is to design efficient dynamic algorithms for the following problems.

Definition 2.1 (stR)**.** *A decremental single-source single-sink reachability (stR) algorithm for a directed graph G undergoing edge deletions, a source node s, and a sink node t, maintains the information whether s can reach t. It supports the following operations:*

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

- QUERY()*: Return 'yes' if s can reach t and 'no' otherwise.*

---

[4]In particular, the oblivious adversary does not know the random choices of our algorithm an cannot infer them by choosing its updates and queries adaptively.

Definition 2.2 (SSR). *A* decremental single-source reachability (SSR) algorithm *for a directed graph G undergoing edge deletions and a source node s, maintains the set of nodes reachable from s in G. It supports the following operations:*

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

- QUERY(*v*): *Return 'yes' if s can reach v and 'no' otherwise.*

Definition 2.3 (SCC). *A* decremental strongly connected components (SCC) algorithm *for a directed graph G undergoing edge deletions maintains a set of IDs of the strongly connected components of G and, for every node v, the ID of the strongly connected component that contains v. It supports the following operations:*

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

- QUERY(*v*): *Return the ID of the strongly connected component that contains v.*

Definition 2.4 (α-approximate stSP). *A* decremental α-approximate single-source single-sink shortest path (stSP) algorithm *for a weighted directed graph G undergoing edge deletions and edge weight increases, a source node s, and a sink node t, maintains a distance estimate $\delta(s, t)$ such that $d_G(s, t) \leq \delta(s, t) \leq \alpha d_G(s, t)$. It supports the following operations:*

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

- INCREASE *(u, v, x): Increase the weight of the edge (u, v) to x.*

- QUERY(): *Return the α-approximate distance estimate $\delta(s, t)$.*

Definition 2.5 (α-approximate SSSP). *A decremental α-approximate single-source shortest paths (SSSP) algorithm for a weighted directed graph G undergoing edge deletions and edge weight increases and a source node s, maintains, for every node v, a distance estimate $\delta(s, v)$ such that $d_G(s, v) \leq \delta(s, v) \leq \alpha d_G(s, v)$. It supports the following operations:*

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

- INCREASE *(u, v, x): Increase the weight of the edge (u, v) to x.*

- QUERY(*v*): *Return the α-approximate distance estimate $\delta(s, v)$.*

2.3   Definitions and Basic Properties

In the following we introduce the basic concepts shared by all our algorithms. The first new concept used by our algorithms is the path union of a pair of nodes.

Definition 2.6 (Path Union). *For every directed graph G, every $D \geq 1$, and all pairs of nodes x and y of G, the path union $\mathcal{P}(x, y, D, G) \subseteq V$ is the set containing all nodes that lie on some path π from x to y in G of weight $w_G(\pi) \leq D$.*

Note that if the shortest path from x to y in G has weight at most D, then the subgraph of G induced by the path union $\mathcal{P}(x, y, D, G)$ contains this shortest path. Thus, instead of finding this shortest path in G directly we can also find it in this potentially smaller subgraph of G. In our algorithms we will be able to bound the sizes of the path union subgraphs, which makes them very useful. Observe that for a fixed value of D, the path union can be computed in nearly linear time:

8

Lemma 2.7 (Linear-Time Path Union). *For every directed graph G, every $D \geq 1$ and all pairs of nodes x and y of G, we have $\mathcal{P}(x, y, D, G) = \{v \in V \mid d_G(x, v) + d_G(v, y) \leq D\}$. We can compute this set in time $\tilde{O}(m)$ in weighted graphs and $O(m)$ in unweighted graphs, respectively.*

Proof. Clearly, if $d_G(x, v) + d_G(v, y) \leq D$, then the concatenation of the shortest path from x to v and the shortest path from v to y is a path from x to y of weight at most D and thus $v \in \mathcal{P}(x, y, D, G)$. Conversely, if $v \in \mathcal{P}(x, y, D, G)$, then there is a shortest path π from x to y containing v of weight at most D. Let π_1 and π_2 be the subpaths of π from x to v and from v to y, respectively. Then $d_G(x, v) + d_G(v, y) \leq w_G(\pi_1) + w_G(\pi_2) = w_G(\pi) \leq D$.

Using Dijkstra's algorithm we compute $d_G(x, v)$ and $d_G(v, y)$ for every node $v \in V$ in time $\tilde{O}(m)$. Afterwards, we iterate over all nodes and check for every node v whether $d_G(x, v) + d_G(v, y) \leq D$ in total time $O(n)$. In unweighted graphs we can compute $d_G(x, v)$ and $d_G(v, y)$ for every node $v \in V$ in time $O(m)$ by performing breadth-first search (BFS). □

We further observe in the following that path unions are monotone in the decremental setting, i.e., no nodes will ever be added to any path union for a fixed value of D while the graph undergoes deletions and weight increases This means that once we have computed the path union we can also use it for future versions of the graph, as long as we do not want to consider larger weights of the paths.

Lemma 2.8. *Let G be a directed graph and let G' be the result of deleting some edge from G or increasing the weight of some edge from G. Then for every $D \geq 1$ and all pairs of nodes x and y of G, $\mathcal{P}(x, y, h, G') \subseteq \mathcal{P}(x, y, h, G)$.*

Proof. Let $v \in \mathcal{P}(x, y, h, G')$, i.e., there is a path π from x to y in G' of weight at most D. All edges of π are also contained in G and thus there is a path from x to y in G' of weight $w_G(\pi) = w_{G'}(\pi) \leq D$. Thus, every node of π is contained in $\mathcal{P}(x, y, h, G)$ which implies that all edges of π are contained in $G[\mathcal{P}(x, y, h, G)]$. Therefore $w_{G[\mathcal{P}(x,y,h,G)]}(\pi) = w_G(\pi) \leq D$ and thus every node on π, and in particular v, is contained in $\mathcal{P}(x, y, h, G)$. □

The last property of path unions we will use repeatedly is that we can update them by "computing the path union of the path union". This rebuilding of path unions will be useful later to restrict the overlap of path unions of different pairs of nodes.

Lemma 2.9. *For every directed graph G, every $D \geq 1$, every pair of nodes x and y, and every set of nodes Q such that $\mathcal{P}(x, y, D, G) \subseteq Q \subseteq V$ we have $\mathcal{P}(x, y, D, G[Q]) = \mathcal{P}(x, y, D, G)$.*

Proof. Let $v \in \mathcal{P}(x, y, D, G[Q])$, which means that v lies on a path π from x to y of weight at most D. As $G[Q]$ is a subgraph of G, this path is also contained in G (with the same weight) and thus $v \in \mathcal{P}(x, y, D, G)$.

Now let $v \in \mathcal{P}(x, y, D, G)$, which means that v lies on a path π from x to y of weight at most D. By the assumption $\mathcal{P}(x, y, D, G) \subseteq Q$ every node v' of π is contained in Q and thus π is contained in $G[Q]$ (and has the same weight as in G). As v lies on a path from x to y of weight at most D in G, we have $v \in \mathcal{P}(x, y, D, G[Q])$. □

We now introduce two concepts that are used frequently, also in other dynamic algorithms. The first such concept is the following algorithmic primitive for maintaining shortest paths trees up to bounded depth.

Theorem 2.10 (Even-Shiloach tree [ES81, HK95, Kin99])**.** *There is a decremental algorithm, called* Even-Shiloach tree *(short: ES-tree), that, given a weighted directed graph G undergoing edge deletions and edge weight increases with positive integer edge weights, a source node s, and a parameter $D \geq 1$, maintains a shortest paths tree from s and the corresponding distances up to depth D with total update time $O(mD + \Delta(G))$, i.e., the algorithm maintains $d_G(s,v)$ and the parent of v in the shortest paths tree for every node v such that $d_G(s,v) \leq D$. By reversing the edges of G it can also maintain the distance from v to s for every node v in the same time.*

Note that if we want to use the ES-tree to maintain a full shortest paths tree (i.e., containing all shortest paths from the source), then we have to set D equal to the maximum (finite) distance from the source. If the graph is unweighted, then it is sufficient to set $D = n - 1$ and we can thus maintain a full shortest paths tree with total update time $O(mn)$. If the graph is weighted and W is the maximum edge weight, then the maximum distance might be as large as $(n - 1)W$ and the algorithm then takes time $O(mnW)$, which is less efficient than recomputation from scratch even if $W = n$. Similarly, if we want to maintain a shortest paths tree up to $h \leq n - 1$ hops (containing all shortest paths with at most h edges), the algorithm above takes time $O(mh)$ in unweighted graphs and $O(mhW)$ in weighted graphs. Using a scaling technique [Ber09, Mad10, Ber13], the running time in weighted graphs can be reduced if we allow approximation: we can maintain a $(1 + \epsilon)$-approximate shortest paths tree containing all shortest paths up to h hops in total time $\tilde{O}(mn \log W/\epsilon)$.

The second concept used repeatedly in our algorithms is sampling nodes or edges at random. All our randomized algorithms will be correct whp against an oblivious adversary who fixes its sequence of updates and queries before the algorithm is initialized, revealing its choices to the algorithm one after the other. It is well-known, and exploited by many other algorithms for dynamic shortest paths and reachability, that by sampling a set of nodes with a sufficiently large probability we can guarantee that certain sets of nodes contain at least one of the sampled nodes. To the best of our knowledge, the first use of this technique in graph algorithms goes back to Ullman and Yannakakis [UY91].

Lemma 2.11. *Let $a \geq 1$, 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 $p = \min(x/q, 1)$ where $x = a \ln (kt) + 1$. Then, with high probability (whp), i.e., probability at least $1 - 1/t^a$, the following two properties hold:*

1. For every $1 \leq i \leq k$, the set S_i contains a node of U, i.e., $S_i \cap U \neq \emptyset$.

2. $|U| \leq 3xt/q = O(at \ln (kt)/q)$.

Proof. 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

$$\Pr(E_i) = (1 - p)^{|S_i|} \leq (1 - p)^q = \left(1 - \frac{a \ln (kt) + 1}{q}\right)^q \leq \frac{1}{e^{a \ln (kt) + 1}} = \frac{1}{ek^a t^a} \leq \frac{1}{2k^a t^a}.$$

10

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

$$\Pr(E) = \Pr\left(\bigcup_{1 \leq i \leq k} E_i\right) \leq \sum_{1 \leq i \leq k} \Pr(E_i) \leq \sum_{1 \leq i \leq k} \frac{1}{2k^a t^a} = k\frac{1}{2k^a t^a} \leq \frac{1}{2t^a}\,.$$

To bound the size of U, let, for every $1 \leq i \leq t$, X_i be the random variable that is 1 if the i-th element is sampled and 0 otherwise, i.e., X is 1 with probability p and 0 with probability $1 - p$. The total size of U is given by the random variable $X = \sum_{1 \leq i \leq t} X_i$. Let μ be the expectation of X. By a standard Chernoff bound we have

$$\Pr[X > (1 + \delta)\mu] \leq \frac{1}{e^{\frac{\delta^2}{2+\delta}\mu}}\,.$$

With $\delta = 2$ we get that the probability that U has size more than $3\mu = 3xt/q$ is at most

$$\frac{1}{e^\mu} = \frac{1}{e^{xt/q}} \leq \frac{1}{e^x} = \frac{1}{e^{a\ln kt+1}} \leq \frac{1}{2k^a t^a} \leq \frac{1}{2t^a}\,.$$

By a union bound, the probability that one of property 1 (U is a hitting set) or property 2 (U is small enough) fails is at most $1/t^a$ and thus the claim follows. $\qquad\square$

We now sketch one example of how we intend to use Lemma 2.11 for dynamic graphs. Consider an unweighted graph G undergoing edge deletions. Suppose that we want the following condition to hold for every pair of nodes x and y with probability at least $1 - 1/n$ in *all* versions of G (i.e., initially and after each deletion): if $d_G(x, y) \geq q$, then there is a shortest path from x to y that contains a node in U. We apply Lemma 2.11 as follows. The set T is the set of nodes of G and has size n. The sets $S_1, S_2, \ldots, S_k$ are obtained as follows: for every version of G (i.e., after each deletion) and every pair of nodes x and y such that $d_G(x, y) \geq q$, we define a set S_i that contains the nodes on the first shortest path from x to y (for an arbitrary, but fixed order on the paths). As the graph undergoes edge deletions, there are at most $m \leq n^2$ versions of the graph. Furthermore, there are n^2 pairs of nodes. Therefore we have $k \leq n^4$ such sets. Thus, for the property above to hold with probability $1 - 1/n$, we simply have to sample each node with probability $(\ln kt)/q = (\ln n^5)/q = (5 \ln n)/q$ by Lemma 2.11.

Finally, we explain how to decrease the number of edge weight increases to be considered. Remember that in general the number of edge deletions and edge weight increases can only be bounded by $\Delta(G) = O(mW)$. However, by the reduction below, we may assume without loss of generality that $\Delta(G') = O(m\log_{1+\epsilon} W)$ as we are only interested in $(1 + \epsilon)$-approximate shortest paths. This for example helps us in reducing the sampling probability for applying Lemma 2.11 as the number of versions of G is reduced significantly.

Lemma 2.12. *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*

11

- $d_G(x, y) \leq d_{G'}(x, y) \leq (1 + \epsilon)d_G(x, y)$ *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 π in G is $(1 + \epsilon)$-approximated by the weight of π in G'. As this is also true for a shortest path, we get $d_G(x, y) \leq d_{G'}(x, y) \leq (1 + \epsilon)d_G(x, y)$ 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. $\qquad\square$

3   Algorithm Overview for *s-t* Reachability

In this section, we illustrate our main ideas by giving simple algorithms for the *s-t* reachability problem (stR). At the heart of all our algorithms is a new way of maintaining reachability or distances between some *nearby* pairs of nodes using small path unions (defined in Section 2.3).

3.1   Decremental Bounded-Hop Multi-Pair Reachability

We first give an algorithm for solving the following "restricted" reachability problem: We are given k pairs of sources and sinks (s_1, t_1), ..., (s_k, t_k) and a parameter h. We want to maintain, for each $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$.

In this algorithm we use the following set $B \subseteq V$ of *hubs* of size $\tilde{O}(b)$ (for some parameter b). Let $U \subseteq V$ be a set of nodes obtained by sampling each node independently with probability $ab \ln n / n$ and let $F \subseteq E$ be a set of edges obtained by sampling each edge independently with probability $ab \ln n / m$ from the initial graph (for a large enough constant a). We let B be the set containing U and the endpoints of every edge in F. For every pair of nodes u and v we define the *hub-distance from u to v* as $d_B(u, v) = \min_{x \in B} d_G(u, x) + d_G(x, v)$.

The hubs are useful in combination with the path unions in the following way. Intuitively, instead of directly maintaining whether $d_G(s_i, t_i) \leq h$, we can maintain whether $d_{G[\mathcal{P}(s_i, t_i, h, G)]}(s_i, t_i) \leq h$ since $G[\mathcal{P}(s_i, t_i, h, G)]$ (the subgraph of G induced by $\mathcal{P}(s_i, t_i, h, G)$) contains all s_i-t_i paths of length at most h in G. This could be helpful when $\mathcal{P}(s_i, t_i, h, G)$ is much smaller than V. Our first key idea is the observation that all (s_i, t_i) pairs for which $\mathcal{P}(s_i, t_i, h, G)$ is large can use their paths through a small number of hubs to check whether $d_{G[\mathcal{P}(s_i, t_i, h, G)]}(s_i, t_i) \leq h$ (thus the name "hub"). In particular, consider the following algorithm. We maintain the distance from each s_i to each t_i in two ways.

1. Maintain $d_B(s_i, t_i)$, for each $1 \leq i \leq k$, as long as $d_B(s_i, t_i) \leq h$.

2. Once $d_B(s_i, t_i) > h$, construct $Q(s_i, t_i) = \mathcal{P}(s_i, t_i, h, G)$ and maintain $d_{G[Q(s_i, t_i)]}(s_i, t_i)$ up to value h.

Our algorithm outputs $d_G(s_i, t_i) \leq h$ if and only if either $d_B(s_i, t_i) \leq h$ or $d_{G[Q(s_i, t_i)]}(s_i, t_i) \leq h$. The correctness is obvious: either $d_B(s_i, t_i) \leq h$ which already implies that $d_G(s_i, t_i) \leq h$ or otherwise we maintain $d_{G[Q(s_i, t_i)]}(s_i, t_i)$ which captures all h-hop s_i-t_i paths. The more

important point is the efficiency of maintaining both distances. Our analysis mainly uses the following lemma.

Lemma 3.1 (Either hub-distance or path-union graph is small). *With high probability, for each i, either $d_B(s_i, t_i) \leq h$, or $G[\mathcal{P}(s_i, t_i, h, G)]$ has at most $\min(m/b, n^2/b^2)$ edges.*

Proof Sketch. By the random sampling of the hubs, if $G[\mathcal{P}(s_i, t_i, h, G)]$ has more than n/b nodes, then one of these nodes, say x, will have been sampled by the algorithm and thus contained in B whp (Lemma 2.11). By definition, x lies on some s_i-t_i path of length at most h. Thus, $d_B(s_i, t_i) \leq d_G(s_i, x) + d_G(x, t_i) \leq h$. Similarly, if $G[\mathcal{P}(s_i, t_i, h, G)]$ has more than m/b edges, then one of these edges, say (x, y), will have been sampled by the algorithm whp. This means that x will be contained in B, and thus $d_B(s_i, t_i) \leq d_G(s_i, x) + d_G(x, t_i) \leq h$. As the number of edges of $G[\mathcal{P}(s_i, t_i, h, G)]$ can be at most $|\mathcal{P}(s_i, t_i, h, G)|^2$, the stated bound follows. $\square$

Lemma 3.1 guarantees that when we construct $Q(s_i, t_i)$, its induced subgraph is much smaller than G whp, and so it is beneficial to maintain the distance in $G[Q(s_i, t_i)]$ instead of G. For each $1 \leq i \leq k$, we can maintain $d_{G[Q(s_i,t_i)]}(s_i, t_i)$ by running an ES-tree rooted at s_i up to distance h in $G[Q(s_i, t_i)]$, which takes time $\tilde{O}(|E[Q(s_i, t_i)]| \cdot h)$ (where $E[Q(s_i, t_i)]$ denotes the set of edges of $G[Q(s_i, t_i)]$). As $|E[Q(s_i, t_i)]| \leq \min(m/b, n^2/b^2)$, this takes time $O(kh\min(m/b, n^2/b^2))$. By Lemma 2.7 we can construct each $Q(s_i, t_i)$ in $O(m)$ time for each $1 \leq i \leq k$, resulting in a total cost of $O(km)$ for computing the path unions. Finally, the time needed for maintaining the hub distances can be analyzed as follows.

Lemma 3.2 (Maintaining all $d_B(s_i, t_i)$). *We can maintain whether $d_B(s_i, t_i) \leq h$, for all $1 \leq i \leq k$, in total time $\tilde{O}(bmh + kbh)$.*

Proof Sketch. For each hub $v \in B$, we maintain the values of $d_G(s_i, v)$ and $d_G(v, t_i)$ up to h using ES-trees up to distance h rooted at each hub. This takes $\tilde{O}(bmh)$ total update time. Every time $d_G(s_i, v)$ or $d_G(v, t_i)$ changes, for some hub $v \in B$ and some $1 \leq i \leq k$, we update the value of $d_G(s_i, t_i) = \min_{v \in B}(d_G(s_i, v) + d_G(v, t_i))$, incurring $\tilde{O}(b)$ time. Since the value of each of $d_G(s_i, v)$ and $d_G(v, t_i)$ can change at most h times, we need $\tilde{O}(kbh)$ time in total. $\square$

It follows that we can maintain, for all $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$ with a total update time of

$$\tilde{O}\Big(\underbrace{bmh + kbh}_{\substack{\text{maintain } d_B(s_i,t_i) \\ \text{(Lemma 3.2)}}} + \underbrace{km}_{\text{construct } Q(s_i,t_i)} + \underbrace{kh \cdot \min(m/b, n^2/b^2)}_{\text{maintain } d_{G[Q(s_i,t_i)]}(s_i,t_i)} \Big). \tag{1}$$

Note that, previously, the fastest way of maintaining, for all $1 \leq i \leq k$, whether $d_G(s_i, t_i) \leq h$ was to maintain an ES-tree separately for each pair, which takes $O(mhk)$ time.

3.2 Decremental *s-t* Reachability in Dense Graphs

In the following we are given a source node s and a sink node t and want to maintain whether s can reach t. In our algorithm we use a set of *centers* $C \subseteq V$ of size $\tilde{O}(c)$ (for some parameter c) obtained by sampling each node independently with probability $ac \ln n/n$ (for

some large enough constant a). Using these centers, we define the *center graph*, denoted by $\mathcal{C}$, as follows. The nodes of $\mathcal{C}$ are the centers in C and for every pair of centers u and v in C, there is a directed edge (u, v) in $\mathcal{C}$ if and only if $d_G(u, v) \leq n/c$.

Lemma 3.3 ($\mathcal{C}$ preserves s-t reachability). *Whp, s can reach t in G if and only if s can reach t in $\mathcal{C}$.*

Proof Sketch. Since we can convert any path in $\mathcal{C}$ to a path in G, the "if" part is clear. To prove the "only if" part, let π be an s-t path in G. By the random sampling of centers, there is a set of centers $c_1, c_2, \ldots, c_k$ on π such that $c_1 = s$, $c_k = t$, and $d_G(c_i, c_{i+1}) \leq n/c$ for all i whp (Lemma 2.11). The last property implies that the edge (c_i, c_{i+1}) is contained in $\mathcal{C}$ for all i. Thus s can reach t in $\mathcal{C}$. $\qquad\square$

Lemma 3.3 implies that to maintain s-t reachability in $\mathcal{C}$ it is sufficient to maintain the edges of $\mathcal{C}$ and to maintain s-t reachability in $\mathcal{C}$. To maintain the edges of $\mathcal{C}$, we simply have to maintain the distances between all pairs of centers up to n/c. This can be done using the bounded-hop multi-pair reachability algorithm from before with $k = c^2$ and $h = n/c$, where we make each pair of centers a source-sink pair. By plugging in these values in Equation (1), we obtain a total update time of $\tilde{O}(bmn/c + bcn + c^2m + cn^3/b^2)$ for this bounded-hop multi-pair reachability. To maintain s-t reachability in $\mathcal{C}$, note that $\mathcal{C}$ is a dynamic graph undergoing only deletions. Thus, we can simply maintain an ES-tree rooted at s in $\mathcal{C}$. As $\mathcal{C}$ has $\tilde{O}(c)$ nodes and $\tilde{O}(c^2)$ edges this takes time $\tilde{O}(c^3)$. It follows that the total update time of this s-t reachability algorithm is

$$\tilde{O}\Big(\underbrace{bmn/c + bnc}_{\substack{\text{maintain } d_B(\cdot,\cdot) \\ \text{(Lemma 3.2)}}} + \underbrace{c^2m}_{\text{construct } Q(\cdot,\cdot)} + \underbrace{cn^3/b^2}_{\text{maintain } d_{G[Q(s_i,t_i)]}(\cdot,\cdot)} + \underbrace{c^3}_{\text{maintain } d_{\mathcal{C}}(s,t)} \Big). \qquad (2)$$

By setting[5] $b = n^{8/7}/m^{3/7}$ and $c = (bn)^{1/3} = n^{5/7}/m^{1/7}$, we get a total update time of[6] $\tilde{O}\left(m^{5/7}n^{10/7} + n^{20/7}/m^{4/7} \right)$. This is $o(mn)$ if $m = \omega(n^{3/2})$.

3.3 Decremental s-t reachability in Sparse Graphs

Maintaining s-t reachability in sparse graphs, especially when $m = \Theta(n)$, needs a slightly different approach. Carefully examining the running time of the previous algorithm in Equation (2) reveals that we *cannot* maintain $d_{G[Q(u,v)]}(u, v)$ for *all* pairs of centers u and v at all times: this would cost $\tilde{O}(\min(cmn/b, cn^3/b^2))$—the third term of Equation (1)—but we always need $c = \omega(b)$ to keep the first term of Equation (1) to $bmn/c = o(mn)$ and, since $b \leq n$, $cmn/b = \omega(mn)$ and $cn^3/b^2 = \omega(n^2) = \omega(mn)$. The new strategy is to maintain $d_{G[Q(u,v)]}(u, v)$ only for *some* pairs of centers at each time step.

---

[5]Note that we have to make sure that $1 \leq b \leq n$ and $1 \leq c \leq n$. It is easy to check that this is the case using the fact that $m \leq n^2$.

[6] *Detailed calculation:* First note that $c \leq n^{5/7} \leq m$. 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 $bnc = c^4 = n^{20/7}/m^{4/7}$.

14

Algorithm. As before, we sample $\tilde{O}(b)$ hubs $\tilde{O}(c)$ centers, and maintain $d_B(\cdot, \cdot)$ between all centers which takes $\tilde{O}(mbn/c + bnc)$ time (Lemma 3.2). The algorithm runs in phases. At the beginning of each phase i, the algorithm does the following. Compute a BFS-tree T on the outgoing edges of every node rooted at the source s in G. If T does not contain t, then we know that s cannot reach t anymore and there is nothing to do. Otherwise, let $L = (c_1, c_2, \ldots, c_k)$ be the list of centers on the shortest path from s to t in T ordered increasingly by their distances to s. For simplicity, we let $c_0 = s$ and $c_{k+1} = t$. Note that we can assume that

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

The first inequality holds because the centers are obtained from random sampling (Lemma 2.11) 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 would induce an s-t path in the "center graph". Our intention in this phase is to maintain whether s can still reach t using this path; in other words, whether $d_G c_i, c_{i+1}) \leq n/c$ for all i. (We start a new phase if this is not the case.) We do this using the framework of Section 3.1: For each pair (c_i, c_{i+1}), we know that $d_G(c_i, c_{i+1}) \leq n/c$ when $d_B(c_i, c_{i+1}) \leq n/c$. Once $d_B(c_i, c_{i+1}) > n/c$, we construct $Q(c_i, c_{i+1}) = \mathcal{P}(c_i, c_{i+1}, n/c, G)$. After each deletion of an edge (u, v) in this phase, we find every index i such that u and v are contained in $Q(c_i, c_{i+1})$ and, using the static BFS algorithm, check whether $d_{G[Q(c_i, c_{i+1})]}(c_i, c_{i+1}) \leq n/c$. We start a new phase when $d_G(c_i, c_{i+1})$ for some pair of centers (c_i, c_{i+1}) changes from $d_G(c_i, c_{i+1}) \leq n/c$ to $d_G(c_i, c_{i+1}) > n/c$.

Running Time Analysis. First, let us bound the number of phases. As for each pair (c_i, c_{i+1}) the distance $d_G(c_i, c_{i+1})$ can only become larger than n/c at most once, there are at most $\tilde{O}(c^2)$ phases. At the beginning of each phase, we have to construct a BFS-tree in G, taking $O(m)$ time and contributing $\tilde{O}(c^2 m)$ to the total time over all phases. During the phase, we have to construct $Q(c_i, c_{i+1})$ for at most c pairs of center, taking $\tilde{O}(cm)$ time (by Lemma 2.7) and contributing $\tilde{O}(c^3 m)$ total time. Moreover, after deleting an edge (u, v), we have to update $d_{G[Q(c_i, c_{i+1})]}(c_i, c_{i+1})$ for every $Q(c_i, c_{i+1})$ containing both u and v, by running a BFS algorithm. This takes $O(|E[Q(c_i, c_{i+1}]|)$ time for each $Q(c_i, c_{i+1})$, which is $\tilde{O}(m/b)$ whp, by Lemma 3.1. The following lemma implies that every node will be contained in only a constant number of such sets $Q(c_i, c_{i+1})$; so, we need $\tilde{O}(m^2/b)$ time to update $d_{G[Q(c_i, c_{i+1})]}(c_i, c_{i+1})$ over all m deletions.

Lemma 3.4. *For any i and $j \geq i + 3$, $Q(c_i, c_{i+1})$ and $Q(c_j, c_{j+1})$ are disjoint.*

Proof. First, we claim that $d_G(c_i, c_{j+1}) > 2n/c$. To see this, let G' be the version of the graph at the beginning of the current phase. We know that $d_{G'}(c_i, c_{i+4}) = d_{G'}(c_i, c_{i+2}) + d_{G'}(c_{i+2}, c_{i+4}) > 2n/c$, where the equality holds because at the beginning of the current phase (i.e., in G'), every c_i lies on the shortest s-t path, and the inequality holds because of Equation (3). Since $j + 1 \geq i + 4$ and both c_{i+4} and c_{j+1} lie on the shortest s-t path in G', $d_{G'}(c_i, c_{j+1}) \geq d_{G'}(c_i, c_{i+4}) > 2n/c$. The claim follows since the distance between two nodes never decreases after edge deletions.

Now, suppose for the sake of contradiction that there is some node v that is contained in both $Q(c_i, c_{i+1}) = \mathcal{P}(c_i, c_{i+1}, n/c, G)$ and $Q(c_j, c_{j+1}) = \mathcal{P}(c_j, c_{j+1}, n/c, G)$. This means that v lies in some c_i-c_{i+1} path and some c_j-c_{j+1} path, each of length at most n/c. This means that $d_G(c_i, v) \leq n/c$ and $d_G(v, c_{j+1}) \leq n/c$ and therefore we get $d_G(c_i, c_{j+1}) \leq d_G(c_i, v) + d_G(v, c_{j+1}) \leq 2n/c$. This contradicts the lower bound above. $\square$

Thus, the total update time is

$$\tilde{O}\Big(\underbrace{mbn/c + bnc}_{\substack{\text{maintain } d_B(\cdot,\cdot) \\ \text{(Lemma 3.2)}}} + \underbrace{mc^2}_{\substack{\text{construct BFS} \\ \text{tree in every phase}}} + \underbrace{mc^3}_{\substack{\text{construct } Q(c_i,c_{i+1}) \\ \text{in every phase}}} + \underbrace{m^2/b}_{\text{update } d_{G[Q(c_i,c_{i+1})]}(c_i,c_{i+1})} \Big).$$

By setting $c = (mn)^{1/7}$ and $b = c^4/n = m^{4/7}/n^{3/7}$, we get a running time of $\tilde{O}\left(m^{10/7}n^{3/7}\right)$.[7] This is $o(mn)$ if $m = o(n^{4/3})$.

4 General Reductions

4.1 From SSSP to stSP

In the following we show a reduction from decremental approximate SSSP to decremental approximate stSP. The naive way of doing this would be to use n instances of the approximate stSP path algorithm, one for every node. We can use much fewer instances by randomly sampling the nodes for which we maintain approximatestSP and by using Bernstein's shortcut edges technique [Ber13].

Theorem 4.1. *Assume we already have the following decremental algorithm that, given a weighted directed graph G undergoing edge deletions and edge weight increases, a source node s, and a set of sinks T of size k, maintains, for every sink $t \in T$, a distance estimate $\delta(s,t)$ such that $d_G(s,t) \leq \delta(s,t) \leq \alpha d_G(s,t))$ for some $\alpha \geq 1$ with constant query time and a total update time of $T(k,m,n)$. Then, for any $k \leq n$, there exists a decremental $(1+\epsilon)\alpha$-approximate SSSP algorithm with constant query time and total update time $O(T(O(k\log{(n\Delta)}),m,n)+mn\Delta/k$, where $\Delta = \lceil \log_{1+\epsilon}(nW) \rceil$, that is correct with high probability against an oblivious adversary.*

Proof. At the initialization we randomly sample each node of G with probability $ak \ln{(n\Delta)}/n$ (for a large enough constant a). We call the sampled nodes sinks. We use the decremental algorithm for a set of sinks from the assumption to maintain a distance estimate $\delta(s,t)$ for every sink t such that $d_G(s,t) \leq \delta(s,t) \leq \alpha d_G(s,t)$. Additionally, we maintain a graph G' which consists of the same nodes as G and the following edges: (1) For every edge (u,v) of G we add an edge (u,v) of weight $w_{G'}(u,v) = (1+\epsilon)^{\lceil \log_{1+\epsilon} w_G(u,v) \rceil}$. (2) For every sink t we add an edge (s,t) of weight $w_{G'}(s,t) = (1+\epsilon)^{\lceil \log_{1+\epsilon} \delta(s,t) \rceil}$ (these edges are called *shortcut edges*). We maintain G' by updating the weights of these edges every time there is an edge weight increase or deletions in G or the distance estimate $\delta(s,t)$ of some sink t changes its value. On G' we use Bernstein's decremental $(1+\epsilon)$-approximate SSSP algorithm [Ber13] with source s and hop count $h = n/k$. This algorithm maintains a distance estimate $\delta'(s,v)$ for every node v such that $d_{G'}(s,v) \leq \delta'(s,v) \leq (1+\epsilon)d_{G'}^h(s,v)$.

First, observe that $d_G(s,v) \leq d_{G'}(s,v) \leq (1+\epsilon)d_G(s,v)$ as the new shortcut edges never under-estimate the true distances in G. We now claim that $d_{G'}^h(s,v) \leq (1+\epsilon)\alpha d_G(s,v)$. If $d_G(s,v) = \infty$, then the claim is trivially true. Otherwise let π be the shortest path from s to v in G. All edges of this path are also contained in G'. Thus, if π has at most h edges,

---

[7] *Detailed calculation:* First note that the third term (mc^2) is dominated by the fourth term (mc^3). Using $b = c^4/n$, the first term is the same as the fourth term (mc^3). Now, the fourth term is $mc^3 = m(mn)^{3/7} = m^{10/7}n^{3/7}$. For the second term, using $bn = c^4$, we have $bnc = c^5 = (mn)^{5/7}$ which is at most $m^{10/7}n^{3/7}$ (using $n \leq m$). For the last term, $m^2/b = m^2/(m^{4/7}/n^{3/7}) = m^{(14-4)/7}n^{3/7}$.

then $d^h_{G'}(s,v) = d_G(s,v)$. If π has more than h edges, then we know that the set of nodes consisting of the last h nodes of π contains a sink t whp by Lemma 2.11 as the sinks are obtained by sampling from the nodes with probability $ak \ln(n\Delta)/n = a \ln(n\Delta)/h$. Thus, the graph G' contains a shortcut edge (s,t) of weight $w_{G'}(s,t) \leq (1+\epsilon)\delta(s,t) \leq (1+\epsilon)\alpha d_G(s,t)$. Now let π' be the path from s to v that starts with this edge (s,t) and then follows the path π from t to v. Clearly, the path π' has at most h edges and is contained in G'. As the weight of π' is at most $w_{G'}(s,t) + (1+\epsilon)d_G(t,v)$ we get

$$d^h_{G'}(s,v) \leq w_{G'}(s,v) + (1+\epsilon)d_G(t,v) \leq (1+\epsilon)\alpha d_G(s,t) + d_G(t,v) \leq (1+\epsilon)\alpha d_G(s,v)\,.$$

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

$$d_G(s,v) \leq d_{G'}(s,v) \leq \delta'(s,v) \leq (1+\epsilon)d^h_{G'}(s,v) \leq (1+\epsilon)^2\alpha d_G(s,v) \leq (1+3\epsilon)\alpha d^h_{G'}(s,v)\,.$$

By running the whole algorithm with $\epsilon' = \epsilon/3$ we obtain a $(1+\epsilon)\alpha$-approximation instead of a $(1+3\epsilon)\alpha$-approximation.

Finally, we argue about the running time. Our running time has two parts. (1) Bernstein's decremental $(1+\epsilon)$-approximate SSSP algorithm [Ber13] has constant query time and a total update time of $O(mh\Delta)$ (here we also use the fact that $w_{G'}(u,t)$ increases at most Δ times for every sink t). Thus, our decremental algorithm also has constant query time and by our choice of $h = n/k$, our total update time contains the term $mn\Delta/k$). (2) Furthermore, we have $O(k \log(n\Delta))$ sinks whp by Lemma 2.11 (and can make the algorithm fail with small probability if not). Thus, the total update time of the decremental algorithm for a set of sinks from the assumption is $T(O(k \log(n\Delta)), m, n)$. □

4.2 From SCC to SSR

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 [RZ08], but in order to work in our setting we have to generalize their running time analysis at the cost of losing a factor of $\log n$. They show that an $O(mn)$ algorithm for single-source reachability implies an $O(mn)$ algorithm for strongly connected components. We show that in fact $o(mn)$ time for single-source reachability implies $o(mn)$ time for strongly connected components. In the following we will often just write "component" instead of "strongly connected component".

In contrast to the rest of this paper, we will here impose the following technical condition on the decremental single-source reachability algorithm: when we update the algorithm after the deletion of an edge, the update procedure will return all nodes that were reachable before the deletion, but are not reachable anymore after this deletion. Note that all the reachability algorithms we present in this paper fulfill this condition.

Algorithm. The algorithm works as follows. For every component we, uniformly at random, choose among its nodes one *representative*. In an array, we store for every node a pointer to the representative of its component. Queries that ask for the component of a node v are answered in constant time by returning (the ID of) the representative of v's component. Using the decremental SSR algorithm, we maintain, for every representative w of a component C, the sets $I(w)$ and $O(w)$ containing all nodes that reach w and that can

17

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 if, after the deletion, $u \notin I(w)$ or $v \notin O(w)$ (which can be checked with the SSR algorithm of w).

We now explain the behavior of the algorithm when a component C decomposes into the new components $C_1, \ldots, C_k$. 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 decremental SSR algorithms. There is one notable exception: If the representative w of C is still contained in one of the components C_j, then for this component we do *not* choose a new representative. Instead, C_j reuses w and its SSR algorithms without any re-initialization. The key to the efficiency of the algorithm is that a large component C_i has a high probability of inheriting the representative from C.

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 $C_1, \ldots, C_{j-1}, C_{j+1}, \ldots C_k$. 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 algorithms of w, which allows us to compute $A \cup B$. The graph G' can be constructed by iterating over all outgoing edges of $A \cup B$ and the components of G' can be found using a static SCC algorithm.

Analysis. The correctness of the algorithm explained above is immediate. For the running time we will argue that, up to a $\log n$-factor, it is the same as the running time of the SSR algorithm. When the running time of SSR is of the form $\tilde{O}(m^\alpha n^\beta)$, our argument works when $\alpha \geq 1$ or $\beta \geq 1$. To understand the basic idea (for $\beta \geq 1$), consider the case that the graph decomposes into only two components C_1 and C_2 (with $n_1, n_2 \leq n$ and $m_1, m_2 \leq m$ being the corresponding number of nodes and edges). We know that one of the two components still contains the representative w. For this component we do not have to spawn a new decremental SSR algorithm. This is an advantage for large components as they have a high probability of containing the representative. The probability of w being contained in C_1 is n_1/n and it is n_2/n for being contained in C_2. Thus, the expected cost of the decomposition is $O(m_2^\alpha n_2^\beta n_1/n + m_1^\alpha n_1^\beta n_2/n)$. We charge this cost to the smaller component, say C_1. As C_1 has n_1 nodes, the average cost we charge to every node in C_1 is $O(m_2^\alpha n_2^\beta/n + m_1^\alpha n_1^{\beta-1} n_2/n)$. This amounts to an average cost of $O(m^\alpha n^{\beta-1})$ per node. As we will charge each node only when the size of its component has halved, the total update time is $O(m^\alpha n^\beta \log n)$.

Theorem 4.2 (From SSR to SCC)**.** *If there is a decremental SSR algorithm with constant query time and a total update time of $O(n t_1(m, n) + m t_2(m, n))$ such that $t_1(m, n) \geq 1$, $t_2(m, n) \geq 1$, and $t_1(m, n)$ and $t_2(m, n)$ are non-decreasing[8] in m and n, then there exists a*

---

[8]The technical assumption that $t_1(m, n)$ and $t_2(m, n)$ are non-decreasing in m and n is natural as usually

decremental SCC *algorithm with constant query time and an expected total update time of* $O((nt_1(m,n) + mt_2(m,n))\log n)$.

Proof. Let us first analyze the costs related to the decomposition of a component. Assume that the component C_0 decomposes into $C_1, \ldots, C_k$. Let n_i and m_i denote the number of nodes and edges in component i, respectively. Let m_i' denote the sum of the out-degrees of the nodes in C_i in the initial graph (i.e., before the first deletion). Note that m_i' is an upper bound on m_i. Furthermore we have $\sum_{i=1}^k n_i = n_0$, $\sum_{i=1}^k m_i \le m_0$, 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 $j \ne i$ we have to pay a cost of $O(n_j t_1(m_j, n_j) + m_j t_2(m_j, n_j))$ for initializing and updating the decremental SSR algorithm of the new representative of C_j. Second, we have to pay for computing the new components. This consists of three steps: (a) computing $A \cup B$, (b) computing G', and (c) computing the components of G'. Remember that $A \cup B$ is the union of A, the set of nodes that cannot reach w anymore after deleting (u,v), and B, the set of nodes that w cannot reach anymore after deleting (u,v). After deleting (u,v) the incoming SSR algorithm of w outputs A and the outgoing SSR algorithm of w outputs B. Thus, the cost of computing $A \cup B$ can be charged to the reachability algorithms of w (which have to output A and B anyway). The graph G' is the subgraph of G induced by the nodes in $A \cup B$. We construct G' by checking, for every node in $A \cup B$, which of its outgoing edges stay in $A \cup B$. This takes time $O(\sum_{j \ne i} m_j')$. Using Tarjan's linear time algorithm [Tar72], we can compute the strongly connected components of G' in the same running time. As $m_j' \ge m_j$ the total cost of C_j is $O(n_j t_1(m_j, n_j) + m_j' t_2(m_j, n_j)) = O(n_j t_1(m,n) + m_j' t_2(m,n))$

By the random choice of the representatives, the probability that w is contained in C_i is n_i/n_0. Thus, the expected cost of the decomposition of C_0 is proportional to

$$\sum_{i=1}^k \frac{n_i}{n_0} \sum_{j \ne i} (n_j t_1(m,n) + m_j' t_2(m,n)) = \sum_{i=1}^k \frac{n_i}{n_0} \sum_{j \ne i} n_j t_1(m,n) + \sum_{i=1}^k \frac{n_i}{n_0} \sum_{j \ne i} m_j' t_2(m,n). \quad (4)$$

We analyze each of these terms individually.

Consider first the cost of $O(\sum_{i=1}^k n_i/n_0 \sum_{j \ne i} n_j t_1(m,n))$. For every pair i, j such that $i \ne j$ we have to pay a cost of $O(n_i n_j (t_1(m,n) + n)/n_0)$. If $n_i \le n_j$ we charge this cost to the component C_i, otherwise we charge it to C_j (i.e., we always charge the cost to the smaller component). Note that the component to which we charge the cost has at most $n_0/2$ nodes (otherwise it would not be the smaller one). For a fixed component i, the total charge is proportional to

$$\sum_{j \ne i} \frac{n_i n_j (t_1(m,n) + n)}{n_0} = n_i (t_1(m,n) + n) \cdot \frac{\sum_{j \ne i} n_j}{n_0} \le n_i (t_1(m,n) + n).$$

We share this cost equally among the nodes in C_i and thus charge $O(t(m,n) + n)$ to every node in C_i. Every time we charge a node, the size of its component halves. Thus, every node is charged at most $\log n$ times and the total update time for the first term in Equation (4) is $O(nt(m,n) \log n)$.

Consider now the cost of $O(\sum_{i=1}^k n_i/n_0 \sum_{j \ne i} m_j' t_2(m,n))$. Note that

$$\sum_{i=1}^k \frac{n_i}{n_0} \sum_{j \ne i} m_j' = \sum_{i=1}^k m_i' \sum_{j \ne i} \frac{n_j}{n_0} = \sum_{i=1}^k m_i' \frac{n_0 - n_i}{n_0}.$$

---

the running time of an algorithm does not improve with increasing problem size.

We now charge $m'_i(n_0 - n_i)/n_0$ to every component C_i ($1 \le i \le k$). In particular we charge $(n_0 - n_i)/n_0$ to every edge (u, v) of the initial graph such that $u \in C_i$. We now argue that in this way every edge is charged only $O(\log n)$ times, which will imply that the total update time for the second term in Equation (4) is $O(mt_2(m, n) \log n)$. Consider an edge (u, v) and the component containing u. We only charge the edge (u, v) when the component containing u decomposes. Let a_0 denote the initial number of nodes of this component and let a_p its number of nodes after the p-th decomposition. As argued above, we charge $(a_{p-1} - a_p)/a_{p-1}$ to (u, v) for the p-th decomposition. Thus, for q decompositions we charge $\sum_{1 \le p \le q}(a_{p-1} - a_p)/a_{p-1}$. Now observe that

$$\sum_{1 \le p \le q} \frac{a_{p-1} - a_p}{a_{p-1}} \le \sum_{1 \le p \le q} \sum_{i=0}^{a_{p-1}-a_p-1} \frac{1}{a_{p-1}} \le \sum_{1 \le p \le q} \sum_{i=0}^{a_{p-1}-a_p-1} \frac{1}{a_{p-1}-i}$$

$$= \sum_{1 \le p \le q} \sum_{i=a_p+1}^{a_{p-1}} \frac{1}{i} = \sum_{i=a_q+1}^{a_0} \frac{1}{i}.$$

Since $a_0 \le n$, this harmonic series is bounded by $O(\log n)$.

Finally, we bound the initialization cost. Let $C_1, \ldots, C_k$ denote the initial components and let n_i and m_i denote the number of nodes and edges of component C_i, respectively. The initial components can be computed in time $O(m)$ with Tarjan's algorithm [Tar72]. Furthermore, each component starts two decremental SSR algorithms and we have to pay for the total update time of these algorithms. This time is proportional to

$$\sum_{i=1}^{k}(n_i t_1(m_i, n_i) + t_2(m_i, n_i)) \le t_1(m, n) \sum_{i=1}^{k} n_i + t_2(m, n) \sum_{i=1}^{k} m_i \le n t_1(m, n) + m t_2(m, n).$$

$\square$

5 Single-Source Single-Sink Reachability

In this section we give an algorithm for maintaining a path from a source node s to a sink node t in a directed graph undergoing edge deletions, i.e., we solve the decremental stR problem. Using the reduction of Section 4.1, this implies an algorithm for the decremental single-source reachability (SSR) problem.

5.1 Algorithm Description

Our s-t reachability algorithm has a parameter $k \ge 1$ and for each $1 \le i \le k$ parameters $b_i \le n$ and $c_i \le n$. We determine suitable choices of these parameters in Section 5.3. For each $1 \le i \le k - 1$, our choice will satisfy $b_i \ge b_{i+1}$ and $c_i \ge 2c_{i+1}$. We also set $b_{k+1} = 1$, $c_0 = n$, $c_{k+1} = 1$, and $h_i = n/c_i$ for all $0 \le i \le k + 1$. Note that this implies $h_{i+1} \ge 2h_i$ for all $1 \le i \le k$. Intuitively, b_i and c_i are roughly the number of i-hubs and i-centers used by our algorithm and h_i is the hop range of the i-centers. In the algorithm we will often consider ordered pairs of centers of the form (x, y).

Initialization. At the initialization (i.e., before the first deletion), our algorithm determines sets of nodes $B_1 \supseteq B_2 \supseteq \cdots \supseteq B_k$ and $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_{k+1}$ as follows. For each $1 \le i \le k$, we sample each node of the graph with probability $ab_i \ln(\Delta(G))/n$ and each edge with probability $ab_i \ln(\Delta(G))/m$ (for a large enough constant a). The set B_i then consists of the sampled nodes and the endpoints of the sampled edges. We set $C_0 = V$ and $C_{k+1} = \{s, t\}$. For each $1 \le i \le k$, we sample each node of the graph with probability $ac_i \ln(\Delta(G))/n$ (for a large enough constant a). The set C_i then consists of the sampled nodes together with the nodes in C_{i+1}. For every $1 \le i \le k$ we call the nodes in B_i i-hubs and for every $0 \le i \le k+1$ we call the nodes in C_i i-centers. Note that, as $\Delta(G) \le m$ in unweighted graphs, the number of i-hubs is $\tilde{O}(b_i)$ whp and the number of i-centers is $\tilde{O}(c_i)$ by Lemma 2.11. We can make the algorithm fail with small probability if these sets are larger.

Data Structures. Our algorithm uses the following data structures:

- For every i-hub z (with $1 \le i \le k$) an ES-tree up to depth $2h_i$ in G (*outgoing* tree of z) and an ES-tree up to depth $2h_i$ in the reverse graph of G (*incoming* tree of z)

- For every pair of i-centers (x, y) (with $1 \le i \le k$) the set of i-hubs from B_i linking x to y.

- For every pair of i-centers (x, y) (with $1 \le i \le k$) a set of nodes $Q(x, y, i)$, which is initially empty.

- For every pair of i-centers (x, y) (with $0 \le i \le k$) a list of pairs of $(i+1)$-centers called $(i+1)$-parents of (x, y).

- For every pair of i-centers (x, y) (with $1 \le i \le k+1$) a list of pairs of $(i-1)$-centers called $(i-1)$-children of (x, y).

Maintaining Hub Links. For every $1 \le i \le k$, we say that an i-hub z *links* an i-center x to an i-center y if $d_G(x, z) \le 2h_i$ and $d_G(z, y) \le 2h_i$. The hub links of every pair of i-centers (x, y), for all $1 \le i \le k$, can be maintained as follows. After every edge deletion in the graph, we report the deletion to the ES-trees maintained by the hubs. Initially, and after each deletion, the level of a node v in the incoming (outgoing) tree of an i-hub z is at most $2h_i$ if and only if $d_G(v, z) \le 2h_i$ ($d_G(z, v) \ge 2h_i$). Thus, we can check whether an i-hub z links an i-center x to an i-center y by summing up the levels of x and y in the incoming and outgoing ES-tree of z, respectively. Thus, for every priority i and every pair of i-centers (x, y) we can initialize the set of i-hubs linking x to y by iterating over all hubs. We can maintain these sets under the edge deletions in G as follows: Every time the level of some i-center x in the incoming ES-tree of some i-hub z exceeds $2h_i$, we iterate over all i-centers y and remove z from the set of hubs linking x to y. We proceed similarly if the level of some i-center y in the outgoing ES-tree of some i hub z exceeds $2h_i$. In this way, we can also generate, after every edge deletion, a list of pairs of i-centers (x, y) such that x is not linked to y by an i-hub anymore (but was linked to an i-hub before the deletion).

Main Algorithm. Initially, the algorithm computes the shortest path π from s to t in G. It then determines a sequence of k-centers $v_1, \ldots, v_l$ on π. For each $1 \le j \le l-1$, the

algorithm now tries to maintain a path from v_j to v_{j+1} under the edge deletions in G. If it fails to do so, it recomputes the shortest path from s to t in G. We call this a *refresh* operation.

For each $1 \leq j \leq l - 1$, the path from v_j to v_{j+1} is maintained as follows. The first case is that there is a k-hub linking v_j to v_{j+1}. As long as such a hub exists, we know that there is a path from v_j to v_{j+1}. If there is no k-hub linking v_j to v_{j+1} anymore, the algorithm computes the path union $\mathcal{P}(v_j, v_{j+1}, 2h_k, G)$. It then recursively maintains the path from v_j to v_{j+1} in $G[\mathcal{P}(v_j, v_{j+1}, 2h_k, G)]$ using $(k-1)$-centers and $(k-1)$-hubs. To keep track of this recursive hierarchy, we make, for every $1 \leq j \leq l - 1$, each pair (v_j, v_{j+1}) an $(i-1)$-child of (x, y) and (x, y) an i-parent of (v_j, v_{j+1}). The end of the recursion is reached in layer 0, where the set of 0-centers consists of all nodes. The only paths between 0-centers the algorithm considers are the edges between them.

Algorithm 1 shows the pseudocode for these procedures. A crucial ingredient of our algorithm is the computation of path unions for pairs of centers. For every $1 \leq i \leq k$ and all i-centers (x, y) the path union from the last recomputation is stored in $Q(x, y, i)$. To simplify the formulation of the algorithm we set $Q(s, t, k + 1) = V$ (which is consistent with our choice of $h_{k+1} = n$). Thus, the procedure REFRESH(s, t, $k + 1$) simply checks whether there exists a path from s to t in G.

5.2 Correctness

It is obvious that the algorithm correctly maintains, for all pairs of i-centers (x, y), the set of i-hubs linking x to y. Furthermore, the algorithm only stops if it has correctly detected that there is no path from s to t in the current graph G. Thus, it remains to show that, as long as the algorithm does not stop, there there is a path from s to t. We say that a pair of i-centers (x, y) is *active* if it has at least one $(i + 1)$-parent. We further define the pair (s, t), the only pair of $(k + 1)$-centers, to be active.

Lemma 5.1. *After finishing the delete-operation, for every $0 \leq i \leq k + 1$ and every active pair of i-centers (x, y), there is a path from x to y in the current graph G.*

Proof. The proof is by induction on i. If y is linked to x by some i-hub z, we know that in G there is a path from x to z as well as a path from z to y. Their concatenation is a path from x to y in G.

Consider now the case that there is no i-hub linking x to y. If $i = 0$, we know that there is an edge from x to y in G as otherwise the pair (x, y) would not be active. If $i \geq 1$, let $(v_1, v_2), (v_2, v_3), \ldots, (v_{l-1}, v_l)$ with $v_1 = x$ and $v_l = y$ denote the $(i + 1)$-children of (x, y) found at the time of the last refresh of (x, y). Note that all these children are active since (x, y) is their i-parent. Thus, by the induction hypothesis, we know that there is a path from v_j to v_{j+1} in G for all $1 \leq j \leq l - 1$. The concatenation of these paths gives a path from x to y in G, as desired. $\square$

5.3 Running Time Analysis

Important Properties. In the running time analysis we will need some properties of the path unions computed by our algorithm.

Algorithm 1: Algorithm for decremental *s-t* reachability

---

1 Procedure REFRESH(x, y, i)

2   Compute shortest path π from x to y in $G[Q(x,y,i)]$

3   **if** *length of $\pi > h_i$* **then**

4   **if** $i = k+1$ **then**

5   Stop and output "*s* cannot reach *t*"

6   **foreach** $(i+1)$-*parent* (x',y') *of* (x,y) **do**

7   REFRESH(x', y', $i+1$)

8   **else**

9   REMOVECHILDREN(x, y, i)

10   Determine $(i-1)$-centers $v_1, \ldots, v_l$ on π in order of appearance on π such that $v_1 = x$, $v_l = y$ and, for each $1 \le j \le l-2$, v_{j+1} is the first $(i-1)$-center following v_j on π at distance at least $h_{i-1}/2$ from v_j.

          `// If `$i = 1$` then the 0-centers are all nodes on `π` because `$h_0 = 1$`.`

11   **foreach** $1 \le j \le l-1$ **do**

12   Make (v_j, v_{j+1}) an $(i-1)$-child of (x,y) and (x,y) an i-parent of (v_j, v_{j+1})

13   **if** v_{j+1} *not linked to v_j by an i-hub* **then**

14   COMPUTEPATHUNION(v_j, v_{j+1}, $i-1$)

15   REFRESH(v_j, v_{j+1}, $i-1$)

16 Procedure COMPUTEPATHUNION(x, y, i)

17   **if** $Q(x,y,i) = \emptyset$ **then**

18   Let (x',y') be any $(i+1)$-parent of (x,y)

19 $Q(x,y,i) \leftarrow \mathcal{P}(x,y,2h_i, G[Q(x',y',i+1)])$

20   **else**

21 $Q(x,y,i) \leftarrow \mathcal{P}(x,y,2h_i, G[Q(x,y,i)])$

22 Procedure REMOVECHILDREN(x, y, i)

23   **foreach** $(i-1)$-*child* (x',y') *of* (x,y) **do**

24   Remove (x',y') from $(i-1)$-children of (x,y) and remove (x,y) from i-parents of (x',y')

25   **if** (x',y') *has no i-parents anymore and $i \ge 1$* **then**

26   REMOVECHILDREN(x', y', $i-1$)

27 Procedure INITIALIZE()

28   **foreach** $1 \le i \le k$ *and all i-centers* (x,y) **do** $Q(x,y,i) \leftarrow \emptyset$

29 $Q(s,t,k+1) \leftarrow V$

30   REFRESH(s, t, $k+1$)

31 Procedure DELETE(u, v)

32   **foreach** 1-*parent* (x,y) *of* (u,v) **do**

33   REFRESH(x, y, 1)

34   For every $1 \le i \le k$ and every pair of i-centers (x,y) that is not linked by an i-hub anymore: REFRESH(x, y, i)

Lemma 5.2. *Let (x', y') be pair of nodes such that $d_G(x', y') \le h_{i+1}$ and let (x, y) be a pair of nodes such that x and y lie on a shortest path from x to y in G (and x appears before v on this shortest path). Then $\mathcal{P}(x, y, 2h_i, G) \subseteq \mathcal{P}(x', y', 2h_{i+1}, G)$.*

Proof. Let $v \in \mathcal{P}(x, y, 2h_i, G)$. We first apply the triangle inequality:

$$d_G(x', v) + d_G(v, y') \le d_G(x', x) + d_G(x, v) + d_G(v, y) + d_G(y, y').$$

Since x and y (in this order) lie on the shortest path from x' to y', we have $d_G(x', x) + d_G(y, y') \le d_G(x', y')$ and by our assumption we have $d_G(x', y') \le h_{i+1}$. Since $v \in \mathcal{P}(x, y, 2h_i, G)$, we have $d_G(x, v) + d_G(v, y) \le 2h_i$ As $2h_i \le h_{i+1}$ by our assumptions on the parameters we get

$$d_G(x', v) + d_G(v, y') \le h_{i+1} + 2h_i \le h_{i+1} + h_{i+1} = 2h_{i+1}$$

which implies that $v \in \mathcal{P}(x', y', 2h_{i+1}, G)$. $\qquad\square$

Lemma 5.3. *For every $1 \le i \le k$ and every pair of i-centers (x, y), if $Q(x, y, i) \ne \emptyset$, then $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x, y, i)$.*

Proof. The proof is by induction on i. We first argue that it is sufficient to show that after the first initialization of $Q(x, y, i)$ in Line 19 we have $Q(x, y, i) = \mathcal{P}(x, y, 2h_i, G)$. After the initialization, and before $Q(x, y, i)$ is recomputed, $Q(x, y, i)$ might be "outdated" due to deletions in G, i.e., not be equal to $\mathcal{P}(x, y, 2h_i, G)$ anymore. However, since $\mathcal{P}(x, y, 2h_i, G)$ only "loses" nodes through the deletions in G, it is still the case that $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x, y, i) \subseteq V$. Thus, this is also true directly before $Q(x, y, i)$ is recomputed for the first time in Line 21. There $Q(x, y, i)$ is updated to $\mathcal{P}(x, y, 2h_i, G[Q(x, y, i)])$. By Lemma 2.9 we know that $\mathcal{P}(x, y, 2h_i, G[Q(x, y, i)]) = \mathcal{P}(x, y, 2h_i, G)$, i.e., after the recomputation, $Q(x, y, i)$ is equal to $\mathcal{P}(x, y, 2h_i, G)$ again. By repeating this argument, we get that $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x, y, i) \subseteq V$.

We now show that after the first initialization of $Q(x, y, i)$ in Line 19 we have $Q(x, y, i) = \mathcal{P}(x, y, 2h_i, G)$. Let (x', y') be the $(i+1)$-parent of (x, y) used in the initialization. We will show that $\mathcal{P}(x, y, 2h_i, G[Q(x', y', i+1)]) = \mathcal{P}(x, y, 2h_i, G)$. For $i = k$ the claim is trivially true because (s, t) is the only $(k+1)$-parent of (x, y) and $Q(s, t, k+1) = V$. For $i \le k-1$, we know by the induction hypothesis that $\mathcal{P}(x', y', 2h_{i+1}, G) \subseteq Q(x', y', i+1)$. We also know that x and y lie on the shortest path from x' to y' in $G[Q(x', y', i+1)]$ such that x precedes y on this shortest path and the path has length at most h_i. As $G[Q(x', y', i+1)]$ is a subgraph of G, this path is also contained in G. Thus, $d_G(x', x) + d_G(y, y') \le h_i$. By Lemma 5.2 this implies that $\mathcal{P}(x, y, 2h_i G) \subseteq \mathcal{P}(x', y', 2h_{i+1}, G)$. It follows that $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x', y', i+1)$. Thus, by Lemma 2.9, we have $\mathcal{P}(x, y, 2h_i, G[Q(x', y', i+1)]) = \mathcal{P}(x, y, 2h_i, G)$ as desired. $\quad\square$

Lemma 5.4. *Let (x, y) be an active pair of i-centers and let (x', y') be an $(i+1)$-parent of (x, y). Then $Q(x, y, i) \subseteq Q(x', y', i+1)$.*

Proof. Consider the point in time when (x', y') becomes an $(i+1)$-parent of (x, y). This can only happen in the else-branch of REFRESH$(x, y, i+1)$. Then we know that x and y lie on a path from x' to y' of length at most h_{i+1} in $G[Q(x', y', i+1)]$ such that x precedes y on this path and thus $d_G(x', x) + d_G(y, y') \le h_{i+1}$. From Lemma 5.2 it follows that $\mathcal{P}(x, y, 2h_i, G) \subseteq$

$\mathcal{P}(x', y', 2h_{i+1}, G)$. Furthermore, $\mathcal{P}(x', y', 2h_{i+1}, G) \subseteq Q(x', y', i+1)$ by Lemma 5.3. As the path-union of (x, y) is recomputed when (x, y) becomes an i-child of (x', y'), we also have $Q(x, y, i) = \mathcal{P}(x, y, 2h_i, G)$. Therefore we get that $Q(x, y, i) \subseteq Q(x', y', i+1)$. Furthermore, every time the algorithm recomputes $Q(x', y', i+1)$, (x', y') will stop being an $(i+1)$-parent of (x, y). It might immediately become in $(i+1)$-parent again and in this case our argument above applies again. $\qquad \square$

Lemma 5.5. *For every $1 \le i \le k+1$ and every pair of i-centers (x, y), we have that if $d_{G[Q(x,y,i)]}(x, y) > h_i$, then $d_G(x, y) > h_i$.*

Proof. We show that $d_G(x, y) \le h_i$ implies $d_{G[Q(x,y,i)]}(x, y) \le h_i$. If $d_G(x, y) \le h_i$, then also $d_{G[\mathcal{P}(x,y,2h_i,G)]}(x, y) \le h_i$ as $\mathcal{P}(x, y, 2h_i, G)$ contains all paths from x to y in G of length at most $2h_i$. Since $\mathcal{P}(x, y, 2h_i, G) \subseteq Q(x, y, i)$ by Lemma 5.3, we have $d_{G[Q(x,y,i)]}(x, y) \le d_{G[\mathcal{P}(x,y,2h_iG)]}(x, y)$. It follows that $d_{G[Q(x,y,i)]}(x, y) \le h_i$. $\qquad \square$

Lemma 5.6. *For every $1 \le i \le k+1$ and every pair of i-centers (x, y) the graph $G[Q(x, y, i)]$ has at most $\min(m/b_i, n^2/b_i^2)$ edges whp.*

Proof. As no edges are ever added to G, we only have to argue that the claim is true at the first initialization of $Q(x, y, i)$ in Line 19, where $Q(x, y, i)$ is equal to $\mathcal{P}(x, y, 2h_i, G)$ (see proof of Lemma 5.4). We argue that at that time $G[\mathcal{P}(x, y, 2h_i, G)]$ as at most $\min(m/b_i, n^2/b_i^2)$ edges. Note to this end it is sufficient to show that $G[\mathcal{P}(x, y, 2h_i, G)]$ has at most n/b_i nodes and m/b_i edges.

Suppose that $\mathcal{P}(x, y, 2h_i, G)$ contains more than n/b_i nodes. Then, by the random sampling of i-hubs, one of these nodes, say v, would have been sampled whp while determining the set B_i at the initialization by Lemma 2.11, making v an i-hub. If $G[\mathcal{P}(x, y, 2h_i, G)]$ contains more than m/b_i edges, then one of these edges, say (u, v), would have been sampled whp while determining the set B_i at the initialization by Lemma 2.11, making both u and v i-hubs.

In both cases, $\mathcal{P}(x, y, 2h_i, G)$ contains some i-hub v, for which $d_G(x, v) + d_G(v, y) \le 2h_i$ by Lemma 2.7. But this means that x is linked to y by the i-hub v and the algorithm would not have executed Line 19, which contradicts our assumption. $\qquad \square$

Lemma 5.7. *Let $1 \le i \le k$ and consider a pair of active $(i+1)$-centers (x', y') and their i-children $(x_j, y_j)_{1 \le j \le l}$ (which are active i-centers). Then, for every node v, there are at most $q = 8$ pairs of i-children (x_j, y_j) of (x', y') such that $v \in Q(x_j, y_j, i)$.*

Proof. We show that at the last time the algorithm has called REFRESH(x', y', $i+1$) (where it determined the current i-children of (x', y')) there are at most $q = 8$ pairs of i-children (x_j, y_j) of (x', y') such that $v \in \mathcal{P}(x_j, y_j, 2h_i, G)$ as, for each $1 \le i \le l$, $Q(x_j, y_j, i)$ (even if initialized later) will always be a subset of this set.

Suppose that v is contained in $q > 8$ path unions $\mathcal{P}(x_j, y_j, 2h_i, G)$ of i-children (x_j, y_j) of (x', y'). Let $j_1, \ldots, j_q$ be the corresponding indices and assume without loss of generality that $j_1 < j_2 \ldots < j_q$. The children of (x', y') all lie on the shortest path from x' to y' and by the way we have selected them we even more have $d_G(x_j, y_j) \ge h_i/2$ for all $1 \le j \le l$ and thus

$$d_G(x_{j_1}, y_{j_q}) \ge \sum_{j \in \{j_1, \ldots, j_q\}} d_G(x_j, y_j) \ge \sum_{j \in \{j_1, \ldots, j_q\}} h_i/2 = qh_i/2 \,. \qquad (5)$$

25

Furthermore, since $v \in \mathcal{P}(x_{j_1}, y_{j_1}, 2h_i, G)$, v lies on a shortest path from x_{j_1} to y_{j_1} of length at most h_i in G and thus $d_G(x_{j_1}, v) \leq 2h_i$. By the same argument we have $d_G(v, y_{j_q}) \leq 2h_i$. We now have

$$d_G(x_{j_1}, y_{j_q}) \leq d_G(x_{j_1}, v) + d_G(v, y_{j_q}) \leq 4h_i \tag{6}$$

by the triangle inequality. Observe that the upper bound 6 and the lower bound 5 contradict each other for $q > 8$ and thus $q \leq 8$. $\qquad\square$

Lemma 5.8. *For every $1 \leq i \leq k+1$ and every node v, there are at most q^{k-i+1} pairs of active i-centers (x, y) such that $v \in Q(x, y, i)$.*

Proof. The proof is by induction on i. The base case is $i = k+1$ and is trivially true because the only pair of $(k+1)$-centers is the pair (s, t). Consider now the case $1 \leq i \leq k$ and fix some node v. Consider a pair (x, y) of active i-centers such that $v \in Q(x, y, i)$. Let (x', y') be a pair of $(i+1)$-centers that is an $(i+1)$-parent of (x, y) (such a parent must exist because otherwise (x, y) would not be active). Since $Q(x, y, i) \subseteq Q(x', y', i+1)$ by Lemma 5.4, $v \in Q(x', y', i+1)$. Thus, $v \in Q(x, y, i)$ only if there is an $(i+1)$-parent (x', y') such that $v \in Q(x', y', i+1)$.

By the induction hypothesis, the number of pairs of active $(i+1)$-centers (x', y') such that $v \in Q(x', y', i+1)$ is at most $q^{k-(i+1)+1} = q^{k-i}$. Let (x', y') be such a pair of $(i+1)$-centers. By Lemma 5.7, the number of i-children (x, y) of (x', y') such that $v \in Q(x, y, i)$ is at most q. Therefore the total number of active pairs of i-centers (x, y) such that $v \in Q(x, y, i)$ is at most $q \cdot q^{k-i} = q^{k-i+1}$ as desired. $\qquad\square$

Lemma 5.9. *For every $0 \leq i \leq k$, each active pair of i-centers has at most $q^{k-i} \leq q^k$ $(i+1)$-parents.*

Proof. Consider any active pair of i-centers (x, y) and fix some node $v \in Q(x, y, i)$. Let the pair of $(i+1)$-centers (x', y') be an $(i+1)$-parent of (x, y). By Lemma 5.4 $Q(x, y, i) \subseteq Q(x', y', i+1)$. Thus, $v \in Q(x', y', i+1)$ for every $(i+1)$-parent of (x, y). If (x, y) had more than q^{k-i} $(i+1)$-parents, then v would be contained in more than $q^{k-i} = q^{k-(i+1)+1}$ path unions $Q(x', y', i+1)$ of $(i+1)$-parents (x', y'), contradicting Lemma 5.8. $\qquad\square$

Maintaining Hub Links. For every i-hub, we maintain an incoming and an outgoing ES-tree of depth $2h_i = 2n/c_i$, which takes time $O(mn/c_i)$. As there are $\tilde{O}(b_i)$ hubs of priority i, the total time needed for maintaining all these ES-trees is $\tilde{O}(\sum_{1 \leq i \leq k} b_i mn/c_i)$. For every pair of i-centers (x, y), the list of hubs linking x to y is initialized by iterating over all i-hubs. As there are $\tilde{O}(c_i)$ i-centers and $\tilde{O}(b_i)$ i-hubs, this takes time $\tilde{O}(\sum_{1 \leq i \leq k} b_i c_i^2)$. Every time the level of an i-center x in the ES-tree of an i-hub z exceeds $2h_i$, we have to remove z from the set of hub links for every possible partner y of x. As this event can occur only once for every i-center x and every i-hub z over the course of the algorithm, maintaining the sets of linking hubs takes time $\tilde{O}(\sum_{1 \leq i \leq k} b_i c_i^2)$.

Computing Path Unions. We now argue about the time needed for maintaining the sets $Q(x, y, i)$ for all pairs of i-centers (x, y) and each $1 \leq i \leq k$. We will show that we can pay for this cost by charging $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ to every pair of i-centers and

$O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$ to every refresh operation of the form REFRESH$(x', y', i+1)$ for some pair of $(i+1)$-centers (x', y'). Note that when $i = k$ then $\min(m/b_{k+1}, n^2/b_{k+1}^2) = m$ since we have set $b_{k+1} = 1$.

Fixing some pair of i-centers (x, y), we first bound the cost for the first initialization of $Q(x, y, i)$, as performed in Line 19 of Algorithm 1. There we have to compute $\mathcal{P}(x, y, 2h_i, G[Q(x', y', i + 1)])$, where (x', y') is an $(i+1)$-parent of (x, y). By Lemma 2.7 this takes time proportional to the number of edges in $G[Q(x', y', i + 1)]$. If $i = k$, then $Q(x', y', i+1) = V$ (and actually $x' = s$ and $y' = t$) and thus computing $\mathcal{P}(x, y, 2h_i, G)$ for all pairs of k-centers (x, y) takes time $\tilde{O}(c_k^2 m)$. If $1 \le i \le k - 1$, then $G[Q(x', y', i + 1)]$ has $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ edges by Lemma 5.6. Thus, the first computation of $Q(x, y, i)$ for all pairs of i-centers (x, y) takes time $\tilde{O}(c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2))$.

Now consider the cost of computing $Q(x, y, i)$ after it has already been initialized for the first time. Let $Q'(x, y, i) = \mathcal{P}(x, y, 2h_i, G[Q(x, y, i)])$ denote the updated path union as computed in Line 21 of Algorithm 1. The cost of computing $Q'(x, y, i)$ is proportional to $|E[Q(x, y, i)]|$, the number of edges in $G[cQ(x, y, i)]$ before the recomputation. Since $Q'(x, y, i) \subseteq Q(x, y, i)$, we have $E[Q(x, y, i)] = E[Q'(x, y, i)] \cup E[Q(x, y, i)] \setminus E[Q'(x, y, i)]$. Note that $Q'(x, y, i)$ is equal to $\mathcal{P}(x, y, 2h_i, G)$. We pay for this cost by charging $O(|E[Q(x, y, i)] \setminus E[Q'(x, y, i)]|)$ to the pair (x, y) and $|E[\mathcal{P}(x, y, 2h_i, G)]|$ to the refresh operation on the $(i+1)$-parent of (x, y) which causes the recomputation.

As the initial size of $G[Q(x, y, i)]$ is $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$ and we only charge edges to the pair (x, y) that will never be contained in $G[Q(x, y, i)]$ anymore, the total time charged to (x, y) is $O(\min(m/b_{i+1}, n^2/b_{i+1}^2))$, which, over all path union computations, results in a cost of $O(\sum_{1 \le i \le k-1} c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2) + c_k^2 m)$. It remains to bound the total cost charged to each refresh operation REFRESH$(x', y', i+1)$ for some pair of $(i+1)$-centers (x', y'). Below we will then separately analyze the total cost of the refresh operations. During the refresh operation we recompute $Q(x, y, i)$ for every i-child (x, y) of (x', y'), for which we have to pay $O(|E[\mathcal{P}(x, y, 2h_i, G)]|)$ per child. By Lemma 5.4 $Q(x, y, i) \subseteq Q(x', y', i + 1)$ for every i-child (x, y) of (x', y'). By Lemma 5.8, each node is contained in at most q^k path unions of i-children of (x', y'). Therefore every edge of $G[Q(x', y', i + 1)]$ is contained in the subgraphs of G induced by the i-path union of at most q^k i-children of (x', y'). Thus, the total cost charged to the refresh operation is $O(q^k |E[Q(x', y', i + 1)]|)$, which is $O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$.

Cost of Refresh. Excluding the recursive calls, each refresh operation of the form REFRESH(x, y, i), where $1 \le i \le k + 1$ and (x, y) is a pair of i-centers, is dominated by two costs: (1) the time needed for computing the shortest path from x to y in $G[Q(x, y, i)]$ and (2) the time needed for computing $Q(x', y', i - 1)$ for every $(i-1)$-child (x', y') of (x, y) if $i \ge 2$. As $G[Q(x, y, i)]$ has at most $\min(m/b_{i+1}, n^2/b_{i+1}^2)$ edges by Lemma 5.6, computing the shortest path takes time $\tilde{O}(\min(m/b_{i+1}, n^2/b_{i+1}^2))$. We have argued above that to pay for step (2) the time we charge to that particular refresh is $O(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$. It remains to analyze how often the refresh operation is called.

We say that a pair of i-centers (x, y) (for $0 \le i \le k$) causes a refresh if the algorithm calls REFRESH $(x', y', i + 1)$ (in line 7), where (x', y') is an $(i+1)$-parent of (x, y), after detecting that the distance from x to y in $G[Q(x, y, i)]$ is more than h_i. If a fixed pair of i-centers (x, y) causes a refresh after some deletion in the graph, it will do so for each of its $(i+1)$-parents. By Lemma 5.9 the number of $(i+1)$-parents is at most q^k. Note that

each pair of i-centers (x, y) will only cause these q^k refreshes of its parents once. At the time the algorithm makes (x, y) the child of some $(i+1)$-center we have $d_G(x, y) \leq h_i$ whp as by the initial random sampling of i-centers, every shortest path consisting of $h_i/2 - 1$ edges contains an i-center whp (Lemma 2.11). Furthermore, the pair (x, y) will never cause a refresh anymore in the future as the refresh implies that $d_G(x, y) > h_i$ by Lemma 5.5 and thus (x, y) will never be active anymore.

Now whenever we refresh a pair of $(i+1)$-centers (x', y'), we charge the running time of $\tilde{O}(q^k \min(m/b_{i+1}, n^2/b_{i+1}^2))$ to the i-child (x, y) causing the refresh. By the argument above, each pair of i-centers will be charged at most q^k times. Thus, the total time needed for all refresh operations on pairs of $(i+1)$-centers over the course of the algorithm is $\tilde{O}(q^{2k} c_i^2 \min(m/b_{i+1}, n^2/b_{i+1}^2))$ if $i \geq 1$. For $i = 0$, we can bound this by $\tilde{O}(q^{2k} m \cdot \min(m/b_1, n^2/b_1^2))$ because every node is a 0-center and thus a pair of 0-centers (x, y) can only be active if the graph contains the edge (x, y).

Total Running Time. Putting everything together, the total running time of our algorithm using k layers is

$$\tilde{O}\left(\sum_{1 \leq i \leq k} b_i c_i^2 + \sum_{1 \leq i \leq k} \frac{b_i m n}{c_i} + q^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2} \right) \right.$$

$$\left. + \sum_{1 \leq i \leq k-1} q^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2} \right) + q^{2k} c_k^2 m \right) .$$

We first balance the terms to obtain a running time of $\tilde{O}(m^{5/4} n^{1/2})$. We achieve this by setting the parameters to $k = \lceil \log \log m \rceil$ and, for every $1 \leq i \leq k$,

$$b_i = \frac{m^{\frac{3 \cdot (2^k - 2^{(i-1)})}{2^{(k+2)} - 3}}}{n^{\frac{2^{(k+1)} - 2^i}{2^{(k+2)} - 3}}}$$

$$c_i = 2^{k-i} m^{\frac{2^{(k+1)} - 3 \cdot 2^{(i-1)}}{2^{(k+2)} - 3}} n^{\frac{2^i - 1}{2^{(k+2)} - 3}} .$$

With this choice of the parameters we get (for all $1 \leq i \leq k$), $b_i \geq b_{i+1}$, $c_i \geq 2c_{i+1}$,

$$\frac{b_i m n}{c_i} \leq \frac{m^2}{b_1} = m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} ,$$

$$\frac{c_i^2 m}{b_{i+1}} \leq c_k^2 m \leq 2^{2k} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} , \text{ and}$$

$$b_i c_i^2 \leq 2^{2k} \frac{m^{\frac{7 \cdot 2^k - 9 \cdot 2^{(i-1)}}{2^{(k+2)} - 3}}}{n^{\frac{2^{(k+1)} - 3 \cdot 2^i + 2}{2^{(k+2)} - 3}}} \leq 2^{2k} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} ,$$

where the last inequality holds due to $m \leq n^2$ and $i \geq 0$. Thus, the total update time is

$$\tilde{O}\left(k 2^{2k} q^{2k} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} \right) .$$

Now observe that

$$m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} = m^{\frac{5}{4} + \frac{3}{4} \cdot \frac{1}{2^{(k+2)} - 3}} n^{\frac{1}{2} - \frac{1}{2} \frac{1}{2^{(k+2)} - 3}} \leq m^{\frac{5}{4} + \frac{3}{4} \cdot \frac{1}{2^{(k+2)} - 3}} n^{\frac{1}{2}} \,.$$

By our choice of $k = \lceil \log \log m \rceil$ we have

$$m^{\frac{3}{4} \cdot \frac{1}{2^{(k+2)} - 3}} \leq m^{\frac{1}{2^k}} \leq m^{\frac{1}{2 \log \log m}} = m^{\frac{1}{\log m}} = 2 \,.$$

Thus, the running time of our algorithm is $\tilde{O}(k q^{2k} m^{5/4} n^{1/2})$. With $k = \lceil \log \log m \rceil$ and $q = 8$ this is $\tilde{O}(m^{5/4} n^{1/2})$.

We now balance the terms to obtain a running time of $O(m^{4/3} n^{2/3 + o(1)})$. We achieve this by setting the parameters to $k = \lfloor \sqrt{\log n / \log q} \rfloor$ and, for every $1 \leq i \leq k$,

$$b_i = \frac{n^{(k+2i-1)/(3k+1)}}{m^{(4i-k-3)/(6k+2)}}$$
$$c_i = 2^{k-i} n^{2i/(3k+1)} m^{(3k+1-4i)/(6k+2)} \,.$$

With this choice of the parameters we get (for all $1 \leq i \leq k$), $b_i \geq b_{i+1}$, $c_i \geq 2c_{i+1}$,

$$\frac{b_i m n}{c_i} \leq \frac{m^2}{b_1} = n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} \,,$$
$$\frac{c_i^2 m}{b_{i+1}} \leq c_k^2 m \leq 2^{2k} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} \,,$$

and

$$b_i c_i^2 \leq 2^{2k} n^{(k+6i-1)/(3k+1)} m^{(7k+5-12i)/(6k+2)}$$
$$= 2^{2k} n^{(k-1)/(3k+1)} m^{(3k+1)/(6k+2)} m^{(2k+2)/(3k+1)} n^{6i/(3k+1)} m^{-6i/(3k+1)}$$
$$\leq 2^{2k} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} \,,$$

where the last inequality holds due to $n \leq m$ and $m \leq n^2$. Thus, the total update time is

$$\tilde{O} \left(k 2^{2k} q^{2k} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} \right) \,.$$

Now observe that

$$m^{\frac{2k+2}{3k+1}} n^{\frac{4k}{3k+1}} = m^{\frac{2}{3} + \frac{4}{3} \cdot \frac{1}{3k+1}} n^{\frac{4}{3} - \frac{4}{3} \cdot \frac{1}{3k+1}} \leq m^{\frac{2}{3} + \frac{4}{3} \cdot \frac{1}{3k+1}} n^{\frac{4}{3}} \leq m^{\frac{2}{3} + \frac{1}{k}} n^{\frac{4}{3}}$$

By our choice of $k = \lfloor \sqrt{\log n / \log q} \rfloor$ we have $2^k \leq q^k \leq n^{1/k}$. Since $k \leq \log n$ and $q = 8$ we thus obtain a total update time of $O(m^{2/3} n^{4/3 + O(1/\sqrt{\log n})}) = O(m^{2/3} n^{4/3 + o(1)})$.

Theorem 5.10. *There is a decremental* stR *algorithm with constant query time and total update time*

$$\tilde{O}(\min(m^{5/4} n^{1/2}, m^{2/3} n^{4/3 + o(1)})) = O(m n^{6/7 + o(1)})$$

that is correct with high probability against an oblivious adversary.

5.4 Extension to Single-Source Reachability

The algorithm above maintains reachability from a single source s to a single sink t. We can easily modify the algorithm to maintain reachability for a set of source-sink pairs $(s_i, t_i)_{1 \le i \le p}$. We simply run p instances of the algorithm, each with a different source-sink pair. Note however that the algorithm can use the same set of hubs and centers for all instances. Thus, the cost of maintaining the ES-trees does *not* have to be multiplied by p. We therefore get a total running time of

$$\tilde{O}\left(\sum_{1 \le i \le k} b_i c_i^2 + \sum_{1 \le i \le k} \frac{b_i mn}{c_i} + pq^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2}\right) \right.$$

$$\left. + \sum_{1 \le i \le k-1} pq^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2}\right) + pq^{2k} c_k^2 m \right) .$$

By setting

$$b_i = \frac{m^{\frac{3 \cdot (2^k - 2^{(i-1)})}{2^{(k+2)} - 3}} p^{\frac{2^{(k+1)} - 2^i}{2^{(k+2)} - 3}}}{n^{\frac{2^{(k+1)} - 2^i}{2^{(k+2)} - 3}}}$$

$$c_i = \frac{m^{\frac{2^{(k+1)} - 3 \cdot 2^{(i-1)}}{2^{(k+2)} - 3}} n^{\frac{2^i - 1}{2^{(k+2)} - 3}}}{p^{\frac{2^i - 1}{2^{(k+2)} - 3}}}$$

we get

$$pc_k^2 m = pm^2/b_1 = b_i mn/c_i = pc_i^2 m/b_{i+1} = m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} p^{\frac{2^{(k+1)} - 1}{2^{(k+2)} - 3}} .$$

$$\frac{b_i mn}{c_i} \le \frac{pm^2}{b_1} = p^{\frac{2^{(k+1)} - 1}{2^{(k+2)} - 3}} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} ,$$

$$\frac{pc_i^2 m}{b_{i+1}} \le pc_k^2 m \le 2^{2k} p^{\frac{2^{(k+1)} - 1}{2^{(k+2)} - 3}} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} , \text{ and}$$

$$b_i c_i^2 \le 2^{2k} \frac{p^{\frac{2^{(k+1)} - 3 \cdot 2^i + 2}{2^{(k+2)} - 3}} m^{\frac{7 \cdot 2^k - 9 \cdot 2^{(i-1)}}{2^{(k+2)} - 3}}}{n^{\frac{2^{(k+1)} - 3 \cdot 2^i + 2}{2^{(k+2)} - 3}}} \le 2^{2k} p^{\frac{2^{(k+1)} - 1}{2^{(k+2)} - 3}} m^{\frac{5 \cdot 2^k - 3}{2^{(k+2)} - 3}} n^{\frac{2^{(k+1)} - 2}{2^{(k+2)} - 3}} ,$$

and by setting

$$b_i = \frac{p^{(k+1-i)/(3k+1)} n^{(k+2i-1)/(3k+1)}}{m^{(4i-k-3)/(6k+2)}}$$

$$c_i = \frac{n^{2i/(3k+1)} m^{(3k+1-4i)/(6k+2)}}{p^{i/(3k+1)}} .$$

we get

$$\frac{b_i mn}{c_i} \le \frac{pm^2}{b_1} = p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} ,$$

$$\frac{pc_i^2 m}{b_{i+1}} \le pc_k^2 m \le 2^{2k} p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} ,$$

and

$$b_i c_i^2 = 2^{2k} p^{(k+1-3i)/(3k+1)} n^{(k+6i-1)/(3k+1)} m^{(7k+5-12i)/(6k+2)}$$
$$\leq 2^{2k} p^{(k+1)/(3k+1)} n^{4k/(3k+1)} m^{(2k+2)/(3k+1)} .$$

By the same choices of k as in the s-t reachability algorithm above we get a running times of $\tilde{O}(p^{1/2} m^{5/4} n^{1/2})$ and $O(p^{1/3} m^{2/3} n^{4/3+o(1)})$, respectively, for maintaining reachability between p source-sink pairs.

Corollary 5.11. *There is a decremental algorithm for maintaining reachability of p source-sink pairs with constant query time and total update time*

$$\tilde{O}(\min(p^{1/2} m^{5/4} n^{1/2}, p^{1/3} m^{2/3} n^{4/3+o(1)}))$$

that is correct with high probability against an oblivious adversary.

Using the reduction of Theorem 4.1 this immediately implies single-source reachability algorithm with a total update time of $\tilde{O}(m^{7/6} n^{2/3})$ (we balance the terms $p^{1/2} m^{5/4} n^{1/2}$ and mn/p by setting $p = n^{1/3}/m^{1/6}$) and $O(m^{3/4} n^{5/4+o(1)})$ (balance $p^{1/3} m^{2/3} n^{4/3}$ and mn/p by setting $p = m^{1/4}/n^{1/4}$).

Corollary 5.12. *There is a decremental* SSR *algorithm with constant query time and total update time*

$$\tilde{O}(\min(m^{7/6} n^{2/3}, m^{3/4} n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$$

that is correct with high probability against an oblivious adversary.

Furthermore, the reduction of Theorem 4.2 gives a decremental algorithm for maintaining strongly connected components.

Corollary 5.13. *There is a decremental* SCC *algorithm with constant query time and expected total update time*

$$\tilde{O}(\min(m^{7/6} n^{2/3}, m^{3/4} n^{5/4+o(1)})) = O(mn^{9/10+o(1)})$$

that is correct with high probability against an oblivious adversary.

6 Approximate Shortest Path

In the following we assume that G is a weighted graph with integer edge weights from 1 to W undergoing edge deletions. We are given some parameter $0 < \epsilon \leq 1$ and our goal is to maintain $(1 + \epsilon)$-approximate shortest paths. Similar to the reachability algorithm, we first give an algorithm for maintaining a $(1 + \epsilon)$-approximate shortest path from a given source s to a given sink t. We then extend it to an algorithm for maintaining $(1 + \epsilon)$-approximate shortest paths from a fixed source node to all other nodes. We give an algorithm that maintains a $(1 + \epsilon)^{2k+2}$-approximate shortest path from s to t. Note that by running the whole algorithm with $\epsilon' = \epsilon/(4k + 2)$ instead of ϵ we obtain a $(1 + \epsilon)$-approximate shortest path. Throughout this section we assume that $\epsilon \geq 1/\log^c n$ and $W \leq 2^{\log^c n}$ for some constant c

6.1 Preliminaries

The first new aspect for approximate shortest paths, compared to reachability, is that we maintain, for each $1 \leq i \leq k + 1$ and every pair of i-centers (x, y), an index $r(x, y, i)$ such that $d_G^h(x, y) \geq (1+\epsilon)^{r(x,y,i)}$ and, whenever (x, y) is active, $d_G(x, y) \leq (1+\epsilon)^{r(x,y,i)+2i}$. Thus, $r(x, y, i)$ indicates the current range of the distance from x to y. Roughly speaking, the algorithm will maintain a sequence of centers and for each pair of consecutive centers (x, y) a path from x to y of weight corresponding to the range of the distance from x to y. This will be done in a way such that whenever the algorithm cannot find such a path from x to y, then the range has increased. By charging this particular increase in the distance from x to y, it can afford updating the sequence of centers. As in the case of reachability such paths from x to y will either be found via a hub or in a path union graph.

The second new aspect is that now we maintain approximate shortest paths up to h_i *hops* for certain pairs of i-centers, whereas in the reachability algorithm h_i was an unweighted distance. This motivates the following modification of the path union that limits allowed paths to a fixed number of hops.

Definition 6.1. *For all nodes x and y of a graph G and all integers $h \geq 1$ and $D \geq 1$, the h-hop path union $\mathcal{P}^h(x, y, D, G)$ is the set containing every node that lies on some path π from x to y in G that has $|\pi| \leq h$ edges (hops) and weight $w(\pi, G) \leq D$.*

Observe that $\mathcal{P}(x, y, D, G) = \mathcal{P}^n(x, y, R, G)$ by Definition 2.6. Dealing with exact bounded-hop paths is usually computationally more expensive than dealing with unbounded paths, even in the static setting.[9] The bounded hop path unions will henceforth only be used in the analysis of the algorithm. In the algorithm itself we will compute unbounded (i.e., n-hop) path unions in graphs with modified edge weights. For every $h \geq 1$ and every $r \geq 0$ we define a graph $\tilde{G}^{h,r}$ that has the same nodes and edges as G and in which we round the weight of every edge (u, v) to the next multiple of $\epsilon(1 + \epsilon)^r / h$ by setting

$$w_{\tilde{G}^{h,r}}(u, v) = \left\lceil \frac{w_G(u, v) \cdot h}{\epsilon(1+\epsilon)^r} \right\rceil \cdot \frac{\epsilon(1+\epsilon)^r}{h} .$$

Note that the definition of $\tilde{G}^{h,r}$ implicitly depends on ϵ as well. To gain some intuition for these weight modifications, observe that for every path π in G consisting of h' edges we have

$$w_{\tilde{G}^{h,r}}(\pi) \leq w_G(\pi) + h' \cdot \frac{\epsilon(1+\epsilon)^r}{h} .$$

Thus, for every path π of weight $w_G(\pi) \geq (1 + \epsilon)^r$ consisting of at most h edges we have

$$w_{\tilde{G}^{h,r}}(\pi) \leq (1 + \epsilon)^r + \epsilon(1+\epsilon)^r = (1 + \epsilon)^{r+1} .$$

As additionally $w_G(u, v) \leq w_{\tilde{G}^{h,r}}(u, v)$ for every edge (u, v) we obtain the following guarantees.

Lemma 6.2. *Let $h \geq 1$ and $r \geq 0$. For all pairs of nodes x and y we have $d_G(x, y) \leq d_{\tilde{G}^{h,r}}(x, y)$ and if $d_G^h(x, y) \geq (1+\epsilon)^r$, then $d_{\tilde{G}^{h,r}}(x, y) \leq (1 + \epsilon)^{r+1}$.*

---

[9] For example, one can compute the shortest paths among all paths with at most h edges from a source node in time $O(mh)$ by running the first h iterations of the Bellman-Ford algorithm. This is not efficient enough for our purposes, as we would only like to spend nearly linear time.

Now observe that h-hop path unions can be computed "approximately" by computing the (unbounded) path union in $\tilde{G}^{h,r}$, which can be done in nearly linear time (Lemma 2.7). In our case the unbounded path union in $\tilde{G}^{h,r}$ will contain the h-hop path union and we will later argue that all unbounded path unions computed by our algorithm have small size. In our analysis the following lemma will have the same purpose as Lemma 2.9 for the s-t reachability algorithm.

Lemma 6.3. *Let (x,y) be a pair of nodes, let $h \geq 1$, $r \geq 0$, and $\alpha \geq 1$, and let $Q \subseteq V$ be a set of nodes such that $\mathcal{P}^h(x,y,\alpha(1+\epsilon)^r, G) \subseteq Q$. Then $\mathcal{P}^h(x,y,\alpha(1+\epsilon)^r, G) \subseteq \mathcal{P}(x,y,\alpha(1+\epsilon)^{r+1}, \tilde{G}^{h,r}[Q])$.*

Proof. Let $v \in \mathcal{P}^h(x,y,\alpha(1+\epsilon)^r, G)$, i.e., there is a path π from x to y in G with at most h edges and weight at most $\alpha(1+\epsilon)^r$ containing v. The weight of π in $\tilde{G}^{h,r}$ is

$$ w_{\tilde{G}^{h,r}}(\pi) \leq w_G(u,v) + h \cdot \frac{\epsilon(1+\epsilon)^r}{h} \leq \alpha(1+\epsilon)^r + \epsilon(1+\epsilon)^r \leq \alpha(1+\epsilon)^{r+1} . $$

By our assumption all nodes of π are contained in Q and thus π is a path in $\tilde{G}^{h,r}[Q]$ of weight at most $\alpha(1+\epsilon)^{r+1}$. Thus, all nodes of π, including v, are contained in $\mathcal{P}(x,y,\alpha(1+\epsilon)^{r+1}, \tilde{G}^{h,r}[Q])$. $\qquad\square$

The advantage of the graph $\tilde{G}^{h,r}$ is that its edge weights are multiples of $\epsilon(1+\epsilon)^r/h$. Thus, after *scaling down* the edge weights by a factor of $h/(\epsilon(1+\epsilon)^r)$ we still have integer weights. This observation can be used to speed up the pseudopolynomial algorithm of Even and Shiloach at the cost of a $(1+\epsilon)$-approximation [Ber09, Mad10, Ber13].

6.2 Algorithm Description

Our approximate s-t shortest path algorithm has a parameter $k \geq 1$ and for each $1 \leq i \leq k$ parameters $b_i \leq n$ and $c_i \leq n$. We also set $b_{k+1} = 1$, $c_0 = n$, $c_{k+1} = 1$, and $h_i = n/c_i$ for all $0 \leq i \leq k+1$. Our algorithm determines sets of nodes $B_1 \supseteq B_2 \supseteq \cdots \supseteq B_k$ and $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_{k+1}$ by random sampling as described in Section 5.1 for the s-t reachability algorithm, i.e., i-hubs are obtained by sampling nodes with probability $ab_i \ln(\Delta(G))/n$ and edges with probability $ab_i \ln(\Delta(G))/m$ and i-centers are obtained by sampling nodes with probability $ac_i \ln(\Delta(G))/n$. Note that, as we may assume $\Delta(G) \leq m \log_{1+\epsilon} W$ by Lemma 2.12 and we further assume that $\epsilon \geq 1/\log^c n$ and $W \leq 2^{\log^c n}$ for some constant c, the number of i-hubs is $\tilde{O}(b_i)$ whp and the number of i-centers is $\tilde{O}(c_i)$ whp by Lemma 2.11. We can make the algorithm fail with small probability if these sets are larger. For each $1 \leq i \leq k$ we order the set of i-hubs in an arbitrary way such that $B_i = \{b_1, b_2, \ldots, b_{|B_i|}\}$. Our algorithm uses the following variables and data structures:

- An estimate $\delta(s,t)$ of the distance from s to t in G

- For every pair of i-centers (x,y) (with $1 \leq i \leq k$) an index $r(x,y,i)$ such that $0 \leq r(x,y,i) \leq \log_{1+\epsilon}(nW)$ for the current range of the distance from x to y.

- For every i-hub z (with $1 \leq i \leq k$) and every $0 \leq r \leq \lfloor \log_{1+\epsilon}(nW) \rfloor$ an ES-tree up to depth $(1+\epsilon)^{r+i+1}$ in $\tilde{G}^{8h_i/\epsilon,r}$ (*outgoing* tree of z) and an ES-tree up to depth $(1+\epsilon)^{r+i+1}$ in the reverse graph of $\tilde{G}^{8h_i/\epsilon,r}$ (*incoming* tree of z).

33

- For every pair of i-centers (x, y) (with $1 \leq i \leq k$) an index $l(x, y, i)$ such that $1 \leq l(x, y, i) \leq |B_i| + 1$ that either gives the index $l(x, y, i)$ of the i-hub $b_{l(x,y,i)} \in B_i$ that links x to y for the current value of $r(x, y, i)$ or, if no such i-hub exists, is set to $l(x, y, i) = |B_i| + 1$.

- For every pair of i-centers (x, y) (with $1 \leq i \leq k$) a set of nodes $Q(x, y, i)$, which is initially empty.

- For every pair of i-centers (x, y) (with $0 \leq i \leq k$) a list of pairs of $(i+1)$-centers called $(i+1)$-parents of (x, y).

- For every pair of i-centers (x, y) (with $1 \leq i \leq k+1$) a list of pairs of $(i-1)$-centers called $(i-1)$-children of (x, y).

- For every $1 \leq i \leq k$, a list A_i of i-centers called *active i-centers*.

Besides the generalizations for weighted graphs introduced in Section 6.1, our approximate s-t shortest path algorithm also deviates from the s-t reachability algorithm in the way it maintains the hub links. The main difference to before is that now we maintain the hub links only between pairs of active centers. In the new algorithm we say, for $1 \leq i \leq k$, that an i-hub $z \in B_i$ *links* an i-center x to an i-center y if

$$d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(x, z) + d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(z, y) \leq (1 + \epsilon)^{r(x,y,i)+2i+1} \,.$$

The algorithm can check whether z links x to y by looking up $d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(x, z)$ and $d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(z, y)$ in the incoming and outgoing ES-tree of z, respectively. For every $1 \leq i \leq k$, and every pair of active i-centers x and y, the algorithm uses the index $l(x, y, i)$ to maintain an i-hub $b_{l(x,y,i)}$ that links x to y. If the current i-hub $b_{l(x,y,i)}$ does not link x to y anymore, the algorithm increases $l(x, y, i)$ until either such an i-hub is found, or $l(x, y, i) = |B_i| + 1$. As soon as $l(x, y, i) = |B_i| + 1$, the algorithm computes the path union between x and y and maintains an approximate shortest path from x to y in the subgraph induced by the path union. If the algorithm does not find such a path anymore it increases the value of $r(x, y, i)$ and sets $l(x, y, i)$ to 1 again, i.e., making the first i-hub the candidate for linking x to y for the new value of $r(x, y, i)$. The generalization of the s-t reachability algorithm is now straightforward and the pseudocode can be found in Algorithm 2. Before the first update the algorithms executes Procedure INITIALIZE. After each increase of the weight of an edge (u, v) or after its deletion the algorithm calls UPDATE(u, v).

6.3 Correctness

To establish the correctness of the algorithm we will show that

$$d_G(s, t) \leq \delta(s, t) \leq (1 + \epsilon)^{2k+1} d_G(s, t) \,.$$

By running the whole algorithm with $\epsilon' = \epsilon/(4k + 2)$ (instead of ϵ), we then obtain a $(1 + \epsilon)$-approximation (see Lemma 6.7 below).

We will achieve this by showing that with the value of $r(x, y, i)$ of a pair of i-centers (x, y) the algorithm keeps track of the range of the distance from x to y. In particular, we will show that $d_G^{h_i}(x, y) \geq (1+\epsilon)^{r(x,y,i)}$ for every pair of i-centers (x, y) and $d_G(x, y) \leq (1+\epsilon)^{r(x,y,i)+2i+1}$

Algorithm 2: Algorithm for decremental approximate *s-t* shortest path

---

1 Procedure REFRESH(x, y, i)

2   REMOVECHILDREN(x, y, i)

3   **if** $i = k + 1$ **then**

4   Compute shortest path π from s to t in G and set $\delta(s,t) \leftarrow (1+\epsilon)^{2k+1} d_G(s,t)$

5   **else**

6   Compute shortest path π from x to y in $\tilde{G}^{h_i,r(x,y,i)}[Q(x,y,i)(x,y)]$

7   **if** $d_{\tilde{G}^{h_i,r(x,y,i)}[Q(x,y,i)]}(x,y) > (1+\epsilon)^{r(x,y,i)+2}$ **then**   // Range has increased

8 $r(x,y,i) \leftarrow r(x,y,i) + 1$

9 $l(x,y,i) \leftarrow 1$

10   **foreach** $(i+1)$-*parent* (x',y') *of* (x,y) **do** REFRESH(x', y', $i+1$)

11   **break**

12   Determine $(i-1)$-centers $v_1, \ldots, v_l$ in order of appearance on π

13   **foreach** $1 \le j \le l-1$ **do**

14   Add (v_j, v_{j+1}) to A_{i-1}

15   Make (v_j, v_{j+1}) an $(i-1)$-child of (x,y) and (x,y) an i-parent of (v_j, v_{j+1})

16   **if** $i \ge 1$ **then**

17   UPDATEHUBLINKS(v_j, v_{j+1}, $i-1$, $r(v_j, v_{j+1}, i-1)$)

18   **if** $l_i(v_j, v_{j+1}) = |B_i| + 1$ **then**

19 $Q(v_j, v_{j+1}, i-1) \leftarrow$
$\mathcal{P}(v_j, v_{j+1}, (1+\epsilon)^{r(v_j,v_{j+1},i-1)+2i-1}, \tilde{G}^{8h_{i-1}/\epsilon, r(v_j,v_{j+1},i-1)}[Q(x,y,i)])$

20   REFRESH(v_j, v_{j+1}, $i-1$)

21 Procedure UPDATEHUBLINKS(x, y, i, r)

22   **while** $l(x,y,i) \le |B_i|$ **and**
$d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(x, b_{l(x,y,i)}) + d_{\tilde{G}^{8h_i/\epsilon, r(x,y,i)}}(b_{l(x,y,i)}, y) > (1+\epsilon)^{r(x,y,i)+2i+1}$ **do**

23 $l(x,y,i) \leftarrow l(x,y,i) + 1$

24   **if** $l(x,y,i) = |B_i| + 1$ **then**

25   Let (x',y') be any $(i+1)$-parent of (x,y)

26 $Q(x,y,i) \leftarrow \mathcal{P}(x,y,(1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)}[Q(x',y',i+1)])$

27 Procedure REMOVECHILDREN(x, y, i)

28   **foreach** $(i-1)$-*child* (x',y') *of* (x,y) **do**

29   Remove (x',y') from $(i-1)$-children of (x,y) and (x,y) from i-parents of (x',y')

30   **if** (x',y') *has no i-parents anymore* **then**

31   Remove (x',y') from A_{i-1}

32   **if** $i \ge 1$ **then** REMOVECHILDREN(x', y', $i-1$)

33 Procedure INITIALIZE()

34   **foreach** $1 \le i \le k$ *and all i-centers* (x,y) **do** $r(x,y,i) \leftarrow 0$, $l(x,y,i) \leftarrow 1$, $Q(x,y,i) \leftarrow \emptyset$

35   **foreach** $1 \le i \le k$ **do** $A_i \leftarrow \emptyset$

36   REFRESH(s, t, $k+1$)

37 Procedure UPDATE(u, v)   // Deletion or weight increase involving (u,v)

38   **foreach** 1-*parent* (x,y) *of* (u,v) **do** REFRESH(x, y, 1)

39   **foreach** $1 \le i \le k$ *and* $(x,y) \in A_i$ **do**

40   UPDATEHUBLINKS(x, y, i, $r(x,y,i)$)

41   **if** $l(x,y,i) = |B_i| + 1$ **then** REFRESH(x, y, i)

---

for every pair of active i-centers $(x, y) \in A_i$. Both inequalities are obtained by a "bottom up" analysis of the algorithm. In order to prove the lower bound on $d_G^{h_i}(x, y)$ we have to show that the subgraph $Q(x, y, i)$ computed by the algorithm indeed contains the corresponding path union. In fact those two invariants rely on each other and we will prove them mutually.

Lemma 6.4. *Algorithm 2 correctly maintains the following invariants whp:*

(I1) For every $1 \le i \le k$ and every pair of i-centers (x, y),

$$d_G^{h_i}(x, y) \ge (1 + \epsilon)^{r(x,y,i)} .$$

(I2) For every $1 \le i \le k$ and every pair of active i-centers $(x, y) \in A_i$,

$$\mathcal{P}^{8h_i/\epsilon}(x, y, (1 + \epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i) .$$

Proof. Both invariants trivially hold directly before the first refresh in Line 36 of the initialization is called because we set $r(x, y, i) = 0$ for all $1 \le i \le k$ and no pairs of centers are active yet.

We first give a proof of Invariant (I1). As distances are non-decreasing in G we only have to argue that the invariant still holds after each time the algorithm changes the value of $r(x, y, i)$. The only place where $r(x, y, i)$ is changed is in Line 8 (where it is increased by 1) and if this line is reached the condition

$$d_{\tilde{G}^{h_i, r(x,y,i)}[Q(x,y,i)]}(x, y) > (1 + \epsilon)^{r(x,y,i)+2}$$

holds.

Suppose that in G there is a path π from x to y with at most h_i edges and weight less than $(1 + \epsilon)^{r(x,y,i)+1}$. The weight of π in $\tilde{G}^{h_i, r(x,y,i)}$ is

$$\begin{aligned} w_{\tilde{G}^{h_i, r(x,y,i)}}(\pi) &\le w_G(\pi) + h_i \cdot \frac{\epsilon(1 + \epsilon)^{r(x,y,i)}}{h_i} \\ &\le (1 + \epsilon)^{r(x,y,i)+1} + \epsilon(1 + \epsilon)^{r(x,y,i)} \\ &\le (1 + \epsilon)^{r(x,y,i)+2} \\ &\le (1 + \epsilon)^{r(x,y,i)+2i} . \end{aligned}$$

By Invariant (I2) all nodes of π are contained in $Q(x, y, i)$. Therefore

$$d_{\tilde{G}^{h_i, r(x,y,i)}[Q(x,y,i)]}(x, y) \le w_{\tilde{G}^{h_i, r(x,y,i)}}(\pi) \le (1 + \epsilon)^{r(x,y,i)+2}$$

which contradicts the assumption that the algorithm has reached Line 8. Thus, the path π does not exist and it follows that $d_G^{h_i}(x, y) \ge (1 + \epsilon)^{r(x,y,i)+1}$ as desired. Therefore Invariant (I1) still holds when the algorithm increases the value of $r(x, y, i)$ by 1.

We now give a proof of Invariant (I2). The proof is by induction on i. There are two places in the algorithm where the value of $Q(x, y, i)$ is changed: Line 19 and Line 26. Note that $r(x, y, i)$ is non-decreasing and $l(x, y, i)$ decreases only if $r(x, y, i)$ increases. Therefore, for a fixed value of $r(x, y, i)$, the algorithm will "recompute" $Q(x, y, i)$ in Line 19 only if it has "initialized" it in Line 26 before.

Consider first the case that $Q(x, y, i)$ is "initialized" (Line 26) and let (x', y') be the $(i+1)$-parent of (x, y) used in this computation. If $i \leq k-1$, then by the induction hypothesis we know that

$$\mathcal{P}^{8h_{i+1}/\epsilon}(x', y', (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G) \subseteq Q(x', y', i+1). \tag{7}$$

If $i = k$, then this inclusion also holds because in that case $x' = s$ and $y' = t$ (as s and t are the only $(k+1)$-centers) and we have set $Q(s, t, k+1) = V$. We will show below that, at the last time the algorithm has called REFRESH(x', y', $i+1$) (where it sets (x, y) as an i-child of (x', y') and (x', y') as an $(i+1)$-parent of (x, y)), we have

$$\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq \mathcal{P}^{8h_{i+1}/\epsilon}(x', y', (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G)$$

and thus (together with (7))

$$\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x', y', i+1). \tag{8}$$

As distances in G are non-decreasing, (8) will still hold at the time the algorithm sets

$$Q(x, y, i) = \mathcal{P}(x, y, (1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)}[Q(x', y', i+1)]). \tag{9}$$

according to Line 26. By Lemma 6.3, (8) will imply that

$$\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G)$$
$$\subseteq \mathcal{P}(x, y, (1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)}[Q(x', y', i+1)]) = Q(x, y, i)$$

as desired. Note again that, since distances in G are nondecreasing, the inclusion $\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i)$ will then continue to hold until $Q(x, y, i)$ is "recomputed" (Line 19) for the first time. Whenever this happens, it will set new the value of $Q(x, y, i)$ equal to $\mathcal{P}(x, y, (1+\epsilon)^{r(x,y,i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x,y,i)}[Q(x, y, i)])$. Thus, by Lemma 6.3 we will again have $\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq Q(x, y, i)$ as demanded by Invariant (I2).

It remains to show that at the last time the algorithm has called REFRESH(x', y', $i+1$) and sets (x', y') as an $(i+1)$-parent of (x, y), we have

$$\mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G) \subseteq \mathcal{P}^{8h_{i+1}/\epsilon}(x', y', (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G).$$

Let π' denote the shortest path from x' to y' in $\tilde{G}^{h_{i+1}, r(x',y',i+1)}[Q(x', y', i+1)]$ computed at the beginning of the last execution of REFRESH(x', y', $i+1$). To enhance the readability of this part of the proof we use the abbreviation $H = \tilde{G}^{h_{i+1}, r(x',y',i+1)}[Q(x', y', i+1)]$. By the if-condition in Line 7 we know that $w(\pi', H) = d_H(x', y') \leq (1+\epsilon)^{r(x',y',i+1)+2}$.

Consider some node $v \in \mathcal{P}^{8h_i/\epsilon}(x, y, (1+\epsilon)^{r(x,y,i)+2i}, G)$ which means that v lies on a path π from x to y in G that has at most $8h_i/\epsilon$ edges and weight at most $(1+\epsilon)^{r(x,y,i)+2i}$. Remember that x and y are consecutive i-centers on π'. Let π'_1 and π'_2 denote the subpaths of π' from x' to x and y to y', respectively. The concatenation $\pi'' = \pi'_1 \circ \pi \circ \pi'_2$ is a path from x' to y' in G. We will show that π'' has at most $8h_{i+1}/\epsilon$ edges and weight at most $(1+\epsilon)^{r(x',y',i+1)+2(i+1)}$. This then proves that all nodes on π are contained in $\mathcal{P}^{8h_{i+1}/\epsilon}(x, y, (1+\epsilon)^{r(x',y',i+1)+2(i+1)}, G)$.

As $2h_i \leq h_{i+1}$, the number of edges of π is at most $8h_i/\epsilon \leq 4h_{i+1}/\epsilon$. Remember that the edge weights of H are multiples of $\epsilon(1+\epsilon)^{r(x',y',i+1)}/h_{i+1}$. Thus, each edge of H has weight

37

at least $\epsilon(1+\epsilon)^{r(x',y',i+1)}/h_{i+1}$. As the weight of π' in H is at most $(1+\epsilon)^{r(x',y',i+1)+2}$, the number of edges of π' is at most

$$\frac{(1+\epsilon)^{r(x',y',i+1)+2}h_{i+1}}{\epsilon(1+\epsilon)^{r(x',y',i+1)}} = \frac{(1+\epsilon)^2 h_{i+1}}{\epsilon} \leq \frac{4}{\epsilon}.$$

It follows that the number of edges of π'' is at most $4h_{i+1}/\epsilon + 4h_{i+1}/\epsilon = 8h_{i+1}/\epsilon$.

By Invariant (I1) we have $d_G^{h_i}(x,y) \geq (1+\epsilon)^{r(x,y,i)}$ and thus

$$w_G(\pi) \leq (1+\epsilon)^{r(x,y,i)+2i} \leq (1+\epsilon)^{2i} d_G^{h_i}(x,y).$$

By the initial random sampling of i-centers, every shortest path consisting of $h_i - 1$ edges contains an i-center whp (Lemma 2.11). Thus, the subpath of π' from x to y has at most h_i edges. Since π' is a shortest path in H we have $d_H(x,y) = d_H^{h_i}(x,y)$. It follows that

$$w_G(\pi) \leq (1+\epsilon)^{2i} d_G^{h_i}(x,y) \leq (1+\epsilon)^{2i} d_{G[Q(x',y',i+1)]}^{h_i}(x,y) \leq (1+\epsilon)^{2i} d_H^{h_i}(x,y)$$
$$= (1+\epsilon)^{2i} d_H(x,y).$$

Furthermore we have $d_H(x',y') = d_H(x',x) + d_H(x,y) + d_H$ (and in particular $d_H(x,y) \leq d_H(x',y')$). As $w_G(\pi'_1) \leq d_H(x',x)$ and $w_G(\pi'_2) \leq d_H(y,y')$ the weight of the path π'' in G can be bounded as follows:

$$\begin{aligned}
w_G(\pi'') &= w_G(\pi'_1) + w_G(\pi'_2) + w_G(\pi) \\
&\leq d_H(x',x) + d_H(y,y') + w_G(\pi) \\
&\leq d_H(x',y') - d_H(x,y) + w_G(\pi) \\
&\leq d_H(x',y') - d_H(x,y) + (1+\epsilon)^{2i} d_H(x,y) \\
&= d_H(x',y') + ((1+\epsilon)^{2i} - 1)d_H(x,y) \\
&\leq d_H(x',y') + ((1+\epsilon)^{2i} - 1)d_H(x',y') \\
&= (1+\epsilon)^{2i} d_H(x',y') \\
&\leq (1+\epsilon)^{2i}(1+\epsilon)^{r(x',y',i+1)+2} \\
&= (1+\epsilon)^{r(x',y',i+1)+2(i+1)}.
\end{aligned}$$

$\square$

Lemma 6.5. *For every $1 \leq i \leq k$ and every active pair of i-centers $(x,y) \in A_i$ we have $d_G(x,y) \leq (1+\epsilon)^{r(x,y,i)+2i+1}$ whp after the algorithm has finished its updates.*

Proof. The proof is by induction on i. If $l(x,y,i) \leq |B_i|$, then there is some i-hub z that links x to y. By the triangle inequality we then have

$$\begin{aligned}
d_G(x,y) &\leq d_G(x,z) + d_G(z,y) \\
&\leq d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(x,z) + d_{\tilde{G}^{8h_i/\epsilon,r(x,y,i)}}(z,y) \\
&\leq (1+\epsilon)^{r(x,y,i)+2i+1}.
\end{aligned}$$

If $l(x,y,i) = |B_i|+1$, then consider the last time the algorithm has called REFRESH(x, y, i). Let G' and $Q'(x,y,i)$ denote the versions of G and $Q(x,y,i)$ at the beginning of the refresh operation and let π be the shortest path from x to y in $(\tilde{G}')^{h_i,r(x,y,i)}[Q'(x,y,i)]$ computed

by the algorithm. To enhance readability we set $H = (\tilde{G}')^{h_i, r(x,y,i)}[Q'(x, y, i)]$ in this proof. The algorithm ensures that $d_H(x, y) \leq (1 + \epsilon)^{r(x,y,i)+2}$ as otherwise, by the if-condition in Line 7, (x, y) would have either become inactive (i.e., removed from A_i) or the algorithm would have called REFRESH(x, y, i) again. Let $v_1, \ldots, v_l$ denote the $(i-1)$-centers in order of appearance on π, i.e., $d_H(x, y) = \sum_{1 \leq j \leq l-1} d_H(v_j, v_{j+1})$.

If $i = 1$, then remember that the set of 0-centers is equal to V. Note that no edge (u, v) of π has been deleted from G since the last time the algorithm has called REFRESH(x, y, i) because (x, y) is a 1-parent of (u, v). Therefore all edges of π still exist in G. Since $r(x, y, i)$ is non-decreasing this means that $d_G(x, y) \leq d_H(x, y) \leq (1 + \epsilon)^{r(x,y,i)+2}$ as desired.

If $i \geq 2$, then let $1 \leq j \leq l - 1$. First of all, note that the value of $r(v_j, v_{j+1}, i)$ has not changed since the last time the algorithm has called REFRESH(x, y, i) because otherwise after Line 8, the algorithm would call REFRESH(x, y, i) as (x, y) is an i-parent of (v_j, v_{j+1}). Therefore, by Invariant (I1) of Lemma 6.4, we have $d_{G'}^{h_{i-1}}(v_j, v_{j+1}) \geq (1 + \epsilon)^{r(v_j, v_{j+1}, i-1)}$. Note that H has the same nodes and edges as G' and the weight of each edge in H is at least its weight in G'. Therefore $d_H^{h_{i-1}}(v_j, v_{j+1}) \geq d_{G'}^{h_{i-1}}(v_j, v_{j+1})$ and thus $d_H^{h_{i-1}}(v_j, v_{j+1}) \geq (1 + \epsilon)^{r(v_j, v_{j+1}, i-1)}$. By the initial random sampling of $(i - 1)$-centers, every shortest path consisting of $h_{i-1}-1$ edges contains an $(i-1)$-center whp (Lemma 2.11). Thus, the subpath of π from v_j to v_{j+1} has at most h_{i-1} edges which means that $d_H(v_j, v_{j+1}) = d_H^{h_{i-1}}(v_j, v_{j+1}) \geq (1 + \epsilon)^{r(v_j, v_{j+1}, i-1)}$.

By the induction hypothesis we have $d_G(v_j, v_{j+1}) \leq (1 + \epsilon)^{r(v_j, v_{j+1}, i-1)+2(i-1)+1}$ in the current version of G, which by the argument above implies that $d_G(v_j, v_{j+1}) \leq (1 + \epsilon)^{2(i-1)+1} d_H(v_j, v_{j+1})$. Thus, by the triangle inequality we get

$$
\begin{aligned}
d_G(x, y) &\leq \sum_{1 \leq j \leq l-1} d_G(v_j, v_{j+1}) \\
&\leq \sum_{1 \leq j \leq l-1} (1 + \epsilon)^{2(i-1)+1} d_H(v_j, v_{j+1}) \\
&= (1 + \epsilon)^{2i-1} d_H(x, y) \\
&\leq (1 + \epsilon)^{2i-1} (1 + \epsilon)^{r(x,y,i)+2} \\
&= (1 + \epsilon)^{2i+1}
\end{aligned}
$$

as desired. $\qquad\square$

Lemma 6.6. *After the algorithm has finished its updates we have* $d_G(s, t) \leq \delta(s, t) \leq (1 + \epsilon)^{k+1} d_G(s, t)$ *whp.*

Proof. Consider the last time the algorithm has called REFRESH(s, t, $k + 1$). Let G' denote the versions of G at the beginning of the refresh operation and let π be the shortest path from x to y in G computed by the algorithm. Note that $\delta(s, t) = (1+\epsilon)^{2k+1} d_{G'}(s, t)$. As distances are non-decreasing under deletions in G we trivially have $\delta(s, t) \leq (1 + \epsilon)^{k+1} d_G(s, t)$.

Let $v_1, \ldots, v_l$ denote the k-centers in order of appearance on π, which means that $d_{G'}(x, y) = \sum_{1 \leq j \leq l-1} d_{G'}(v_j, v_{j+1})$, and let $1 \leq j \leq l - 1$. The value of $r(v_j, v_{j+1}, k)$ has not changed since the last time the algorithm has called REFRESH(x, y, i) because otherwise after Line 8, the algorithm would call REFRESH(s, t, k) as (s, t) is a $(k + 1)$-parent of (v_j, v_{j+1}). By Lemma 6.4, we have $d_{G'}^{h_k}(v_j, v_{j+1}) \geq (1 + \epsilon)^{r(v_j, v_{j+1}, k)}$ and by Lemma 6.5 we have $d_G(v_j, v_{j+1}) \leq (1 + \epsilon)^{r(v_j, v_{j+1}, k)+2k+1}$. It follows that $d_G(v_j, v_{j+1}) \leq (1 + \epsilon)^{2k+1} d_{G'}(v_j, v_{j+1})$.

By the initial random sampling of k-centers, every shortest path consisting of $h_k - 1$ edges contains a k-center whp (Lemma 2.11). Thus, the subpath of π from v_j to v_{j+1} has at most h_k edges which means that $d_{G'}(v_j, v_{j+1}) = d_{G'}^{h_k}(v_j, v_{j+1})$. Thus, by the triangle inequality we get

$$d_G(s,t) \leq \sum_{1 \leq j \leq l-1} d_G(v_j, v_{j+1}) \leq \sum_{1 \leq j \leq l-1} (1+\epsilon)^{2k+1} d_{G'}(v_j, v_{j+1})$$

$$= (1+\epsilon)^{2k+1} d_{G'}(s,t) = \delta(s,t). \quad \square$$

Lemma 6.7. *For all $0 \leq x \leq 1$ and all $y > 0$,*

$$\left(1 + \frac{x}{2y}\right)^y \leq 1 + x.$$

Proof. Let e denote Euler's constant. We will use the following well-known inequalities: $(1 + 1/z)^z \leq e$ (for all $z > 0$), $e^z \leq 1/(1-z)$ (for all $z < 1$), and $1/(1-z) \leq 1 + 2z$ (for all $0 \leq z \leq 1/2$). We then get:

$$\left(1 + \frac{x}{2y}\right)^y = \left(\left(1 + \frac{x}{2y}\right)^{\frac{2y}{x}}\right)^{\frac{x}{2}} \leq e^{\frac{x}{2}} \leq \frac{1}{1 - \frac{x}{2}} \leq 1 + x. \qquad \square$$

6.4 Running Time

The running time analysis follows similar arguments as for the s-t reachability algorithm in Section 5.3. For every pair of i-centers (x, y), the algorithm never decreases $r(x, y, i)$ and (x, y) causes a refresh only if $r(x, y, i)$ increases, which happens $O(\log W/\epsilon)$ times. Furthermore it can be argued in a straightforward way that for every $1 \leq i \leq k+1$ and every pair of i-centers (x, y) the graph $G[Q(x, y, i)]$ has at most $\min(m/b_i, n^2/b_i^2)$ edges whp. Now the only differences compared to the running time analysis of the s-t reachability algorithm are the number of path unions of active pairs of centers each node is contained in and the time needed for maintaining the hub links, both of which are analyzed below.

Lemma 6.8. *Let $1 \leq i \leq k$ and consider a pair of active $(i+1)$-centers (x', y') and their i-children $(x_j, y_j)_{1 \leq j \leq l}$ (which are active i-centers). Then, for every node v, there are at most $q = 2^{2i+2}\lceil\log_{1+\epsilon}(nW)\rceil$ pairs of i-children (x_j, y_j) of (x', y') such that $v \in Q(x_j, y_j, i)$.*

Proof. We show that at the last time the algorithm has called REFRESH(x', y', $i+1$) (where it determined the current i-children of (x', y')) there are at most $q = 2^{2i+2}\lceil\log_{1+\epsilon}(nW)\rceil$ pairs of i-children (x_j, y_j) of (x', y') such that

$$v \in \mathcal{P}(x_j, y_j, (1+\epsilon)^{r(x_j, y_j, i)+2i+1}, \tilde{G}^{8h_i/\epsilon, r(x_j, y_j, i)}[Q(x', y', i+1)]).$$

Note that, for each $1 \leq i \leq l$, $r(x_j, y_j, i)$ has not changed since the last refresh (as such a change implies refreshing (x', y')). Thus, for each $1 \leq i \leq l$, $Q(x_j, y_j, i)$ is a subset of the path union above, which means that the lemma will be implied by this claim. Now we actually prove the following slightly stronger claim: for every every node v and every $0 \leq r \leq \lfloor\log_{1+\epsilon}(nW)\rfloor$, there are at most $q' = 2^{2i+2}$ pairs of i-children (x_j, y_j) of (x', y') such that

$$v \in \mathcal{P}(x_j, y_j, (1+\epsilon)^{r+2i+1}, \tilde{G}^{8h_i/\epsilon, r}[Q(x', y', i+1)]) \text{ and } d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r.$$

In this proof we use the abbreviation $H = \tilde{G}^{8h_i/\epsilon,r}[Q(x',y',i+1)]$. Suppose that v is contained in $q' > 2^{2i+2}$ path unions $\mathcal{P}(x_j, y_j, (1+\epsilon)^{r+2i+1}, H)$ of i-children (x_j, y_j) such that $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$. Let $j_1, \ldots, j_{q'}$ be the corresponding indices and assume without loss of generality that $j_1 < j_2 \ldots < j_{q'}$. By the initial random sampling of i-centers, every shortest path consisting of $h_i - 1$ edges contains an i-center whp (Lemma 2.11). Therefore, for every $1 \leq j \leq l$, there is a shortest path between x_j and y_j in H with at most h_i edges, i.e., $d_H(x_j, y_j) = d_H^{h_i}(x_j, y_j)$. Furthermore, for every $j \in \{j_1, \ldots, j_{q'}\}$, we have $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$ by our assumption and thus

$$d_H(x_{j_1}, y_{j_{q'}}) \geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_H(x_j, y_j) = \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_H^{h_i}(x_j, y_j)$$
$$\geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} d_G^{h_i}(x_j, y_j) \geq \sum_{j \in \{j_1, \ldots, j_{q'}\}} (1+\epsilon)^r = q'(1+\epsilon)^r \quad (10)$$

We now derive an upper bound on $d_H(x_{j_1}, y_{j_{q'}})$ contradicting this lower bound. Since v is contained in the path union $\mathcal{P}(x_j, y_j, (1+\epsilon)^{r+2i+1}, H)$, we know by the definition of the path union that v lies on a shortest path from x_{j_1} to y_{j_1} in H of weight at most $(1+\epsilon)^{r+2i+1}$ and thus $d_H(x_{j_1}, v) \leq (1+\epsilon)^{r+2i+1}$. The same argument shows that $d_H(v, y_{j_{q'}}) \leq (1+\epsilon)^{r+2i+1}$. By the triangle inequality we therefore have

$$d_H(x_{j_1}, y_{j_{q'}}) \leq d_H(x_{j_1}, v) + d_H(v, y_{j_{q'}}) \leq 2(1+\epsilon)^{r+2i+1} \quad (11)$$

By combining Inequalities (10) and (11) we get $q' \leq 2(1+\epsilon)^{2i+1}$, which is a contradictory statement for $q' > 2^{2i+2}$ because $\epsilon \leq 1$. Thus, v is contained in at most $q' = 2^{2i+2}$ path unions $\mathcal{P}(x_j, y_j, (1+\epsilon)^r, H)$ of i-children (x_j, y_j) such that $d_G^{h_i}(x_j, y_j) \geq (1+\epsilon)^r$. $\qquad \square$

We now bound the time needed for maintaining the hub links as follows. For every i-hub z (with $1 \leq i \leq k$) and every $0 \leq r \leq \log_{1+\epsilon}(nW)$ we maintain both an incoming and an outgoing ES-tree up to depth $(1+\epsilon)^{r+2i+1}$ in $\tilde{G}^{8h_i/\epsilon,r}$. In $\tilde{G}^{8h_i/\epsilon,r}$ every edge weight is a multiple of $\rho = \epsilon^2(1+\epsilon)^r/(8h_i)$ and thus we can scale down the edge weights by the factor $1/\rho$ and maintain the ES-tree in the resulting integer-weighted graph up to depth

$$\frac{(1+\epsilon)^{r+2i+1}}{\rho} \leq \frac{2^{2k+1}(1+\epsilon)^r}{\rho} = \frac{2^{2k+1}(1+\epsilon)^r 8h_i}{\epsilon^2(1+\epsilon)^r} = O(2^{2k}h_i/\epsilon^2) .$$

Thus, maintaining these two trees takes time $O(2^{2k}mh_i/\epsilon^2)$ which is $O(2^{2k}mn/(c_i\epsilon^2))$ as $h_i = n/c_i$. As we have $O(\log_{1+\epsilon}(nW)) = \tilde{O}(\log W/\epsilon)$ such trees for every i-hub and there are $\tilde{O}(b_i)$ i-hubs in, maintaining all these trees takes time $\tilde{O}(\sum_{1 \leq i \leq k} 2^{2k}b_imn\log W/(c_i\epsilon^3))$.

We now bound the time needed for maintaining the index $l(x,y,i)$ for every pair of i-centers (x,y). First, observe that $l(x,y,i)$ assumes integer values from 1 to $|B_i|$ (the number of i-hubs) and it only decreases (to 0) if $r(x,y,i)$ increases. The index $r(x,y,i)$ on the other hand is non-decreasing and assumes integer values from 0 to $\lceil \log_{1+\epsilon}(nW) \rceil = \tilde{O}(\log W/\epsilon)$. As $|B_i| = \tilde{O}(b_i)$, the value of $l(x,y,i)$ therefore changes $\tilde{O}(b_i \log W/\epsilon)$ times. As there are $\tilde{O}(2^k c_i)$ i-centers, the indices of all pairs of centers together change at most $\tilde{O}(\sum_{1 \leq i \leq k} 2^{2k}b_ic_i^2 \log W/\epsilon)$ times. It remains to bound the time spent in total for calls of the form UPDATEHUBLINKS(x, y, i, r) in which the index $l(x,y,i)$ does not increase but the algorithm still spends constant

time for checking whether $l(x, y, i)$ should increase. In such a case we charge the running time to the pair of i-centers (x, y). Note that the call UPDATEHUBLINKS(x, y, i, r) has either happened because (x, y) is made an i-child of some pair of $i+1$-centers or because (x, y) is an active pair of i-centers (i.e., in A_i) after the deletion of a node. We only have to focus on the second case because in the first case the charge of $O(1)$ on (x, y) can be neglected. As argued above, at any time, there are at most $O(q^k \log_{1+\epsilon}(nW))$ active i-centers. Furthermore, there are at most m deletions in G. Therefore the total time needed for maintaining all indices $l(x, y, i)$ is $\tilde{O}(\sum_{1 \leq i \leq k} 2^{2k} b_i c_i^2 \log W / \epsilon + q^k m \log W + \epsilon)$.

Putting everything together, the total running time of our algorithm using k layers is

$$
\tilde{O}\left(\sum_{1 \leq i \leq k} \frac{b_i c_i^2 \log W}{\epsilon} + \sum_{1 \leq i \leq k} \frac{b_i m n \log W}{c_i \epsilon^3} + q^{2k} m \cdot \min\left(\frac{m}{b_1}, \frac{n^2}{b_1^2}\right) \right.
$$

$$
\left. + \sum_{1 \leq i \leq k-1} q^{2k} c_i^2 \min\left(\frac{m}{b_{i+1}}, \frac{n^2}{b_{i+1}^2}\right) + 2^{2k} q^{2k} c_k^2 m \right)
$$

where $q = 2^{2k+2} \lfloor \log_{1+\epsilon} W \rfloor$. Assuming that $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ and setting $k = \lfloor \log^{1/4} n \rfloor$ we get $2^k q = O(n^{O(\log \log n / \log^{1/4} n)}) = O(n^{o(1)})$. We now simply set the parameters $b_i \leq n$ and $c_i \leq n$ for each $1 \leq i \leq k$ in the same way as in Section 5.3 to obtain the same asymptotic running time (in terms of polynomial factors) as for the s-t reachability algorithm.

Theorem 6.9. *For every $W \geq 1$ and every $0 < \epsilon \leq 1$ such that $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c, there is a decremental $(1 + \epsilon)$-approximate stSP algorithm with constant query time and total update time*

$$
O(\min(m^{5/4} n^{1/2+o(1)}, m^{2/3} n^{4/3+o(1)})) = O(mn^{6/7+o(1)})
$$

that is correct with high probability against an oblivious adversary.

Similar to the s-t reachability algorithm we can extend the $(1 + \epsilon)$-approximate stSP algorithm in the following ways.

Corollary 6.10. *For every $W \geq 1$ and every $0 < \epsilon \leq 1$ such that $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c, there is a decremental $(1 + \epsilon)$-approximate algorithm for maintaining shortest paths between p source-sink pairs with constant query time and total update time*

$$
O(\min(p^{1/2} m^{5/4} n^{1/2+o(1)}, p^{1/3} m^{2/3} n^{4/3+o(1)}))
$$

that is correct with high probability against an oblivious adversary.

Corollary 6.11. *For every $W \geq 1$ and every $0 < \epsilon \leq 1$ such that $W \leq 2^{\log^c n}$ and $\epsilon \geq 1/\log^c n$ for some constant c, there is a decremental $(1 + \epsilon)$-approximate SSSP algorithm with constant query time and total update time*

$$
O(\min(m^{7/6} n^{2/3+o(1)}, m^{3/4} n^{5/4+o(1)})) = O(mn^{9/10+o(1)})
$$

that is correct with high probability against an oblivious adversary.

7 Faster Single-Source Reachability in Dense Graphs

In this section we first introduce a path union data structure that is more efficient than the naive approach for repeatedly computing path unions between one fixed node and other variable nodes. We then show how to combine it with a multi-layer approach to obtain a faster decremental single-source reachability algorithm for dense graphs.

7.1 Approximate Path Union Data Structure

In the following we present a data structure for a graph G undergoing edge deletions, a fixed node x, and a parameter h. Given a node y, it computes an "approximation" of the path union $\mathcal{P}(x, y, h, G)$. Using a simple static algorithm the path union can be computed in time $O(m)$ for each pair (x, y). We give an (almost) output-sensitive data structure for this problem, i.e., using our data structure the time will be proportional to the size of the approximate path union which might be $o(m)$. Additionally, we have to pay a global cost of $O(m)$ that is amortized over *all* approximate path union computations for the node x and *all* nodes y. This will be useful because in our reachability algorithm we can use probabilistic arguments to bound the size of the approximate path unions.

Proposition 7.1. *There is a data structure that, given a graph G undergoing edge deletions, a fixed node x, and a parameter h, provides a procedure* APPROXIMATEPATHUNION *such that, given sequence of nodes $y_1, \ldots, y_k$, this procedure computes sets $F_1, \ldots F_k$ guaranteeing $\mathcal{P}(x, y, h, G) \subseteq F_i \subseteq \mathcal{P}(x, y, (\log m + 3)h, G)$ for all $1 \le i \le k$. The total running time is $O(\sum_{1 \le i \le k} |F_i| + m)$.*

7.1.1 Algorithm Description

Internally, the data structure maintains a set $R(x)$ of nodes, initialized with $R(x) = V$, such that the following invariant is fulfilled at any time: all nodes that can be reached from x by a path of length at most h are contained in $R(x)$ (but $R(x)$ might contain other nodes as well). Observe that thus $R(x)$ contains the path union $\mathcal{P}(x, y, h, G)$ for every node y.

To gain some intuition for our approach consider the following way of computing an approximation of the path union $\mathcal{P}(x, y, h, G)$ for some node y. First, compute $B_1 = \{v \in R(x) \mid d_{G[R(x)]}(v, y) \le h\}$ using a backward breadth-first search (BFS) to y in $G[R(x)]$, the subgraph of G induced by $R(x)$. Second, compute $F = \{v \in R(x) \mid d_{G[B_1]}(x, v) \le h\}$ using a forward BFS from x in $G[B_1]$. It can be shown that $\mathcal{P}(x, y, h, G) \subseteq F \subseteq \mathcal{P}(x, y, 2h, G)$.[10] Given B_1, we could charge the time for computing F to the set F itself, but we do not know how to pay for computing B_1 as $B_1 \setminus F$ might be much larger than F.

Our idea is to additionally identify a set of nodes $X \subseteq \{v \in V \mid d_G(x, v) > h\}$ and remove it from $R(x)$. Consider a second approach where we first compute B_1 as above and then compute $B_2 = \{v \in R(x) \mid d_{G[R(x)]}(v, y) \le 2h\}$ and $F = \{v \in R(x) \mid d_{G[B_2]}(x, v) \le h\}$. It can be shown that $\mathcal{P}(x, y, h, G) \subseteq F \subseteq \mathcal{P}(x, y, 3h, G)$. Additionally, all nodes in $X = B_1 \setminus F$ are at distance more than h from x and therefore we can remove X from $R(x)$. Thus, we

---

[10]Indeed, F might contain some node v with $d_G(x, v) = h$ and $d_G(v, y) = h$, but it will not contain any node w with either $d_G(x, w) > h$ or $d_G(w, y) > h$.

can charge the work for computing B_1 and F to X and F, respectively.[11] However, we now have a similar problem as before as we do not know whom to charge for computing B_2.

We resolve this issue by simply computing $B_i = \{v \in R(x) \mid d_{G[R(x)]}(v,y) \le ih\}$ for increasing values of i until we arrive at some i^* such that the size of B_{i^*} is at most double the size of B_{i^*-1}. We then return $F = \{v \in R(x) \mid d_{G[B_i]}(x,v) \le h\}$ and charge the time for computing B_i to $X = B_{i-1} \setminus F$ and F, respectively. As the size of B_i can double at most $O(\log n)$ times we have $\mathcal{P}(x,y,h,G) \subseteq F \subseteq \mathcal{P}(x,y,O(h\log n),G)$, as we show below. Procedure 3 shows the pseudocode of this algorithm. Note that in the special case that x cannot reach y the algorithm returns the empty set. In the analysis below, let i^* denote the final value of i before Procedure 3 terminates.

Procedure 3: APPROXIMATEPATHUNION(y)

    // All calls of APPROXIMATEPATHUNION(y) use fixed x and h.

1 Compute $B_1 = \{v \in R(x) \mid d_{G[R(x)]}(v,y) \le h\}$ // backward BFS **to** y in subgraph
       induced by $R(x)$

2 **for** $i = 2$ **to** $\lceil \log m \rceil + 1$ **do**

3   Compute $B_i = \{v \in R(x) \mid d_{G[R(x)]}(v,y) \le ih\}$ // backward BFS **to** y in
          subgraph induced by $R(x)$

4   **if** $|E[B_i]| \le 2|E[B_{i-1}]|$ **then**

5   Compute $F = \{v \in B_i \mid d_{G[B_i]}(x,v) \le h\}$ // forward BFS **from** x in
            subgraph induced by B_i

6 $X \leftarrow B_{i-1} \setminus F$, $R(x) \leftarrow R(x) \setminus X$

7   **return** F

7.1.2   Correctness

We first prove Invariant (I): the set $R(x)$ always contains all nodes that are at distance at most h from x in G. This is true initially as we initialize $R(x)$ to be V and we now show that it continues to hold because we only remove nodes at distance more than h from x.

Lemma 7.2. *If $R(x) \subseteq \{v \in V \mid d_G(x,v) \le h\}$, then for every node $v \in X$ removed from $R(x)$, we have $d_G(x,v) > h$.*

Proof. Let $v \in X = B_{i^*-1} \setminus F$ and assume by contradiction that $d_G(x,v) \le h$. Since $v \in B_{i^*-1}$ we have $d_{G[R(x)]}(v,y) \le (i^*-1)h$. Now consider the shortest path π from x to v in G, which has length at most h. By the assumption, every node on π is contained in $G[R(x)]$. Therefore, for every node v' on π, we have $d_{G[R(x)]}(v',v) \le h$ and thus

$$d_{G[R(x)]}(v',y) \le d_{G[R(x)]}(v',v) + d_{G[R(x)]}(v,y) \le h + (i^*-1)h \le i^*h$$

which implies that $v' \in B_{i^*}$. Thus, every node on π is contained in B_{i^*}. As π is a path from x to v of length at most h it follows that $d_{G[B_{i^*}]}(x,v) \le h$. Therefore $v \in F$, which contradicts the assumption $v \in X$. $\qquad\square$

---

[11]Note that in our first approach removing $B_1 \setminus F$ would not have been correct as F was computed w.r.t to $G[B_1]$ and not w.r.t. $G[B_2]$.

We now complete the correctness proof by showing that the set of nodes returned by the algorithm approximates the path union.

Lemma 7.3. *Procedure 3 returns a set of nodes F such that $\mathcal{P}(x, y, h, G) \subseteq F \subseteq \mathcal{P}(x, y, (\log m + 3)h, G)$.*

Proof. We first argue that the algorithm actually returns some set of nodes F. Note that in Line 4 of the algorithm we always have $|E[B_i]| \geq |E[B_{i-1}]|$ as $B_{i-1} \subseteq B_i$. As $E[B_i]$ is a set of edges and the total number of edges is at most m, the condition $|E[B_i]| \leq |E[B_{i-1}]|$ therefore must eventually be fulfilled for some $2 \leq i \leq \lceil \log m \rceil + 1$.

We now show that $\mathcal{P}(x, y, h, G) \subseteq F$. Let $v \in \mathcal{P}(x, y, h, G)$, which implies that v lies on a path π from x to y of length at most h. For every node v' on π we have $d_G(x, v') \leq h$, which by Invariant (I) implies $v' \in R(x)$. Thus, the whole path π is contained in $G[R(x)]$. Therefore $d_{G[R(x)]}(v', y) \leq h$ for every node v' on π which implies that π is contained in $G[B_{i^*}]$. Then clearly we also have $d_{G[B_{i^*}]}(x, v) \leq h$ which implies $v \in F$.

Finally we show that $F \subseteq \mathcal{P}(x, y, (\log m + 3)h, G)$ by proving that $d_G(x, v) + d_G(v, y) \leq (\log m + 3)h$ for every node $v \in F$. As $G[B_{i^*}]$ is a subgraph of G, we have $d_G(x, v) \leq d_{G[B_{i^*}]}(x, v)$ and $d_G(v, y) \leq d_{G[B_{i^*}]}(v, y)$. By the definition of F we have $d_{G[B_{i^*}]}(x, v) \leq h$. As $F \subseteq B_{i^*}$ we also have $d_{G[B_{i^*}]}(v, y) \leq i^* h \leq (\lceil \log m \rceil + 1)h \leq (\log m + 2)h$. It follows that $d_G(x, v) + d_G(v, y) \leq h + (\log m + 2)h = (\log m + 3)h$. $\qquad\square$

7.1.3 Running Time Analysis

To bound the total running time we prove that each call of Procedure 3 takes time proportional to the number of edges in the returned approximation of the path union plus the number of edges incident to the nodes removed from $R(x)$. As each node is removed from $R(x)$ at most once, the time spent on *all* calls of Procedure 3 is then $O(m)$ plus the sizes of the subgraphs induced by the approximate path unions returned in each call.

Lemma 7.4. *The running time of Procedure 3 is $O(|E[F]| + |E[X, R(x)]| + |E[R(x), X]|)$ where F is the set of nodes returned by the algorithm, and X is the set of nodes the algorithm removes from $R(x)$.*

Proof. The running time in iteration $2 \leq j \leq i^* - 1$ is $O(|E[B_j]|)$ as this is the cost of the breadth-first-search performed to compute B_j. In the last iteration i^*, the algorithm additionally has to compute F and X and remove X from $R(x)$. As F is computed by a BFS in $G[B_{i^*}]$ and $X \subseteq B_{i^*-1} \subseteq B_{i^*}$, these steps take time $O(|E[B_{i^*}]|)$. Thus the total running time is $O(\sum_{1 \leq j \leq i^*} |E[B_j]|)$.

By checking the size bound in Line 4 of Procedure 3 we have $|E[B_j]| > 2|E[B_{j-1}]|$ for all $1 \leq j \leq i^* - 1$ and $|E[B_{i^*}]| \leq 2|E[B_{i^*-1}]|$. By repeatedly applying the first inequality it follows that $\sum_{1 \leq j \leq i^* - 1} |E[B_j]| \leq 2|E[B_{i^*-1}]|$. Therefore we get

$$\sum_{1 \leq j \leq i^*} |E[B_j]| = \sum_{1 \leq j \leq i^*-1} |E[B_j]| + |E[B_{i^*}]| \leq 2|E[B_{i^*-1}]| + 2|E[B_{i^*-1}]| = 4|E[B_{i^*-1}]|$$

and thus the running time is $O(|E[B_{i^*-1}]|)$. Now observe that by $X = B_{i^*-1} \setminus F$ we have $B_{i^*-1} \subseteq X \cup F$ and thus

$$E[B_{i^*-1}] \subseteq E[F] \cup E[X] \cup E[X, F] \cup E[F, X] \subseteq E[F] \cup E[X, R(x)] \cup E[R(x), X].$$

Therefore the running time is $O(|E[F]| + |E[X, R(x)]| + |E[R(x), X]|)$. $\qquad\square$

7.2 Reachability via Center Graph

We now show how to combine the approximate path union data structure with a hierarchical approach to get an improved decremental reachability algorithm for dense graphs. The algorithm has a parameter $1 \leq k \leq \log n$ and for each $1 \leq i \leq k$ a parameter $c_i \leq n$. We determine suitable choices of these parameters later in this section. For each $1 \leq i \leq k-1$, our choice will satisfy $c_i \geq c_{i+1}$ and $c_i = \hat{O}(c_{i+1})$. Furthermore, we set $h_i = (3+\log m)^{i-1} n/c_1$ for $1 \leq i \leq k$. At the initialization, the algorithm determines sets of nodes $C_1 \supseteq C_2 \supseteq \cdots \supseteq C_k$ such that $s, t \in C_1$ as follows. For each $1 \leq i \leq k$, we sample each node of the graph with probability $ac_i \ln n/n$ (for a large enough constant a), where the value of c_i will be determined later. The set C_i then consists of the sampled nodes, and if $i \leq k-1$, it additionally contains the nodes in C_{i+1}. For every $1 \leq i \leq k$ we call the nodes in C_i i-centers. In the following we describe an algorithm for maintaining pairwise reachability between all 1-centers.

7.2.1 Algorithm Description

Data Structures. The algorithm uses the following data structures:

- For every i-center x and every $i \leq j \leq k$ an approximate path union data structure (see Proposition 7.1) with parameter h_j.

- For every k-center x an incoming and an outgoing ES-tree of depth h_k in G.

- For every pair of an i-center x and a j-center y such that $l := \max(i,j) \leq k-1$, a set of nodes $Q(x,y,l) \subseteq V$. Initially, $Q(x,y,l)$ is empty and at some point the algorithm might compute $Q(x,y,l)$ using the approximate path union data structure of x.

- For every pair of an i-center x and a j-center y such that $l := \max(i,j) \leq k-1$ an ES-tree of depth h_l from x in $Q(x,y,l)$.

- For every pair of an i-center x and a j-center y such that $l := \max(i,j) \leq k-1$ a set of $(l+1)$-centers certifying that x can reach y.

Certified Reachability Between Centers (Links). The algorithm maintains the following limited path information between centers, called *links*, in a top-down fashion. Let x be a k-center and let y be an i-center for some $1 \leq i \leq k-1$. The algorithm links x to y if and only if y is contained in the outgoing ES-tree of depth h_k of x. Similarly the algorithm links y to x if and only if y is contained in the incoming ES-tree of depth h_k of x. Let x be an i-center and let y be a j-center such that $l := \max(i,j) \leq k-1$. If there is an $(l+1)$-center z such that x is linked to z and z is linked to y, the algorithm links x to y (we also say that z links x to y). Otherwise, the algorithm computes $Q(x,y,l)$ using the approximate path union data structure of x and starts to maintain an ES-tree from x up to depth h_l in $G[Q(x,y,l)]$. It links x to y if and only if y is contained in the ES-tree of x. Using a list of centers z certifying that x can reach y, maintaining the links between centers is straightforward.

Center Graph. The algorithm maintains a graph called *center graph*. Its nodes are the 1-centers and it contains the edge (x, y) if and only if x is linked to y. The algorithm maintains the transitive closure of the center graph. A query asking whether a center y is reachable from a center x in G is answered by checking the reachability in the center graph. As s and t are 1-centers this answers *s-t* reachability queries.

7.2.2 Correctness

For the algorithm to be correct we have to show that there is a path from s to t in the center graph if and only if there is a path from s to t in G. We will in fact show in more generality that this is the case for any pair of 1-centers.

Lemma 7.5. *For every pair of an i-center x and a j-center y, if x is linked to y, then there is a path from x to y in G.*

Proof. The proof is by induction on $l = \max(i, j)$. As x is linked to y, one of the following three cases applies:

1. $j = k$ and x is contained in the incoming ES-tree of depth h_k of y in G.

2. $i = k$ and y is contained in the outgoing ES-tree of depth h_k of x in G.

3. There is an $(l + 1)$-center z such that x is linked to z and z is linked to y.

4. y is contained in the ES-tree of depth h_l of x in $G[Q(x, y, l)]$.

In the first two cases there obviously is a path from x to y in G by the correctness of the ES-tree. In the third case we may apply the induction hypothesis and find a path from x to z as well as a path from z to y in G. The concatenation of these paths is a path from x to y in G. In the fourth case we know by the correctness of the ES-tree that there is a path from x to y in $G[Q(x, y, l)]$ and therefore also in G. □

Lemma 7.6. *For every pair of an i-center x and a j-center y, if $d_G(x, y) \leq h_l$, then x is linked to y.*

Proof. Set $l = \max(i, j)$. If $l = k$, then assume that $l = i$ (the proof for $l = j$ is symmetric). Since $d_G(x, y) \leq h_k$, y is contained in the outgoing ES-tree of depth h_k of x by the correctness of the ES-tree. Thus, x is linked to y.

  If $l \leq k - 1$ and there is an $(l + 1)$-center z such that x is linked to z and z is linked to y, then also x is linked to y. If this is not the case, then the algorithm has computed $Q(x, y, l)$ using the approximate path union data structure and at that time we have $\mathcal{P}(x, y, h_l, G) \subseteq Q(x, y, l)$ by Proposition 7.1. By Lemma 2.8 the set $Q(x, y, l)$ contains $\mathcal{P}(x, y, h_l, G)$, also in the current version of G. As $d_G(x, y) \leq h_l$, all nodes on the shortest path from x to y in G are contained in $\mathcal{P}(x, y, h_l, G)$ and thus in $Q(x, y, l)$. Therefore the ES-tree from x up to depth h_l in $G[Q(x, y, l)]$ contains y which means that x is linked to y. □

Lemma 7.7. *For every pair of 1-centers x and y, there is a path from x to y in the center graph if and only if there is a path from x to y in G.*

Proof. Assume that there is a path from x to y in the center graph. Every edge (x', y') in the center graph (where x' and y' are 1-centers) can only exist if the algorithm has linked x' to y'. By Lemma 7.5 this implies that there is a path from x' to y' in G. By concatenating all these paths we obtain a path from x to y in G.

Now assume that there is a path from x to y in G. Consider the shortest path π from x to y in G and any two consecutive 1-centers x' and y' on π. By the initial random sampling of 1-centers, every shortest path consisting of $h_1 - 1$ edges contains a 1-center whp (Lemma 2.11) and thus $d_G(x', y') \leq h_1$. By Lemma 7.6 the algorithm has linked x' to y' and thus there is an edge from x' to y' in the center graph. As such an edge exists for all consecutive 1-centers on π, there is a path from x to y in the center graph. $\qquad\square$

7.2.3 Running Time Analysis

The key to the efficiency of the algorithm is to bound the size of the graphs $Q(x, y, l)$.

Lemma 7.8. *Let x be an i-center and let y be a j-center such that $l := \max(i, j) \leq k - 1$. If x is not linked to y by an $(l+1)$-center, then $Q(x, y, l)$ contains at most n/c_{l+1} nodes with high probability.*

Proof. Suppose that $Q(x, y, l)$ contains more than n/c_{l+1} nodes. Then by the random sampling of centers, $Q(x, y, l)$ contains an $(l+1)$-center z with high probability by Lemma 2.11. By Proposition 7.1 we have $d_G(x, z) + d_G(z, y) \leq (3 + \log m)h_l = h_{l+1}$ and thus $d_G(x, z) \leq h_{l+1}$ and $d_G(z, y) \leq h_{l+1}$. It follows that x is linked to z and z is linked to y by Lemma 7.6. But then x is linked to y, contradicting our assumption. $\qquad\square$

With the help of this lemma we first analyze the running time of each part of the algorithm and argue that our choice of parameters gives the desired total update time.

Parameter Choice. We carry out the running time analysis with regard to two parameters $1 \leq b \leq c \leq n$ which we will set at the end of the analysis. We set

$$k = \left\lceil \frac{\log(c/b)}{\sqrt{\log n \cdot \log \log n}} \right\rceil + 1 \,,$$

$c_k = b$ and $c_i = 2^{\sqrt{\log n \cdot \log \log n}} c_{i+1} = \hat{O}(c_{i+1})$ for $1 \leq i \leq k - 1$. Note that the number of i-centers is $\tilde{O}(c_i)$ whp. Observe that

$$(3 + \log m)^{k-1} = O((\log n)^k) \leq O((\log n)^{\sqrt{\log n / \log \log n}})$$
$$= O(2^{\sqrt{\log n \cdot \log \log n}}) = O(n^{\sqrt{\log \log n / \log n}}) = O(n^{o(1)}) \,.$$

Furthermore we have

$$c_1 = \left(2^{\sqrt{\log n \cdot \log \log n}}\right)^{k-1} c_k \geq 2^{\log(c/b)} b = \frac{c}{b} \cdot b = c$$

and by setting $k' = (\log(c/b))/(\sqrt{\log n \cdot \log \log n})$ we have $k \leq k' + 2$ and thus

$$c_1 = \left(2^{\sqrt{\log n \cdot \log \log n}}\right)^{k-1} c_k \leq \left(2^{\sqrt{\log n \cdot \log \log n}}\right)^{k'+1} c_k = 2^{\sqrt{\log n \cdot \log \log n}} c = \hat{O}(c) \,.$$

Remember that $h_i = (3 + \log m)^{i-1} n/c_1$ for $1 \leq i \leq k$. Therefore we have $h_i = \hat{O}(n/c_1) = \hat{O}(n/c)$.

Maintaining ES-Trees. For every k-center we maintain an incoming and an outgoing ES-tree of depth h_k, which takes time $O(mh_k)$. As there are $\tilde{O}(c_k)$ k-centers, maintaining all these trees takes time $\tilde{O}(c_k m h_k) = \hat{O}(bmn/c)$.

For every i-center x and every j-center y such that $l := \max(i,j) \leq k-1$, we maintain an ES-tree up to depth h_l in $G[Q(x,y,l)]$. By Lemma 7.8 $Q(x,y,l)$ has at most n/c_{l+1} nodes and thus $G[Q(x,y,l)]$ has at most n^2/c_{l+1}^2 edges. Maintaining this ES-tree therefore takes time $O((n^2/c_{l+1}^2) \cdot h_l) = \hat{O}(n^2/c_{l+1}^2(n/c_1)) = \hat{O}(n^3/(c_1 c_{l+1}^2))$. In total, maintaining all these trees takes time

$$\hat{O}\left(\sum_{1\leq i\leq k-1}\sum_{1\leq j\leq i} c_i c_j \frac{n^3}{c_1 c_{i+1}^2}\right) = \hat{O}\left(\sum_{1\leq i\leq k-1}\sum_{1\leq j\leq i} \frac{c_i c_1 n^3}{c_{i+1} c_1 c_k}\right)$$

$$= \hat{O}\left(\sum_{1\leq i\leq k-1}\sum_{1\leq j\leq i} \frac{n^3}{c_k}\right) = \hat{O}\left(k^2 \frac{n^3}{c_k}\right) = \hat{O}\left(\frac{n^3}{b}\right).$$

Computing Approximate Path Unions. For every i-center x and every $i \leq j \leq k$ we maintain an approximate path union data structure with parameter h_j. By Proposition 7.1 this data structures has a total running time of $O(m)$ and an additional cost of $O(|E[Q(x,y,j)]|)$ each time the approximate path union $Q(x,y,j)$ is computed for some j-center y. By Lemma 7.8 the number of nodes of $Q(x,y,j)$ is n/c_{j+1} with high probability and thus its number of edges is n^2/c_{j+1}^2. Therefore, computing all approximate path unions takes time

$$\tilde{O}\left(\sum_{1\leq i\leq k-1}\sum_{i\leq j\leq k}\left(c_i m + c_i c_j \frac{n^2}{c_{j+1}^2}\right)\right) = \tilde{O}\left(\sum_{1\leq i\leq k-1}\sum_{i\leq j\leq k}\left(c_1 m + \frac{c_1 c_j n^2}{c_{j+1} c_k}\right)\right)$$

$$= \hat{O}\left(\sum_{1\leq i\leq k-1}\sum_{i\leq j\leq k}\left(c_1 m + \frac{c_1 n^2}{c_k}\right)\right) = \hat{O}(k^2 c_1 m + k^2 c_1 n^2/c_k) = \hat{O}(cm + cn^2/b).$$

Maintaining Links Between Centers. For each pair of an i-center x and a j-center y there are at most $\tilde{O}(c_{l+1})$ $(l+1)$-centers that can possibly link x to y. Each such $(l+1)$-center is added to and removed from the list of $(l+1)$-centers linking x to y at most once. Thus, the total time needed for maintaining all these links is $\tilde{O}(\sum_{1\leq i\leq k-1}\sum_{1\leq j\leq i} c_i c_j c_{i+1}) = \tilde{O}(k^2 c_1^3) = \tilde{O}(c^3)$.

Maintaining Transitive Closure in Center Graph. The center graph has $\tilde{O}(c_1)$ nodes and thus $\tilde{O}(c_1^2)$ edges. During the algorithm edges are only deleted from the center graph and never inserted. Thus we can use known $O(mn)$-time decremental algorithms for maintaining the transitive closure [RZ08, Lac13] in the center graph in time $\tilde{O}(c_1^3) = \tilde{O}(c^3)$.

Total Running Time. Since the term cn^2/b is dominated by the term n^3/b, we obtain a total running time of $\hat{O}\left(bmn/c + n^3/b + cm + c^3\right)$. By setting $b = n^{5/3}/m^{2/3}$ and $c = n^{4/3}/m^{1/3}$ the running time is $\hat{O}(m^{2/3}n^{4/3} + n^4/m)$ and by setting $b = n^{9/7}/m^{3/7}$ and $c = m^{1/7}n^{4/7}$ the running time is $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7})$.

7.2.4   Decremental Single-Source Reachability

The algorithm above works for a set of randomly chosen centers. Note that the algorithm stays correct if we add any number of nodes to C_1, thus increasing the number of 1-centers for which the algorithm maintains pairwise reachability. If the number of additional centers does not exceed the number of randomly chosen centers, then the same running time bounds still apply. Thus, from the analysis above, we obtain the following result.

Theorem 7.9. *Let $S \subseteq V$ be a set of nodes. If $|S| \leq n^{4/3}/m^{1/3}$, then there is a decremental algorithm for maintaining pairwise reachability between all nodes in S with constant query time and a total update time of $\hat{O}(m^{2/3}n^{4/3} + n^4/m)$ that is correct with high probability against an oblivious adversary. If $|S| \leq m^{1/7}n^{4/7}$, then there is a decremental algorithm for maintaining pairwise reachability between all nodes in S with constant query time and total update time $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7})$ that is correct with high probability against an oblivious adversary.*

Using the reduction of Theorem 4.1 this also gives us a single-source reachability algorithm.

Corollary 7.10. *There are decremental SSR and SCC algorithms with constant query time and total update time*

$$\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$$

that are correct with high probability against an oblivious adversary. The total update time of the SCC algorithm is only in expectation.

Corollary 7.11. *There is a decremental SSR algorithm with constant query time and total update time*

$$\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$$

that is correct with high probability against an oblivious adversary.

Proof. First, set $c_1 = n^{4/3}/m^{1/3}$ and observe that by Theorem 7.9 we have an algorithm that can maintain reachability from a source to c_1 sinks with a total update time of $\hat{O}(m^{2/3}n^{4/3} + n^4/m)$. By Theorem 4.1 this implies a decremental single-source reachability algorithm with a total update time of $\hat{O}(m^{2/3}n^{4/3}+n^4/m+mn/c_1)$. The same argument gives a decremental single-source reachability algorithm with a total update time of $\hat{O}(m^{3/7}n^{12/7} + m^{8/7}n^{4/7} + mn/c_2)$ where $c_2 = m^{1/7}n^{4/7}$.

If $m \geq n^{8/5}$ we have $n^4/m \leq m^{2/3}n^{4/3}$ and $mn/c_1 = m^{4/3}/n^{1/3} \leq m^{2/3}n^{4/3}$. If $m \leq n^{8/5}$ we have $m^{8/7}n^{4/7} \leq m^{3/7}n^{12/7}$ and $mn/c_2 = m^{6/7}/n^{3/7} \leq m^{3/7}n^{12/7}$. Thus, by running the first algorithm if $m \geq n^{8/5}$ and the second one if $m < n^{8/5}$ we obtain a total update time of $\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$ (Note that $m^{3/7}n^{12/7} \leq m^{2/3}n^{4/3}$ if and only if $m \geq n^{8/5}$). □

Furthermore, the reduction of Theorem 4.2 gives a decremental algorithm for maintaining strongly connected components.

Corollary 7.12. *There is a decremental SCC algorithm with constant query time and expected total update time*

$$\hat{O}(m^{2/3}n^{4/3} + m^{3/7}n^{12/7})$$

that is correct with high probability against an oblivious adversary.

8 Conclusion

In this paper we have presented decremental algorithms for maintaining approximate single-source shortest paths (and thus also single-source reachability) with constant query time and a total update time of $o(mn)$. This result implies that the thirty-year-old algorithm with $O(mn)$ total update time is not the best possible. Since it is hard to believe that the running time obtained in this paper is tight, it is an important open problem to search for faster (and perhaps simpler) algorithms. Note that decremental approximate SSSP in *undirected* graphs can be solved with almost linear total update time [HKN14b]. Thus, it is interesting to see if the same total update time can be obtained for the directed case. Getting this result for *s-t* reachability will already be a major breakthrough.

Given recent progress in lower bounds for dynamic graph problems [Pat10, AVW14, HKN+15], it might also be possible that an almost linear update time is not possible for decremental approximate SSSP and SSR in directed graphs. However, it might be challenging to prove this as all known constructions give the same lower bounds for both the incremental and the decremental version of a problem. As incremental SSR has a total update time of $O(m)$ we cannot hope for lower bounds for decremental SSR using these known techniques. This only leaves the possibility of finding a lower bound for decremental approximate SSR or of developing stronger techniques. It also motivates studying the incremental approximate SSSP problem which is intuitively easier than its decremental counterpart, but no as easy as incremental SSR.

Finally, we ask whether it is possible to remove the assumption that the adversary is oblivious, i.e., to allow an *adaptive adversary* that may choose each new update or query based on the algorithm's previous answers. This would immediately be guaranteed by a deterministic algorithm.

References

[ACT14]   Ittai Abraham, Shiri Chechik, and Kunal Talwar. "Fully Dynamic All-Pairs Shortest Paths: Breaking the $O(n)$ Barrier". In: *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. 2014, pp. 1–16 (cit. on p. 3).

[AVW14]   Amir Abboud and Virginia Vassilevska Williams. "Popular conjectures imply strong lower bounds for dynamic problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 434–443 (cit. on pp. 6, 51).

[BDBK+94]   Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. "On the Power of Randomization in On-Line Algorithms". In: *Algorithmica* 11.1 (1994). Announced at STOC'90, pp. 2–14 (cit. on p. 7).

[BHS03]   Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Maintaining All-Pairs Approximate Shortest Paths Under Deletion of Edges". In: *Symposium on Discrete Algorithms (SODA)*. 2003, pp. 394–403 (cit. on p. 3).

[BHS07]   Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths". In: *Journal of Algorithms* 62.2 (2007). Announced at STOC'02, pp. 74–92 (cit. on p. 3).

[BR11]   Aaron Bernstein and Liam Roditty. "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions". In: *Symposium on Discrete Algorithms (SODA)*. 2011, pp. 1355–1365 (cit. on p. 3).

[Ber09]   Aaron Bernstein. "Fully Dynamic $(2 + \epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 693–702 (cit. on pp. 4, 10, 33).

[Ber13]   Aaron Bernstein. "Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 725–734 (cit. on pp. 3, 4, 10, 16, 17, 33).

[Coh98]   Edith Cohen. "Fast Algorithms for Constructing t-Spanners and Paths with Stretch t". In: *SIAM Journal on Computing* 28.1 (1998). Announced at FOCS'93, pp. 210–236 (cit. on p. 4).

[Din06]   Yefim Dinitz. "Dinitz' Algorithm: The Original Version and Even's Version". In: *Essays in Memory of Shimon Even*. 2006, pp. 218–240 (cit. on p. 3).

[ES81]   Shimon Even and Yossi Shiloach. "An On-Line Edge-Deletion Problem". In: *Journal of the ACM* 28.1 (1981), pp. 1–4 (cit. on pp. 1, 3, 10).

[HK95]   Monika Henzinger and Valerie King. "Fully Dynamic Biconnectivity and Transitive Closure". In: *Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 664–672 (cit. on pp. 3, 10).

[HK99]   Monika Rauch Henzinger and Valerie King. "Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time per Operation". In: *Journal of the ACM* 46.4 (1999). Announced at STOC'95, pp. 502–516 (cit. on p. 3).

[HKN13]   Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 538–547 (cit. on p. 3).

[HKN14a]   Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Dynamic Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072 (cit. on p. 3).

[HKN14b]   Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155 (cit. on pp. 3, 51).

[HKN14c]   Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 674–683 (cit. on pp. 1, 6).

[HKN15]   Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Improved Algorithms for Decremental Single-Source Reachability on Directed Graphs". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2015, pp. 725–736 (cit. on p. 1).

[HKN+15]   Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. "Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture". In: *Symposium on Theory of Computing (STOC)*. 2015, pp. 21–30 (cit. on pp. 4, 6, 51).

[HLT01]   Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *Journal of the ACM* 48.4 (2001). Announced at STOC'98, pp. 723–760 (cit. on p. 3).

[Ita86]   Giuseppe F. Italiano. "Amortized Efficiency of a Path Retrieval Data Structure". In: *Theoretical Computer Science* 48.3 (1986), pp. 273–281 (cit. on p. 3).

[Ita88]   Giuseppe F. Italiano. "Finding Paths and Deleting Edges in Directed Acyclic Graphs". In: *Information Processing Letters* 28.1 (1988), pp. 5–11 (cit. on p. 3).

[KKM13]   Bruce M. Kapron, Valerie King, and Ben Mountjoy. "Dynamic graph connectivity in polylogarithmic worst case time". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1131–1142 (cit. on p. 3).

[Kin08]   Valerie King. "Fully Dynamic Transitive Closure". In: *Encyclopedia of Algorithms*. 2008 (cit. on p. 4).

[Kin99]   Valerie King. "Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 1999, pp. 81–91 (cit. on pp. 3, 10).

[Lac13]   Jakub Łącki. "Improved Deterministic Algorithms for Decremental Reachability and Strongly Connected Components". In: *ACM Transactions on Algorithms* 9.3 (2013). Announced at SODA'11, p. 27 (cit. on pp. 4, 49).

[Mad10]   Aleksander Mądry. "Faster Approximation Schemes for Fractional Multicommodity Flow Problems via Dynamic Graph Algorithms". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 121–130 (cit. on pp. 4, 10, 33).

[Pat10]   Mihai Patrascu. "Towards Polynomial Lower Bounds for Dynamic Problems". In: *Symposium on Theory of Computing (STOC)*. 2010, pp. 603–610 (cit. on p. 51).

[RT87]   Prabhakar Raghavan and Clark D. Thompson. "Randomized rounding: a technique for provably good algorithms and algorithmic proofs". In: *Combinatorica* 7.4 (1987). Announced at STOC'85, pp. 365–374 (cit. on p. 4).

[RZ08]   Liam Roditty and Uri Zwick. "Improved Dynamic Reachability Algorithms for Directed Graphs". In: *SIAM Journal on Computing* 37.5 (2008). Announced at FOCS'02, pp. 1455–1471 (cit. on pp. 1, 3, 4, 17, 49).

[RZ11]   Liam Roditty and Uri Zwick. "On Dynamic Shortest Paths Problems". In: *Algorithmica* 61.2 (2011). Announced at ESA'04, pp. 389–401 (cit. on pp. 3, 6).

[Rod13]   Liam Roditty. "Decremental maintenance of strongly connected components". In: *Symposium on Discrete Algorithms (SODA)*. 2013, pp. 1143–1150 (cit. on p. 4).

[Tar72]   Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972). Announced at SWAT'71 (FOCS), pp. 146–160 (cit. on pp. 19, 20).

[Tho00]   Mikkel Thorup. "Near-optimal fully-dynamic graph connectivity". In: *Symposium on Theory of Computing (STOC)*. 2000, pp. 343–350 (cit. on p. 3).

[UY91]   Jeffrey D. Ullman and Mihalis Yannakakis. "High-Probability Parallel Transitive-Closure Algorithms". In: *SIAM Journal on Computing* 20.1 (1991). Announced at SPAA'90, pp. 100–125 (cit. on p. 10).

[VWW10]   Virginia Vassilevska Williams and Ryan Williams. "Subcubic Equivalences between Path, Matrix and Triangle Problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 645–654 (cit. on p. 6).

[Zwi02]   Uri Zwick. "All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication". In: *Journal of the ACM* 49.3 (2002). Announced at FOCS'98, pp. 289–317 (cit. on p. 4).