# A Subquadratic-Time Algorithm for Decremental Single-Source Shortest Paths*

Monika Henzinger† Sebastian Krinninger† Danupon Nanongkai‡

**Abstract**

We study dynamic $(1+\epsilon)$-approximation algorithms for the single-source shortest paths problem in an unweighted undirected $n$-node $m$-edge graph under edge deletions. The fastest algorithm for this problem is an algorithm with $O(n^{2+o(1)})$ total update time and constant query time by Bernstein and Roditty (SODA 2011). In this paper, we improve the total update time to $O(n^{1.8+o(1)} + m^{1+o(1)})$ while keeping the query time constant. This running time is essentially tight when $m = \Omega(n^{1.8})$ since we need $\Omega(m)$ time even in the static setting. For smaller values of $m$, the running time of our algorithm is *subquadratic*, and is the first that breaks through the quadratic time barrier.

In obtaining this result, we develop a fast algorithm for what we call *center cover data structure*. We also make non-trivial extensions to our previous techniques called *lazy-update and monotone Even-Shiloach trees* (ICALP 2013 and FOCS 2013). As by-products of our new techniques, we obtain two new results for the decremental all-pairs shortest-paths problem. Our first result is the *first* approximation algorithm whose total update time is faster than $\widetilde{O}(mn)$ for *all* values of $m$. Our second result is a new trade-off between the total update time and the additive approximation guarantee.

## 1 Introduction

Dynamic graph algorithms are data structures that support update as well as query operations on a graph. They are usually classified according to the types of updates allowed: *decremental* if only deletions are allowed, *incremental* if only insertions are allowed, and *fully dynamic* if both types of updates are allowed.

**The Problem.** We consider the *decremental single-source shortest paths (SSSP)* problem where we wish to maintain the distances between every node and a given *source node* $s$ in an undirected unweighted graph under a sequence of the following delete and distance query operations:

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

We use the term *decremental all-pairs shortest paths (APSP)* to refer to the more general case where the distance query is of the form DISTANCE$(x, y)$, and we have to return $d_G(x, y)$, the distance between nodes $x$ and $y$. The efficiency is judged by two parameters: *query time* denoting the time needed to answer *each* distance query, and *total update time* denoting the time needed to process *all* edge deletions. The running time will be in terms of $n$, the number of nodes in the graph, and $m$, the number of edges *before* any deletion.

We use $\widetilde{O}$-notation to hide an $O(\text{poly}\log n)$ term. When it is clear from the context, we say "time" instead of "total update time", and, unless stated otherwise, the query time is $O(1)$. The goal is to *optimize the total update time* while keeping the query time and *approximation guarantees* small. We say that an algorithm provides an $(\alpha, \beta)$-*approximation* if the distance query on nodes $x$ and $y$ on the current graph $G$ returns $\hat{d}(x, y)$ such that $d_G(x, y) \leq \hat{d}(x, y) \leq \alpha d_G(x, y) + \beta$. We call $\alpha$ and $\beta$ *multiplicative* and *additive errors*, respectively. When $\beta = 0$, we say $\alpha$-*approximation* instead of $(\alpha, 0)$-approximation.

**Previous Results.** The best exact algorithm has $O(mn)$ total update time and goes back three decades to Even and Shiloach [14]. King [19] later generalized this result to directed weighted graphs. Roditty and Zwick [21] showed that this total update time is optimal for exact distances, unless there is a major breakthrough for Boolean matrix multiplication and many other long-standing problems [27]. It is thus natural to shift the focus to *approximation algorithms*. The only approximation algorithm prior to our work is due to Bernstein and Roditty [8]; it has a $(1+\epsilon)$ approximation guarantee and an expected total update time of $O(n^{2+O(1/\sqrt{\log n})})$. This time bound is only slightly larger than quadratic time and beats the $O(mn)$ time unless the input graph is very sparse. For more detail, see Section 2.

*A full version containing all omitted proofs is available at http://eprints.cs.univie.ac.at/3785.

†University of Vienna, Faculty of Computer Science, Austria. Supported by the Austrian Science Fund (FWF): P23499-N23, the Vienna Science and Technology Fund (WWTF) grant ICT10-002, and the University of Vienna (IK I049-N). The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532.

‡Division of Mathematical Sciences, Nanyang Technological University (NTU), Singapore 637371. Supported in part by the following research grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 2 grant MOE2010-T2-2-082, and Singapore MOE AcRF Tier 1 grant MOE2012-T1-001-094.

**1.1 Our Results** The algorithm of Bernstein and Roditty naturally leads to the question whether there is a *subquadratic-time* algorithm, i.e., an algorithm with $O(n^{2-\delta})$ running time for some constant $\delta > 0$. In this paper, we show that, as long as $m$ is subquadratic, decremental SSSP can be solved in subquadratic time:

THEOREM 1.1. (MAIN RESULT) *For any constant $0 < \epsilon < 1$, there is a $(1+\epsilon)$-approximation algorithm for decremental SSSP with worst-case constant query time and expected total update time of $O(n^{1.8+O(1/\sqrt{\log n})} + m^{1+O(1/\sqrt{\log n})})$.*[1]

Theorem 1.1 also implies that when $m = \Omega(n^{1.8})$ our total update time is *almost linear* ($O(m^{1+o(1)})$ time) and thus almost tight. The algorithm is, as all algorithms in this paper, correct with high probability. Also note that the expected total update time guarantee can be turned into a high probability guarantee by simply running $\Theta(\log n)$ copies of the algorithm in parallel[2]. Additionally, as by-products of the techniques developed in this paper, we obtain two new results for decremental APSP.

THEOREM 1.2. ($(3+\epsilon)$-APPROXIMATION FOR APSP) *For any constant $0 < \epsilon < 1$, there is a $(3+\epsilon)$-approximation algorithm for decremental APSP with $O(\log\log n)$ worst-case query time and expected total update time of $\widetilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$.*

THEOREM 1.3. (TRADE-OFF FOR APSP) *For any constant $0 < \epsilon < 1$ and integer $2 \le k \le \log n$, there is a $(1+\epsilon, 2(1+2/\epsilon)^{k-2})$-approximation algorithm for decremental APSP with constant worst-case query time and expected total update time of $\widetilde{O}(n^{2+1/k}(37/\epsilon)^{k-1})$.*

Prior to these results, Roditty and Zwick [22] presented a $(1+\epsilon)$-approximation algorithm with $\widetilde{O}(mn)$ total update time. Bernstein and Roditty [8] presented algorithms with $\widetilde{O}(n^{2+1/k+O(1/\sqrt{\log n})})$ total update time and $(2k-1+\epsilon, 0)$ approximation guarantee. We [16] recently showed an algorithm with $\widetilde{O}(n^{2.5})$ total update time guaranteeing both $(1+\epsilon, 2)$ and $(2+\epsilon, 0)$ approximation factors. Both Bernstein and Roditty's and our algorithms improve the $\widetilde{O}(mn)$ time of Roditty and Zwick when the input graph is dense. Our result in Theorem 1.2 is the *first* approximation algorithm that is faster than $\widetilde{O}(mn)$ for *all* values of $m$. Our result in Theorem 1.3 complements the result of Bernstein and Roditty and generalizes our previous result in [16].

**1.2 Techniques** In obtaining the subquadratic-time algorithm we develop new techniques, as well as extend some old ones, which we hope will also be useful in other situations.

**Review of Even-Shiloach Tree (ES-Tree).** An *Even-Shiloach tree* (ES-tree) has two parameters: a root node $r$ and the range (or depth) $\tau$. It maintains a breadth-first search tree rooted at $r$ and the distances between $r$ and all other nodes in the dynamic graph, up to distance $\tau$ (if the distance is greater than $\tau$, it will be set to $\infty$). This is done by repeatedly updating every node's distance to $r$ every time it increases (in the decremental setting) or decreases (in the incremental setting), which requires time proportional to its degree. This gives a query time of $O(1)$ and, since every node's distance changes at most $\tau$ times, the total update time is $O(m\tau)$; to be more precise, it is $O(m_r\tau)$ where $m_r$ is the number of edges within the distance of $\tau$ from $r$ (this fact will play an important role in our analysis).

**Faster Algorithm for Center Cover Data Structure.** In the center cover data structure, we are given a parameter $h$ and a constant $\alpha$. We would like to maintain $\widetilde{O}(h)$ nodes, called *centers*, and ES-trees of depth $O(n/h)$ from these centers. The goal is that every node $v$ is of distance at most $n/h$ from some center $c$ (we say that $v$ is "covered" by $c$) whose ES-tree is of depth at least $\alpha n/h$, where $\alpha$ is usually constant or near-constant. The main property of this data structure is that every node that is of distance at most $(\alpha-1)n/h$ to $v$ is contained in the ES-tree of $c$. This data structure has played an important role in solving decremental APSP (e.g. [22, 16]). In this paper, we present a new algorithm which has the same aforementioned property and is faster than the previous ones (with some caveats) when $h$ is not too small, and show how to use this new algorithm to improve over previous decremental SSSP algorithms.

Previously this data structure was implemented by randomly picking $\widetilde{O}(h)$ nodes as centers. Following an argument of Ullman and Yannakakis [26], it can be shown that every node will have a center of distance at most $n/h$ from it with high probability, assuming that edge deletions are done independently from the random choice of centers (i.e., assuming an *oblivious* adversarial model). From each of these $h$ centers an ES-tree of

[1] To be precise, the $O(1/\sqrt{\log n})$ term in Theorem 1.1 is $\sqrt{\log(12/\epsilon)/\log n}$.

[2] Whenever we get an update (delete) operation, we send this to a copy $A$ only after $A$ has finished processing all previous update operations. At any point in time, we say that we have finished updating if one of the copies has finished updating. We then are ready to take a new query. When we get a distance query, we query a copy that has finished updating. Observe that the query time remains the same and the total update time is at most the minimum total update time among all copies.

depth $\alpha \cdot n/h$ is maintained in time $O(\alpha mn/h)$, leading to a total update time (over $\widetilde{O}(h)$ centers) of $\widetilde{O}(\alpha mn)$.

To improve this update time, we introduce the notion of *rich and poor centers*. We call the above randomly selected centers *poor centers*. For each poor center $r$, we maintain an ES-tree from it only when there are at most $\rho$ edges of distance at most $\alpha n/h$ from $r$, for some parameter $\rho$ to be fixed later. Thus, the total update time of maintaining such an ES-tree is $O(\rho n/h)$. We call a poor center for which we maintain an ES-tree *active*. Additionally, we sample $\widetilde{O}(m/\rho)$ edges and call their end-nodes *rich centers*. We maintain ES-trees of depth $2\alpha n/h$ from all these rich centers. By a standard argument, we can prove the following claim.

CLAIM 1.4. *Every node is of distance at most $n/h$ from some active poor center (maintaining an ES-tree of depth $\alpha n/h$) or distance at most $(1+\alpha)n/h$ from some rich center (maintaining an ES-tree of depth $2\alpha n/h$).*

Observe that we can use Claim 1.4 to guarantee the main property that, for every node $v$, every node in distance at most $(\alpha - 1)n/h$ to $v$ is contained in the ES-tree of its nearest active poor center or rich center. When $\alpha$ is a constant, we need a total update time of $\widetilde{O}(h\rho n/h) = \widetilde{O}(n\rho)$ for maintaining ES-trees from at most $h$ active poor centers and $\widetilde{O}(m^2/\rho \cdot n/h)$ for maintaining ES-trees from $m/\rho$ rich centers. By setting $\rho = m/\sqrt{h}$, we get a total update time of $\widetilde{O}(n\rho + m^2 n/(\rho h)) = \widetilde{O}(mn/\sqrt{h})$. This gives an improvement over the previous $\widetilde{O}(mn)$ time when $h$ is $\omega(1)$.

**Faster Decremental SSSP Algorithm on Sparse Graphs (Details in Section 3).** We now show how to use the above new algorithm to obtain an improved decremental SSSP algorithm on sparse graphs. We note that this is already a non-trivial task since, in the case $m = O(n)$, *no* previous dynamic algorithm can beat the naive solution where we recompute the breadth-first search (BFS) tree *from scratch* after every deletion (i.e. no previous algorithm achieves $o(n^2)$ total update time). Our general approach to attack this special case is inspired by our recent technique called *lazy-update Even-Shiloach tree* introduced in [17]. In [17], we used this technique to obtain a $(1+\epsilon)$-approximation $O(n^{1.8})$-time algorithm for *incremental* SSSP (allowing only edge insertions) when $m = O(n)$[3]. It is difficult to extend this approach to the decremental setting since it involves estimating each node's distance from the source node after each edge insertion and deletion, which is easy to do in the incremental setting but hard in the decremental one. In this paper, we modify (and arguably simplify) this approach so that we can avoid this problem and make use of our new center cover data structure, as follows.

Recall that the total update time of the ES-tree crucially relies on the number of time the nodes' distances to the root change. It is thus natural to try to reduce it by making the distance update *lazy* in the sense that we will change a node's distance only when this change is larger than some parameter $\delta$. Clearly, this modification reduces the number of changes of each node's distance from $O(n)$ to $O(n/\delta)$, and thus can potentially reduce the total update time to $O(mn/\delta)$. However, it also leads to a distance error. In [17], we showed that by regularly recomputing the BFS tree from scratch, we can keep this distance error small. One crucial subroutine of this approach is checking, after an edge deletion, whether a node's distance (from the source) increases by at least $\delta$. In this paper, we provide the new insight that *this task can be done using the center cover data structure*, as follows. Let $G$ and $G'$ be graphs before and after deleting edge $(u, v)$, respectively. Assume that we maintain a center cover data structure with parameters $h = n/\delta$ and $\alpha = 2$. By Claim 1.4, there is a center $c$ such that $d_{G'}(c, v) \leq 3\delta$, at which we maintain an ES-tree of depth $\Theta(\delta)$ which is at least $d_{G'}(c, v) + \delta$; let $T'$ be such tree in $G'$. This implies that $T'$ contains all nodes of distance (in $G'$) at most $\delta$ from $v$. Also recall that every node in $T'$ is of distance at most $\Theta(\delta)$ from $v$ (because of the depth of $T'$). These two facts intuitively imply that we can use $T'$ to determine whether $v$'s distance increases by $\Omega(\delta)$:

CLAIM 1.5. *If $\min_{x \in T'} d_G(s, x) < d_G(s, v)$, then $d_{G'}(s, v) \leq d_G(s, v) + O(\delta)$; otherwise, $d_{G'}(s, v) \geq d_G(s, v) + \delta$.*

*Proof.* [Sketch] Consider when $\min_{x \in T'} d_G(s, x) < d_G(s, v)$. Let $x = \arg\min_{x \in T'} d_G(s, x)$. (i) Note that $d_{G'}(s, x) < d_G(s, v)$. This is because we know that $d_{G'}(s, x) = d_G(s, x)$ since deleting $(u, v)$ does *not* affect the distance of any node $y$ such that $d_G(s, y) < d_G(s, v)$. (ii) Also note that $d_{G'}(v, x) = O(\delta)$ because $v$ and $x$ are in the same tree $T'$ of depth $O(\delta)$. So, we have $d_{G'}(s, v) \leq d_{G'}(s, x) + d_{G'}(x, v) \leq d_G(s, v) + O(\delta)$ (the last inequality is by (i) and (ii)).

Now consider when $\min_{x \in T'} d_G(s, x) \geq d_G(s, v)$. Assume for a contradiction that $d_{G'}(s, v) \leq d_G(s, v) + \delta - 1$. Then, there must be a path of length at most $\delta$ in $G'$ between $v$ and some node $y$ such that $d_{G'}(s, y) < d_G(s, v)$. Node $y$ must be in $T'$ since $T'$ contains all nodes of distance at most $\delta$ from $v$. This means that $\min_{x \in T'} d_G(s, x) \leq d_G(s, y) \leq$

[3] More generally, the total update time is $O(mn^{2/5}q^{2/5})$ where $m$ is the number of edges in the final graph (after all insertions) and $q$ is the number of edge insertions.

$d_{G'}(s,y) < d_G(s,v)$. This contradicts the assumption that $\min_{x \in T'} d_G(s,x) \geq d_G(s,v)$. □

Observe that the condition $\min_{x \in T'} d_G(s,x) < d_G(s,v)$ in Claim 1.5 does not require the knowledge of the distance in $G'$; in other words, we can estimate the distance change of $v$ *after* the edge deletion by examining the distance of every node *before* the deletion and updating the center cover data structure. This is the key insight that allows us to check whether $v$'s distance increases by $\Omega(\delta)$. We can thus obtain the same result as we have obtained in the case of the incremental model [17], i.e., $\widetilde{O}(n^{1.8})$ total update time when $m = \widetilde{O}(n)$.

**Monotone ES-Trees on Sparse Emulator (Details in Section 4).** To prove Theorem 1.1 when the input graph is not sparse, we extend the *monotone ES-tree* technique we developed in [16]. This technique is designed to maintain an ES-tree on an *emulator*, a sparse dynamic graph that preserves the distance between all nodes in the dynamic input graph. It is generally hard to bound the error incurred by running the monotone ES-tree on an emulator. In [16], we showed that the error can be bounded if we run the monotone ES-tree on a certain type of emulator called *locally-persevering*, and showed that we can dynamically maintain such an emulator having $\widetilde{O}(n^{1.5})$ edges. We will be done if we can show that a similar result can be obtained for some *very sparse* emulator (having $\widetilde{O}(n)$ edges). The difficulty is, however, that we are *not* aware of any locally-persevering emulator having $o(n^{1.5})$ edges. We get around this by combining our previous technique in [16] and the proof of Thorup and Zwick [25] to show that our monotone ES-tree works well on their emulator [24, 25]. In particular, Thorup and Zwick [25] showed that, for any $k \geq 2$, their emulator has $\widetilde{O}(n^{1+1/k})$ edges and approximates the distances between all pairs with $(1+\epsilon)$-multiplicative and $(O(1/\epsilon))^k$-additive error. We show that the same error guarantee can be obtained when we run the monotone ES-tree on this emulator. In principle we apply the same proof idea as Thorup and Zwick [25], except that we have to carefully argue that the error bound holds despite the edge insertions. Our new analysis also leads to new algorithms for decremental APSP as in Theorem 1.3.

## 2 Related Work

The decremental SSSP problem was the first dynamic problem studied in theoretical computer science. Besides being interesting in its own right, it also arises as a subproblem in many other dynamic algorithms and can be adapted to other settings (e.g. [5, 6, 7, 16, 17, 19, 21, 22]). The first non-trivial result is from 1981 by Even and Shiloach [14]. They showed an algorithm with $O(mn)$ total update time and $O(1)$ query time, which, as observed by King [19], works even on directed unweighted graphs. (King also observed a pseudopolynomial total update time of $O(mnW)$, where $W$ is the largest edge weight.) King and Thorup [20] later presented a technique that allows us to implement this algorithm using less space. More recently, Bernstein [6] extended Even and Shiloach's algorithm to a $(1+\epsilon)$-approximation one on directed weighted graphs. This in turn serves as an important subroutine for his recent breakthrough for the decremental all-pairs shortest paths problem on directed weighted graphs [7].

Achieving an $o(mn)$ total update time with a small query time for decremental SSSP has been a long-standing open problem but no progress has been made since 1981. Roditty and Zwick [21] provided an explanation for this by showing that achieving an $o(mn)$-time *exact* combinatorial algorithm for the incremental and decremental SSSP problems on unweighted undirected graphs is impossible unless we make a major breakthrough for Boolean matrix multiplication and many other long-standing problems. For approximation algorithms, Bernstein and Roditty [8] recently achieved a $(1+\epsilon)$-approximation algorithm with $\widetilde{O}(n^{2+O(1/\sqrt{\log n})})$ expected total update time and $O(1)$ query time, thus improving the previous $O(mn)$ total update time as long as the input graph is not too sparse. The present paper further improves the total update time further.

Much more progress has been seen for decremental APSP. Dor et al. [12] showed that unless we make a major breakthrough for Boolean matrix multiplication and many other long-standing problems, we cannot achieve a combinatorial algorithm for APSP with an approximation factor less than 2, *even in the static setting* (see [16, Fact 2] for more detail). The current fastest exact algorithms for both incremental and decremental setting are due to Ausiello et al. [2, 3], Baswana et al. [5], and Demetrescu and Italiano [11], which have $\widetilde{O}(n^3)$ expected total update time and work even on directed graphs. The current fastest $(1+\epsilon)$-approximation algorithm on unweighted undirected graphs was presented by Roditty and Zwick [22]. Its expected total update time is $\widetilde{O}(mn)$ (improving from Baswana et al. [5, 4]), and is likely to be tight unless we make a major breakthrough as explained above. As noted after Theorem 1.3, results in [8, 16] and this paper build on and extend Roditty and Zwick's algorithms. One of their algorithms was also recently derandomized in [16]. Also very recently, a $(1+\epsilon)$-approximation $\widetilde{O}(mn \log W)$-time algorithm for the directed weighted case was presented in [7], where $W$ is the ratio of the largest edge weight ever seen in the graph to the smallest such weight. We

refer the readers to, e.g., [1, 12, 13] for previous static algorithms and [9, 10, 11, 15, 18, 19, 23] for previous fully dynamic algorithms for APSP.

## 3 New Decremental SSSP Algorithm for Sparse Graphs

We now give an algorithm that maintains approximate shortest paths from a source node $s$ up to depth $\tau$ under edge deletions. We first present an algorithm that computes approximate distances with *additive* error. Then we will turn the additive error into a multiplicative error of $1+\epsilon$. We denote the current distance between the nodes $x$ and $y$ in the graph $G$ by $d_G(x,y)$. When the graph we refer to is clear from the context, we omit the subscript in the distance function. The goal of the $(1+\epsilon)$-approximate algorithm is to maintain a distance estimate $\hat{d}(v,s)$ for every node $v$ such that $\hat{d}(v,s) \geq d_G(v,s)$ and if $d_G(v,s) \leq \tau$, then also $\hat{d}(v,s) \leq (1+\epsilon)d_G(v,s)$. A decremental SSSP algorithm for *full depth* sets $\tau = n$.

### 3.1 Additive Approximation Algorithm

We first explain how the additive approximation algorithm works (see Algorithm 1 for the pseudocode). We assume that the graph from which we delete edges always stays connected (see Section 3.3 how to remove this assumption). The algorithm has the parameters $\kappa \leq m$, $\delta \leq \tau$, and $\rho \leq m$. It will use a special set of nodes called *centers* that "cover" other nodes. We say that a node $v$ is *$t$-covered* by a center $c$ if $d(c,v) \leq t$. We initialize the algorithm by sampling two sets of random nodes. First, we sample each edge independently with probability $\frac{a \log m}{\rho}$ for a large enough constant $a$. The nodes incident to the sampled edges will be used as *rich centers*. Note that the expected number of rich centers is $O(\frac{m \log m}{\rho})$. Second, we sample each node independently with probability $\frac{a \log n}{\delta}$. The sampled nodes will be used as *poor centers*. The expected number of poor centers is $O(\frac{n \log n}{\delta})$. The algorithm will make a poor center active if it is not $3\delta$-covered by any rich center. By a standard "hitting set" argument, we can argue that every node is in small distance to a center.

LEMMA 3.1. *Every node is either $4\delta$-covered by a rich center or $\delta$-covered by an active poor center with high probability (whp).*

*Proof.* Suppose that $v$ is not $\delta$-covered by a rich center. (If it is $\delta$-covered by a rich center, it is also $4\delta$-covered.) Let $c$ be the closest rich center to $v$ . Since $c$ does not $\delta$-cover $v$, we have $d(v,c) > \delta$. Therefore the shortest path from $v$ to $c$ contains at least $\delta$ nodes (Note that the shortest path exists because we assume that the graph stays connected). By our sampling process one of the first $\delta$ nodes on this path is a poor center whp. Thus, $v$ is $\delta$-covered by a poor center $c$. If $c$ is inactive, then it is $3\delta$-covered by a rich center $c'$. In that case $v$ is $4\delta$-covered by $c'$. □

For every rich center we maintain an Even-Shiloach tree (ES-tree) up to depth $6\delta$ and for every active poor center we maintain an ES-tree up to depth $3\delta$. Note that a poor center $c$ might not be active from the beginning. We will start maintaining the ES-tree of $c$ only when $c$ becomes active. It is clear that as soon as a poor center becomes active it will stay active because distances never decrease in the decremental setting. Furthermore, for every center we will keep a priority queue whose elements are the nodes of the graph and the key of each node $v$ is $L(v)$, a certain estimate of the distance between $v$ and $s$ to be defined below. Since we maintain these ES-trees, we can, for every node $v$, easily maintain a list of rich centers and poor centers by which it is $4\delta$-covered or $\delta$-covered, respectively. After the ES-tree of a center $c$ is initialized, we add $c$ to the list of centers of every node $v$ such that $v$ is at depth at most $4\delta$ or $\delta$ in the ES-tree, respectively. When the depth of a node $v$ in the ES-tree of a center $c$ increases to more than $4\delta$ or $\delta$, respectively, then we delete $c$ from the list of centers of $v$. Now every node can always find a rich center by which it is $4\delta$-covered or a poor center by which it is $\delta$-covered by retrieving the first element from this list.

The main algorithm works in phases. At the beginning of each phase, we first compute a BFS tree $T$ rooted at $s$ up to depth $\tau$. This computation also determines the current distance from $s$ of every node. For every node $v$, we define the *level* $l(v)$ of $v$ as the distance between $v$ and $s$ at the *beginning* of the current phase. Furthermore we define the *rounded level* $L(v)$ of $v$ as the value of $l(v)$ rounded down to the nearest multiple of $\delta$, i.e., $L(v) = \lfloor l(v)/\delta \rfloor \cdot \delta$. Note that $L(v) \leq l(v) < L(v) + \delta$ . The second step at the beginning of each phase is to update the priority queues of the centers. For every node $v$ we check whether $L(v)$ has increased since the last phase. If yes, then for every center $c$ we set the key of $v$ in the priority queue of $c$ to $L(v)$.

We now explain how the algorithm proceeds after each edge deletion. We first check whether the number of edges deleted since the beginning of the current phase has reached $\kappa$. If yes, we start a new phase. This ensures that there are at most $\kappa$ edge deletions in each phase. Otherwise, we start the following reconnection procedure. Let $U$ be the set of nodes for which the edge to their parent in the tree $T$ has been deleted since the beginning of the current phase. We now do the following for every node $u \in U$ to find a new parent to connect to

**Algorithm 1:** Additive approximation algorithm

```
// After deleting an edge (x,y), the deletion procedure has to be executed for both (x,y) and (y,x).
initialize()
    // First initialization and re-initialization at beginning of new phase
    Compute BFS tree T rooted at s up to depth τ
    foreach node v do set level l(v) to current distance between v and s and L(v) to ⌊l(v)/δ⌋ · δ
    foreach node v such that L(v) has increased do
        foreach center c do set key of v in priority queue of c to L(v)
    U ← ∅
delete(x, y)
    if y is parent of x in tree T then add x to U
    if there were κ updates since the beginning of the current phase then
        initialize() // start new phase
    else
        // ''reconnection procedure''
        foreach u ∈ U do
            Find rich center c that 4δ-covers u or active poor center c that δ-covers u
            if s is contained in ES-tree of c then
                Make s the parent of u in T
            else
                Find node v with minimum rounded level L(v) in ES-tree of c
                if L(v) < l(u) − δ then
                    Make v the parent of u in T
                else
                    initialize() // start new phase
```

in $T$. Let $c$ be a rich center that $4\delta$-covers $u$ or an active poor center that $\delta$-covers $u$. By Lemma 3.1 such a center exists whp. Let $v$ be a node of minimum rounded level $L(v)$ in the ES-tree of $c$, which can be obtained from the priority queue of $c$.[4] If $L(v) < l(u) - \delta$, the algorithm will make $v$ the parent of $u$ in $T$ and mark this edge with the center $c$. For the correctness of the algorithm we will show that taking the edge $(u, v)$ instead of the deleted tree edge increases the additive error by at most $10\delta$. If $L(v) \geq l(u) - \delta$, we start a new phase. For the running time analysis we will argue later that in this case the algorithm makes progress: the distance between $u$ and $s$ has increased by at least $\delta$ and the sum of all distances from $s$ has increased by $\Omega(\delta^2)$. An exception is the situation that the source $s$ is contained in the ES-tree of $c$ in which we make $s$ the parent of $u$ in $T$ and mark this edge with $s$.

The algorithm answers distance and path queries as follows. A query for the approximate distance between a node $x$ and $s$ is answered by returning $l(x)$, the level computed at the beginning of the current phase.[5] An approximate shortest path of $x$ to $s$ is generated by following the path from $x$ to $s$ in $T$. Every tree edge of $T$ traversed in this process is either also contained in the tree computed at the beginning of the current phase or has been introduced in the reconnection procedure. In the first case, we simply output $(u, v)$ as the next part of the approximate shortest path and then proceed with the node $v$. In the second case, this edge is marked with a center $c$. In that case we output the shortest paths from $u$ to $c$ and from $c$ to $v$ that can be obtained from the ES-tree of $c$ and then proceed with the node $v$.

Obviously, the algorithm builds an exact SSSP tree each time a new phase starts. We can show that we introduce an additive error of at most $10\delta$ every time we connect a node $u \in U$ to a node $v$ of lower level by going over one of the centers. As there are at most $\kappa$ deletions in each phase, the additive error is at most $10\kappa\delta$ at any time.

LEMMA 3.2. *After every update processed by the algorithm the graph $T$ is a tree and, for every node $x$, $d(x, s) - 10\kappa\delta \leq l(x) \leq d(x, s)$.*

*Proof.* The second inequality is true by the definition of the level as the distance between $x$ and $s$ at the beginning of the current phase: since distances never

[4]Note that the level of $v$ refers to the distance between $v$ and $s$ in the initial BFS tree and not to the distance between $v$ and $c$ in the ES-tree of $c$.

[5]To be precise, we return, for a node $x$, the distance estimate $l(x) + 10\kappa\delta$ since we demand that the returned distance estimate does not underestimate the real distance.

decrease under edge deletions, the inequality $d(x,s) \geq l(x)$ always holds.

For the first inequality, consider the tree $T$ built by the algorithm and the (unique) path $P$ from $x$ to $s$ in $T$ that contains for every node, starting from $x$, the edge to its parent in $T$. Let $u$ and $v$ be nodes in $T$ such that $v$ is the parent of $u$. If $u \notin U$, then $v$ also was the parent of $u$ in the BFS tree computed at the beginning of the current phase and thus $l(v) < l(u)$. If $u \in U$, then either $v = s$ or $L(v) < l(u) - \delta$. If $v = s$, then $l(v) = 0$ and thus $l(v) < l(u)$ because 0 is the minimum level and is reached only by $s$. If $L(v) < l(u) - \delta$, then because of $l(v) \leq L(v) + \delta$ we also get $l(v) < l(u)$. Since every parent has smaller level than its children (and every node has a parent), we infer that $T$ is indeed a tree. This fact also implies that $l(x)$ is an upper bound on the number of edges of $P$.

Now let $(u,v)$ be an edge of the path $P$, which means that $v$ is the parent of $u$ in $T$. If $u \notin U$, then the edge $(u,v)$ is contained in $G$. As argued above, this case can happen at most $l(x)$ times. If $u \in U$, then the edge is marked with a center $c$ in $T'$ such that the following holds: (i) Either $c$ is a rich center such that $u$ is $4\delta$-covered by $c$ and $v$ is in the ES-tree of depth $6\delta$ of $c$ or (ii) $c$ is a poor center such that $u$ is $\delta$-covered by $c$ and $v$ is in the ES-tree of depth $3\delta$ of $c$. In any case we have $d(u,v) \leq d(u,c) + d(c,v) \leq 10\delta$. Note that there are at most $\kappa$ nodes in $U$. Therefore the path $P$ in the tree $T$ corresponds to a path from $x$ to $s$ in $G$ that has length at most $l(x) + 10\kappa\delta$. Since the length of this path is at least the length of a shortest path we get $d(x,s) \leq l(x) + 10\kappa\delta$. □

We finally explain how to implement the priority queues. We could simply implement them by min-heaps, but there also is a more efficient data structure since the keys $L(v)$ never decrease and only have $O(n/\delta)$ different values. For each $i$ we maintain a doubly linked list $L_i$ of elements corresponding to nodes for which $L(v) = i\delta$ and for each node we maintain a pointer to its list element. If the key of a node $v$ increases from $i\delta$ to $j\delta$, we first use the pointer to find its list element in $L_i$, remove it from $L_i$, and insert it into $L_j$. Furthermore, we maintain a pointer $p$ to the first non-empty list. Every time the list to which $p$ points becomes empty, we move $p$ "to the right" until we find the first non-empty list again. Using the pointer $p$, we can always return a node of minimum key in the priority queue.

**3.2 Running Time Analysis** We now analyze the expected running time of the algorithm and explain how to obtain a multiplicative $(1+\epsilon)$-approximation from the additive approximation. Our algorithm incurs the following (expected) costs over all deletions:

(1) Maintaining the ES-trees of the rich centers takes time $O(m\delta \cdot \frac{m \log m}{\rho})$ since there are $O(\frac{m \log m}{\rho})$ rich centers in expectation.

(2) Maintaining the priority queue of a rich center takes total time $O(n\tau/\delta)$: increasing the key $L(v)$ of a node $v$ takes constant time and each key $L(v)$ increases at most $O(\tau/\delta)$ times; similarly, the pointer to the list of minimum elements is moved at most $O(\tau/\delta)$ times. Therefore the total time needed for maintaining the priority queues of all rich centers is $O(n \cdot \frac{\tau}{\delta} \cdot \frac{m \log m}{\rho})$. (Note that this step would be less efficient if we used exact levels instead of rounded levels.)

(3) Similarly, the time needed for maintaining the priority queues of all poor centers is $O(n \cdot \frac{\tau}{\delta} \cdot \frac{n \log n}{\delta})$.

(4) The running time of the reconnection procedure after a deletion is $O(1)$ for every node in $U$. As there are at most $\kappa$ nodes in $U$ and at most $m$ deletions this takes time $O(m\kappa)$ in total.

It remains to bound the costs of maintaining the ES-trees of the poor centers and of doing the re-initialization at the beginning of each phase. The first cost can be bounded by showing that poor centers have a sparse neighborhood. If there were more than $\rho$ edges in the neighborhood of a poor center, then one of them would have been selected in the sampling step whp and thus some rich center would cover this poor center and make it inactive.

LEMMA 3.3. *For every active poor center $c$, the number of edges incident to the ES-tree of $c$ is at most $\rho$ whp.*

*Proof.* Assume by contradiction that the number of edges incident to the ES-tree of $c$ is more than $\rho$. We will argue that whp $c$ is $3\delta$-covered by a rich center which contradicts the assumption that $c$ is active. The ES-tree of $c$ has depth $3\delta$. Since there are more than $\rho$ edges incident to the ES-tree of $c$, one of them has been sampled whp. Therefore, there is some rich center $c'$ in the ES-tree of $c$. But this means that $c$ is $3\delta$-covered by $c'$ and thus $c$ would not be active, which contradicts the assumptions. □

(5) Maintaining the ES-trees of active poor centers takes time $O(\rho\delta \cdot \frac{n \log n}{\delta}) = O(n\rho \log n)$ since there are $O(\frac{n \log n}{\delta})$ poor centers in expectation.

For the second remaining cost we want to bound the number of phases of the algorithm. To do this we first show that every time the reconnection fails for some

node $u$ (i.e., $L(v) \geq l(u) - \delta$) the distance between $u$ and $s$ has increased by at least $\delta$ since the beginning of the current phase. The intuition is as follows: Consider the node $x$ at distance exactly $2\delta$ from $u$ on the shortest path from $u$ to $s$. This node is contained in the ES-tree that covers $u$. Thus $L(v) \leq L(x)$. If $L(v) \geq l(u) - \delta$, then also $L(x) \geq l(u) - \delta$, which implies that $x$ is at distance $2\delta$ from the current level of $u$ but at distance at most $\delta$ from the "old" level $l(u)$ of $u$. It follows that the level of $u$ must have increased by at least $\delta$.

LEMMA 3.4. *Let $u \in U$ be a node whose edge to its parent has been deleted since the beginning of the current phase. Let $c$ be a rich center that $4\delta$-covers $u$ or an active poor center that $\delta$-covers $u$. Let $v$ be a node of minimum rounded level $L(v)$ in the ES-tree of $c$ and assume that $s$ is not contained in the ES-tree of $c$. If $L(v) \geq l(u) - \delta$, then $d(u,s) \geq l(u) + \delta$.*

*Proof.* We give a proof by contradiction. Assume that $L(v) \geq l(u) - \delta$ and $d(u,s) < l(u) + \delta$. Consider first the case that $d(u,s) \geq 2\delta$. Let $x$ be a node on a shortest path from $u$ to $s$ in distance $2\delta$ to $u$, i.e., $d(u,s) = d(u,x) + d(x,s)$ and $d(u,x) = 2\delta$. By our assumption it follows that $d(x,s) + 2\delta = d(u,s) < l(u) + \delta$ and thus $d(x,s) < l(u) - \delta$. Since distances are non-decreasing under edge deletions we have $l(x) \leq d(x,s)$ and thus $l(x) < l(u) - \delta$.

We now argue that $x$ is in the ES-tree of $c$. If $c$ is a rich center, then $u$ is $4\delta$-covered by $c$. Since $d(u,x) = 2\delta$, we get that $d(c,x) \leq 6\delta$ and thus $x$ is contained in the ES-tree of $c$ which has depth $6\delta$. If $c$ is an active poor center, then $u$ is $\delta$-covered by $c$. Since $d(u,x) = 2\delta$, we get that $d(c,x) \leq 3\delta$ and thus $x$ is contained in the ES-tree of $c$ which has depth $3\delta$.

Since $x$ is contained in the ES-tree of $c$ we have $L(v) \leq L(x)$ because $v$ is a node with minimum rounded level $L(v)$ in the ES-tree of $c$. Furthermore, by the definition of the rounded level we have $L(x) \leq l(x)$. Therefore we get

$$L(v) \leq L(x) \leq l(x) < l(u) - \delta$$

which contradicts the assumption $L(v) \geq l(u) - \delta$.

Finally, observe that the case $d(u,s) < 2\delta$ is not possible because by the same argument as above we would get that $s$ is contained in the ES-tree of $c$, which contradicts the assumption. □

Thus, every time the reconnection fails (which leads to a new phase being started), we can be sure that there is a node $u$ such that the distance between $u$ and $s$ has increased a lot. We can now bound the number of phases by using our previous technique [17]. First of all, since we are given an unweighted undirected graph the distance from $s$ does not only increase for $u$ but also for other nodes in the "neighborhood" of $u$.

LEMMA 3.5. ([17]) *If there is a node $u$ such that $\infty > d(u,s) > l(u) + \delta$, then $\sum_{v \in V}(d(v,s) - l(v)) = \Omega(\delta^2)$.*

Second, since the distance from $s$ is bounded by $\tau$ for every node and the sum of these distances over all nodes is bounded by $n\tau$, we can bound the number of this type of phase restarts by $O(n\tau/\delta^2)$.

LEMMA 3.6. *The number of phases is $O(m/\kappa + n\tau/\delta^2)$.*

*Proof.* There are two conditions in the algorithm that cause a new phase to be started. The first condition is that $\kappa$ edges have been deleted since the beginning of the current phase. As there are at most $m$ edge deletions, this can happen $O(m/\kappa)$ times. The second condition is that the reconnection procedure fails for some node $u \in U$. By Lemma 3.4 this implies that the distance between $u$ and $s$ has increased by at least $\delta$ since the beginning of the current phase. By Lemma 3.5 this means that the sum of all distances from $s$ has increased by $\Omega(\delta^2)$. Note that there are $n$ nodes and the distance from $s$ is bounded by $\tau$. Therefore the sum of all distances from $s$ is bounded by $n\tau$. Thus, the second condition can happen $O(n\tau/\delta^2)$ times and in total there are $O(m/\kappa + n\tau/\delta^2)$ phases. □

We can now bound the cost of the re-initializations as follows:

(6) The total time needed for re-initializations at the beginning of a phase is $O(m \cdot (m/\kappa + n\tau/\delta^2))$ because computing a BFS tree takes time $O(m)$.

Having analyzed the individual components of the additive approximation algorithm, we now bound its overall running time. By appropriate choices of the parameters $\delta$, $\kappa$, and $\rho$ we can balance some of the terms in the running times listed above and can also show that some terms are dominated by others.

LEMMA 3.7. *There is an algorithm that maintains approximate shortest paths under edge deletions up to depth $\tau \leq n$ with an additive error of $\alpha \leq \tau$ in total time*

$$O\left(\frac{m^{5/3}n^{1/3}\tau^{1/3}\log n}{\alpha^{2/3}} + m^{5/6}n^{2/3}\tau^{1/6}\alpha^{1/6}\log n\right).$$

*Proof.* We simply have to give a choice of parameters that guarantees the desired running time. We choose the parameters as follows:

$$\delta = \frac{n^{1/3}\tau^{1/3}\alpha^{1/3}}{m^{1/3}} \quad \kappa = \frac{m^{1/3}\alpha^{2/3}}{10n^{1/3}\tau^{1/3}} \quad \rho = \frac{m^{5/6}\tau^{1/6}\alpha^{1/6}}{n^{1/3}}$$

It can easily be checked that $\delta \le n$, $\kappa \le m$, $\rho \le m$ and that the additive error of $10\kappa\delta$ is equal to $\alpha$.

It is also simple to verify that by our choice of parameters the running time of item (6) is $O(m^{5/3}n^{1/3}\tau^{1/3}/\alpha^{2/3})$, the running time of item (1) is $O(m^{5/6}n^{2/3}\tau^{1/6}\alpha^{1/6}\log n)$ and the running time of item (5) is $O(m^{5/6}n^{2/3}\tau^{1/6}\alpha^{1/6}\log n)$.

It can easily be checked that $\delta \le \rho$ due to $\tau \le n$ and $\alpha \le n$. Now observe that the term $mn\tau/(\rho\delta)$ in the running time of item (2) is bounded by the term $O(mn\tau/\delta^2)$ that appears in the running time of item (6). Therefore the running time of item (2) is $O(m^{5/3}n^{1/3}\tau^{1/3}\log n/\alpha^{2/3})$. The term $n^2\tau/\delta^2$ in the running time of item (3) is also dominated by the term $mn\tau/\delta^2$ because $n \le m$. Therefore the running time of item (3) is $O(m^{5/3}n^{1/3}\tau^{1/3}\log n/\alpha^{2/3})$. Finally, since $10\kappa\delta = \alpha \le \tau$ and $\delta \le n$, we get $m\kappa \le m\alpha/(10\delta) \le m\tau/(10\delta) \le mn\tau/(10\delta^2)$ which is also $O(mn\tau/\delta^2)$. Thus, the running time of item (4) is $O(m^{5/3}n^{1/3}\tau^{1/3}\log n/\alpha^{2/3})$. We now have bounded all operations that we listed above by the desired running time. □

Finally, we turn the additive approximation into a multiplicative $(1+\epsilon)$-approximation. Observe that the running time of the additive algorithm has two terms. Thus, we want to distinguish between choices of $\tau$ that make the first term dominant and those that make the second term dominant. We first devise a $(1+\epsilon)$-approximation for distances up to depth $\tau \le m^{5/4}/n^{1/2}$ as follows. We run multiple instances of the additive approximation algorithm where the $i$-th instance, with depth $\tau_i$ and additive approximation $\alpha_i$ is responsible for providing $(1+\epsilon)$-approximate distances for the range from $2^i$ to $2^{i+1}$. In this first step we deal with values of $\tau_i \le \tau$ that make the first term dominant. In each of the multiple instances of the algorithm we set $\alpha_i$ to be a fraction of $\tau_i$ such that one factor of $\tau_i^{1/3}$ disappears in the term $m^{5/3}n^{1/3}\tau_i^{1/3}\log n/\alpha_i^{2/3}$. For small depths the additive approximation algorithm will be inefficient and we use the exact algorithm of Even and Shiloach instead.

LEMMA 3.8. *Given $\tau \le m^{5/4}/n^{1/2}$, there is an algorithm that maintains $(1+\epsilon)$-approximate single-source shortest paths up to depth $\tau$ in total time $O(m^{3/2}n^{1/4}\log n/\epsilon)$.*

*Proof.* We maintain an exact ES-tree rooted at $s$ up to depth $\tau_0 = m^{1/2}n^{1/4}$. For every node $v$ let $\hat{d}_0(v,s)$ denote the distance returned by this algorithm (which is $\infty$ for nodes that are in distance greater than $\tau_0$ from $s$).

Additionally, we run multiple instances of the additive approximation algorithm of Lemma 3.7. For each $1 \le i \le \lceil \log(\tau/\tau_0)\rceil$ we use the parameters $\tau_i = 2^i\tau_0$ and $\alpha_i = \epsilon\tau_{i-1}$ to compute an additive $\alpha_i$-approximation of shortest paths from $s$ up to depth $\tau_i$. For every node $v$ let $\hat{d}_i(v)$ denote the approximate distance between $v$ and $s$ returned by this algorithm. If $\tau_{i-1} \le d(v,s) \le \tau_i$, we get by Lemma 3.7:[6]

$$\begin{aligned} d(v,s) \le \hat{d}_i(v,s) &\le d(v,s) + \alpha_i \\ &= d(v,s) + \epsilon\tau_{i-1} \\ &\le d(v,s) + \epsilon d(v,s) = (1+\epsilon)d(v,s)\,. \end{aligned}$$

Now the whole distance range is covered and we simply have to return the minimum of $\hat{d}_0(v)$ and all $\hat{d}_i(v)$ to obtain a $(1+\epsilon)$-approximation of $d(v,s)$.

Maintaining an ES-tree up to depth $\tau_0$ takes time $O(m\tau_0) = O(m^{3/2}n^{1/4})$. By Lemma 3.7, running the additive approximation algorithm with parameters $\alpha_i = \epsilon\tau_{i-1}$ and $\tau_i = 2\tau_{i-1}$ takes time

$$O\left(\frac{m^{5/3}n^{1/3}\tau_{i-1}^{1/3}\log n}{\alpha_i^{2/3}} + m^{5/6}n^{2/3}\tau_{i-1}^{1/6}\alpha_i^{1/6}\log n\right)$$

which is

$$O\left(\frac{m^{5/3}n^{1/3}\log n}{\tau_i^{1/3}\epsilon} + m^{5/6}n^{2/3}\tau_i^{1/3}\log n\right).$$

(Note that $t_{i-1} = O(t_i)$.) Now we bound the running time for all multiple instances of this algorithm. Observe that

$$\sum_{i=1}^{\lceil \log(\tau/\tau_0)\rceil} \tau_i^{1/3} = \tau_0^{1/3} \sum_{i=1}^{\lceil \log(\tau/\tau_0)\rceil} 2^{i/3} = O(\tau^{1/3})$$

and similarly $\sum_{i=1}^{\lceil \log(\tau/\tau_0)\rceil} 1/\tau_i^{1/3} = O(1/\tau^{1/3})$. Thus, using the bounds $m^{1/2}n^{1/4} \le \tau \le m^{5/4}/n^{1/2}$, the running time of all multiple instances of the algorithm is

$$\begin{aligned} &O\left(\frac{m^{5/3}n^{1/3}\log n}{\tau^{1/3}\epsilon} + m^{5/6}n^{2/3}\tau^{1/3}\log n\right) \\ &\qquad = O(m^{3/2}n^{1/4}\log n/\epsilon)\,. \end{aligned}$$

As we want to answer distance queries in constant time, we keep a heap for every node that maintains the minimum of the distance estimates of all multiple instances of the additive approximation algorithm. In

[6] Precisely speaking, the algorithm returns a distance estimate $\hat{d}'_i(v,s)$ such that $d(v,s) - \alpha \le \hat{d}'_i(v,s) \le d(v,s)$. By defining $\hat{d}_i(v,s) = \hat{d}'_i(v,s) + \alpha$ we get $d(v,s) \le \hat{d}_i(v,s) \le d(v,s) + \alpha$.

the additive algorithm, the distance estimate changes only at the beginning of a new phase. The running time for updating the distance estimates in the heaps is $O(\log\log n)$ per node as every heap has $O(\log n)$ elements. We charge this additional running time of $O(n \log\log n)$ to the beginning of the new phase. This does not affect the worst-case time bound of Lemma 3.7. □

We obtain a $(1+\epsilon)$-approximation for arbitrary depth by using the algorithm above for small depth and repeating the idea of multiple instances of the additive approximation algorithm. In this second step, the second term in the running time of Lemma 3.7 is the dominating one. Note that we will set the additive approximation $\alpha_i$ of the $i$-th instance of the algorithm in a way that balances the running times of Lemma 3.7 and Lemma 3.8.

THEOREM 3.9. *There is an algorithm that maintains $(1+\epsilon)$-approximate single-source shortest paths under edge deletions up to depth $\tau$ in total time $O(mn^{3/5}\tau^{1/5}\log n/\epsilon + m^{3/2}n^{1/4}\log n/\epsilon)$.*

Our $(1+\epsilon)$-approximate decremental SSSP algorithm for full depth sets $\tau = n$ and thus has a total update time of $O(mn^{4/5}\log n/\epsilon + m^{3/2}n^{1/4}\log n/\epsilon)$.

### 3.3 Removing the Connectedness Assumption

The algorithm presented above only works when the graph stays connected. If a nodes gets disconnected from $s$, then we cannot guarantee any distance increases of other nodes (i.e., Lemma 3.5 does not work anymore). If we would simply run the algorithm above, we would have to start a new phase every time a node gets disconnected from $s$ which would lead to an $O(mn)$ time bound. There is a simple way to avoid this problem.

We run our $(1+\epsilon)$-approximate decremental SSSP algorithm on a graph $G'$ that, in addition to the nodes and edges of $G$, contains the following nodes and edges: nodes $u_i$ for $1 \le i \le 2n-1$, an edge $(u_1, s)$, edges $(u_{i+1}, u_i)$ for $1 \le i \le 2n-2$, and edges $(v, u_{2n-1})$ for every node $v$. Thus, we have added to $G'$ a path of length $2n$ between $s$ and every node of $G$. As we only have to consider deletions of the original edges of $G$, $G'$ is always connected. If a node $v$ of $G$ is connected to $s$ in $G$, then clearly $d_{G'}(v,s) = d_G(v,s)$. Otherwise we have $d_G(v,s) = 2n$. We answer a query for the distance between $s$ and a node $v$ as follows. We first query the algorithm on $G'$ to obtain a distance estimate $\delta(v,s)$ such that $d_{G'}(v,s) \le \delta(v,s) \le (1+\epsilon)d_{G'}(v,s)$. If $\delta(v,s) < 2n$, then $d_{G'}(v,s) < 2n$ and therefore also $d_{G'}(v,s) = d_G(v,s)$. Thus, we correctly answer the query by returning $\delta(v,s)$. If $\delta(v,s) \ge 2n$, then $d_{G'}(v,s) \ge 2n/(1+\epsilon) \ge n$ because $\epsilon \le 1$. As all nodes and edges of $G$ are contained in $G'$ we have $d_G(v,s) \ge d_{G'}(v,s)$ and therefore $d_G(v,s) \ge n$. As every path in $G$ has length at most $n-1$ this actually means that $d_G(v,s) = \infty$, i.e., $v$ and $s$ are not connected in $G$. Thus, we correctly answer the query by returning $\infty$. Clearly, this approach gives constant query time and, as $G'$ has $O(n)$ nodes and $O(m)$ edges, does not worsen the running time bound.

## 4 Running Decremental SSSP Algorithm on an Emulator

In the following we run our algorithm on the Thorup-Zwick emulator $H$ which is a sparse graph that has the same nodes as $G$ and approximates distances in $G$. This is non-trivial as we have to adopt our algorithm to deal with several complications. To avoid confusion we denote the current distance between $x$ and $y$ in $G$ by $d_G(x,y)$ and the current distance between $x$ and $y$ in $H$ by $d_H(x,y)$.

### 4.1 The Thorup-Zwick Emulator

We first review the central notions of Thorup and Zwick [24, 25] and define an (undirected) emulator $H$ of the graph $G$. This emulator has also been used in the decremental shortest-paths algorithms of Bernstein [6] and Bernstein and Roditty [8]. Given an integer $k$ such that $2 \le k \le \log n$, we define a hierarchy of *centers* $A_0 \supseteq A_1 \supseteq \ldots \supseteq A_k$ as follows. We let $A_0 = V$ and $A_k = \emptyset$, and for $1 \le i < k$ we obtain $A_i$ by picking each node from $A_{i-1}$ with probability $(\ln n/n)^{1/k}$. We say that a node $v$ has *priority* $i$ if $v \in A_i \setminus A_{i+1}$ (for $0 \le i \le k-1$).

The central notion of Thorup and Zwick is the *bunch* of a node $u$. As usual, we denote by $d_G(u, A_i) = \min_{v \in A_i} d_G(u,v)$ the distance between the node $u$ and the set of nodes $A_i$. Now the bunch of $u$ is defined as

$$B(u) = \bigcup_{0 \le i \le k-1} \{v \in A_i \setminus A_{i+1} \mid d_G(u,v) < d_G(u, A_{i+1})\}.$$

Intuitively, a node $v$ of priority $i$ is in the bunch of $u$ if $v$ is closer to $u$ than any node of priority greater than $i$. We will only need the following "truncated" version of the bunches: $B_\gamma(u) = \{v \in B(v) \mid d_G(u,v) \le \gamma\}$ where $\gamma = 2/\epsilon \cdot (1+2/\epsilon)^{k-2}$. We may assume that $\gamma \le n$. The relevance of this parameter choice will become clear later on. Given nodes $u$ and $v$, the emulator $H$ contains an edge $(u,v)$ if and only if $v \in B_\gamma(u)$. The weight of each edge $(u,v)$ in $H$ is set to $d_G(u,v)$, the distance between $u$ and $v$ in $G$. Thus each edge in $H$ has weight at most $\gamma$.

LEMMA 4.1. ([24]) *At any time, the size of the bunch $B(v)$ of every node $v$ is at most*

$20n^{1/k}\ln^{1-1/k} n = O(n^{1/k}\log^{1-1/k} n)$ *whp. Therefore* $H$ *has* $O(n^{1+1/k}\log^{1-1/k} n)$ *edges whp.*

It has been shown that $H$ is a $(1+\epsilon, \epsilon\gamma)$-emulator [25][7], which means that $H$ has the same set of nodes as $G$ and $d_G(x,y) \le d_H(x,y) \le (1+\epsilon)d_G(x,y) + (1+2/\epsilon)^{k-2}$ for all pairs of nodes $x$ and $y$. As the graph $G$ undergoes edge deletions, three types of *updates* can occur in $H$: edge deletions, edge insertions, and edge weight increases. The edge *insertions* are the reason why it is challenging to use this emulator in the decremental setting. By a result of Roditty and Zwick [22] the emulator can be efficiently maintained in the sense that after each deletion in $G$ the corresponding updates in $H$ are computed.

LEMMA 4.2. ([22]) *The emulator* $H$ *can be maintained in expected total time* $O(\gamma m n^{1/k})$.

Although the emulator is not purely decremental, the edge insertions into $H$ are not completely arbitrary. Specifically, there are strong bounds on the total number of edges inserted into the emulator and on the total number of update operations on the emulator.

LEMMA 4.3. ([8]) *The number of nodes inserted into the bunch* $B(v)$ *of a node* $v$ *is at most* $20k\gamma n^{1/k}\ln n = O(k\gamma n^{1/k}\log n)$ *whp. Thus, the number of edges inserted into* $H$ *is* $O(k\gamma n^{1+1/k}\log n)$ *whp.*

LEMMA 4.4. *The number of updates in* $H$ *is* $O(k\gamma^2 n^{1+1/k}\log n)$ *whp.*

Another useful observation bounds the maximum distance between the two endpoints of an inserted edge by $3\gamma$. This implies that each edge insertion can reduce the distance between any two nodes in $H$ by at most $3\gamma$.

LEMMA 4.5. ([8]) *If an edge* $(x,y)$ *is inserted into* $H$*, then before the insertion* $d_H(x,y) \le 3\gamma$.

**4.2 Monotone Even-Shiloach Tree** In a previous paper [16], we introduced a data structure called *monotone Even-Shiloach tree (monotone ES-tree)*, which is a modification of the well-known algorithm of Even and Shiloach [14]. The standard ES-tree allows edge deletions [14] and weight increases [19]. Our modification also allows edge insertions and is suited for certain dynamic weighted emulators of decremental graphs where the insertions are the result of edge deletions in the original graph. Previously we gave an emulator such that the monotone ES-tree maintains $(1+\epsilon, 2)$-approximate distances in time $\widetilde{O}(n^{2.5})$ [16]. In the following we provide a generalization of this result. By running the monotone ES-tree on the Thorup-Zwick emulator $H$ it can provide $(1+\epsilon, 2(1+2/\epsilon)^{k-2})$-approximate distances.

THEOREM 4.6. *Let* $0 < \epsilon < 1$*, let* $k$ *be an integer such that* $2 \le k \le \log n$ *and set* $\gamma = 2/\epsilon \cdot (1+2/\epsilon)^{k-2}$*. The monotone ES-tree with root* $r$ *up to depth* $\tau$ *on the Thorup-Zwick emulator* $H$ *maintains, for every node* $v$*, a (non-decreasing) distance estimate* $\hat{d}(v,r) \in \{0, 1, \ldots, \lfloor(1+\epsilon)\tau+\epsilon\gamma\rfloor, \infty\}$ *such that* $d_G(v,r) \le \hat{d}(v,r)$*. If* $d_G(v,r) \le \tau$*, then* $\hat{d}(v,r) \le (1+\epsilon)d_G(v,r) + 2(1+2/\epsilon)^{k-2} = (1+\epsilon)d_G(v,r) + \epsilon\gamma$*. Its total update time is* $O((\tau+\gamma)|E'|\log n + \#up)$*. Here* $\#up$ *is the total number of updates in* $H$ *and* $E' \subseteq E(H)$ *is the set of edges incident to nodes that are contained in the monotone ES-tree at some time (i.e., all nodes* $v$ *for which* $\hat{d}(v,r) \ne \infty$ *at some time).*

The running time analysis of the monotone ES-tree follows standard arguments (see Lemma A.1 and [16] for details). Note that a useful bound on $|E'|$ is the number of initial edges in $H$ plus the number of edges inserted into $H$. We now give a short description of the monotone ES-tree for completeness and afterwards prove the claimed approximation guarantee, which is a new contribution of this paper.

**Algorithm Description.** The monotone ES-tree maintains an approximate shortest paths tree up to depth $\tau$ to a given root node $r$ (see Algorithm 3 for the pseudocode). For every node $x$, the algorithm keeps a label $l(x)$, called the *level* of $x$, which is intended to provide the approximate distance between $x$ and the root $r$. Note that $l(x)$ is *not necessarily* equal to the distance between $x$ and $r$ in $H$. We set the maximum allowed level of a node in the tree to $(1+\epsilon)\tau + \epsilon\gamma$ as we are interested in $(1+\epsilon, \epsilon\gamma)$-approximate distances. For all nodes that are not connected to the tree we set the level to $\infty$. After the deletion of an edge in $G$, we report the corresponding updates in $H$ to the ES-tree in the following order: first we pass on the edge insertions, then we pass on the edge deletions and weight increases.[8]

[7] Thorup and Zwick [25] only considered the static setting and gave a $(1+\epsilon, \epsilon\gamma)$-spanner. A spanner is an emulator that is a subgraph of the original graph. Bernstein [6], who worked in the dynamic setting, slightly modified their construction and obtained an emulator with the same approximation guarantee. In particular, he observed that edges of weight greater than $\gamma$ can be ignored without affecting the approximation guarantee, which has advantages in the dynamic setting.

[8] If the edge deletions or edge weight increases were reported first, some nodes might undergo unnecessary level increases as the inserted edges might justify lower levels. Clearly, this should be avoided since the algorithm never performs any level decreases.

Our algorithm now proceeds just like the usual Even-Shiloach algorithm, except for the case of insertions. In the monotone ES-tree we process the insertion of an edge $(u, v)$ as follows. The current level $l(u)$ of $u$ is compared to the value $l'(u) = l(v) + w(u, v)$ that could be achieved by connecting $u$ to $v$ in the tree using the new edge $(u, v)$. If $l'(u) < l(u)$, then the algorithm makes $v$ the parent of $u$ in the tree, otherwise the tree is not changed. In either case the level $l(u)$ of $u$ is *not* changed. Edge deletions and edge weight increases are handled just like in the usual ES-tree by running a reconnection procedure that tries to connect every node $u$ to a neighbor $v$ in $G$ such that $l(v) + w(u, v)$ is minimized.

**Approximation guarantee.** The proof of Thorup and Zwick [25] shows that $H$ (without any updates) is a $(1 + \epsilon, 2(1 + 2/\epsilon)^{k-2})$-emulator. We want to prove that the level $l(v)$ of a node $v$ in our monotone ES-tree provides the same approximation guarantee for the distance between $v$ and the root $r$. In principle we apply the same proof idea as Thorup and Zwick. There is however a technical challenge. The proof of Thorup and Zwick goes by induction on the distance between $v$ and $r$. To make their approach work for the monotone ES-tree, we use "double induction" increasing the distance between $v$ and $r$ and decreasing the priority of $v$, where the priority of a node in $H$ is defined as in Section 4.1.

For our proof, we define the numbers $a_i$ and $b_i$ and for $1 \le i \le k-1$ as follows:

$$a_0 = 1 \qquad a_i = \frac{2}{\epsilon}\left(1 + \frac{2}{\epsilon}\right)^{i-1} \text{ (if } i \ge 1)$$

$$b_0 = 2\left(1 + \frac{2}{\epsilon}\right)^{k-2} \qquad b_i = b_0 - 2\left(1 + \frac{2}{\epsilon}\right)^{i-1} \text{ (if } i \ge 1)$$

We will show that for a node of priority $i$, the monotone ES-tree provides a $(1 + \epsilon, b_i)$-approximation. This implies that the approximation guarantee is $(1 + \epsilon, \epsilon\gamma)$ for every node because $b_0 = \epsilon\gamma$ and $b_i \le b_0$ for every $i \le k-1$.

Let us first briefly sketch the proof idea. Consider a node $x$ that is by distance $a_i$ nearer to the root than $v$. If $H$ contains the edge $(v, x)$, then we argue as follows. By properties of the monotone ES-tree we may assume that $l(v) \le l(x) + w(v, x)$, where $w(v, x) = d_G(v, x)$ is the weight of the edge $(v, x)$ in $H$. We then only have to apply induction on $x$ (distance to $r$ has decreased) because we add no further error by going from $v$ to $x$. If the edge $(x, y)$ is not contained in $H$, then the properties of $H$ guarantee that there is a node $y$ of priority at least $i+1$ in distance at most $2a_i$ to $v$. Therefore we can apply induction on $y$ (priority has increased). We then have to show that the error introduced by going from $v$ to $y$ is within the bounds we claim.

To carry out the proof in full detail we introduce the concept of stretched nodes and provide a few simple observations about the monotone ES-tree (see also [16]). We say that a node $u$ is *stretched* if $l(u) > l(v) + w(u, v)$ for some edge $(u, v)$. Note that for a node $u$ that is not stretched we have $l(u) \le l(v) + w(u, v)$ for every edge $(u, v) \in E(H_i)$. It is easy to see that the following holds for the monotone ES-tree:

- The level of a node never decreases.
- A node $x$ can only become stretched after the insertion of an edge $(x, y)$ into $H$.
- As long as a node is stretched, its level does not change.
- For every tree edge $(u, v)$ (where $v$ is the parent of $u$), $l(u) \ge l(v) + w(u, v)$.

The formal statement of the approximation guarantee in the next lemma assumes that $1/\epsilon$ is integer. However, this restriction does not limit our algorithms at all because given any $\epsilon \le 1$ we can quickly find the largest $\epsilon'$ smaller than $\epsilon$ that satisfies our restriction.

LEMMA 4.7. *Let $0 < \epsilon < 1$ such that $1/\epsilon$ is integer. For every node $v$ of priority $i$, we have $l(v) \le (1 + \epsilon)d_G(v, r) + b_i$.*

*Proof.* Note that we have set the $a_i$'s and $b_i$'s in a way such that $a_0 < a_1 < \ldots < a_{k-1} = \gamma$, $\epsilon\gamma = b_0 > b_1 > \ldots > b_{k-1}$ and three useful inequalities hold.

LEMMA 4.8. *If $\epsilon \le 1$, then the following inequalities hold: (1) $b_0 \le b_i + \epsilon a_i$, (2) $b_{i+1} + 2a_i = b_i$, (3) $b_{i+2} + (4 + 2\epsilon)a_i \le b_i$.*

The claim that $l(v) \le (1+\epsilon)d_G(v, r) + b_i$ is certainly true after the initialization and we show that it also holds every time the algorithm has processed an edge deletion (assuming it was true before the deletion). If $v$ is stretched, then the claim is trivially true because the level of $v$ has not changed since before the edge deletion and the distance between $v$ and $r$ did not decrease since then. Thus, we assume in the following that $v$ is not stretched. We prove the claim by induction on the priority $i$ of $v$ and the distance $d_G(v, r)$ between $v$ and $r$. To be precise, our proof is by induction on the following function $f$. For every node $u$ and each priority $j$ we define $f(u, j) = 0$ if $u = r$ and $f(u, j) = d_G(u, r) + b_j$ otherwise. Note that $f(u, j)$ will only assume integer values and if $u \ne r$, then $f$ is monotonically decreasing in its second parameter. In our proof we will argue that for every node $u$ with priority $j$ to which we apply the induction hypothesis we have $f(u, j) < f(v, i)$.

We define the node $x$ as follows. If $d_G(v,r) < a_i$, then we set $x = r$. If $d_G(v,r) \geq a_i$, then let $x$ be a node on a shortest path from $v$ to $r$ in $G$ such that $d_G(v,x) = a_i$. (Note that we assume that $1/\epsilon$ is integer and therefore also $a_i$ is integer which means that such a node $x$ always exists.)

*Case 1:* The edge $(v,x)$ is contained in the emulator $H$. We then have $l(v) \leq l(x) + d_G(v,x)$ because $v$ is not stretched and the edge $(v,x)$ in $H$ has weight $d_G(v,x)$. If $x = r$, then we are done because $l(r) = 0$ and thus we trivially get $l(v) \leq d_G(v,r) \leq (1+\epsilon)d_G(v,x) + b_i$. We show below that we can apply the induction hypothesis, which gives $l(x) \leq (1+\epsilon)d_G(x,r) + b_0$ ($x$ does not necessarily have priority 0, but the bound still holds for any priority). By combining both inequalities we get

$$l(v) \leq (1+\epsilon)d_G(x,r) + b_0 + d_G(v,x)\,.$$

By Lemma 4.8 we have $b_0 \leq b_i + \epsilon a_i$. Since $a_i = d_G(v,x)$ (because $x \neq r$) and $d_G(v,r) = d_G(v,x) + d_G(x,r)$ we get $l(v) \leq (1+\epsilon)d_G(v,r) + b_i$. We now argue that we actually may apply the induction hypothesis on the node $x$ with priority $j_x \geq 0$ by showing that $f(x,j_x) < f(v,j)$. If $x$ is the root node $r$, then trivially $f(x,j_x) = 0 < f(v,i)$. If $x \neq r$, then $d_G(v,x) = a_i$ and we get the following chain of inequalities:

$$\begin{aligned} f(x,j_x) \leq f(x,0) &= d_G(x,r) + b_0 \\ &\leq d_G(x,r) + b_i + \epsilon a_i \\ &= d_G(x,r) + b_i + \epsilon d(v,x) \\ &< d_G(x,r) + b_i + d(v,x) \\ &= d_G(v,r) + b_i = f(v,i)\,. \end{aligned}$$

*Case 2:* The edge $(v,x)$ is not contained in $H$. Since $d_G(x,v) \leq a_i \leq \gamma$, this implies that the node $v$ of priority $i$ is not in the bunch of $x$. Thus $d_G(x,v) \geq d_G(x,A_{i+1})$ and there must be some node $y \in A_{i+1}$ of priority at least $i+1$ such that $d_G(x,y) \leq d_G(x,v)$.

*Case 2.1:* The edge $(v,y)$ is contained in $H$. We can then bound $l(v)$ as follows. Since $v$ is not stretched, we have $l(v) \leq l(y) + d_G(v,y)$. We show below that we can apply the induction hypothesis, which gives $l(y) \leq (1+\epsilon)d_G(y,r) + b_{i+1}$ because $y$ has priority at least $i+1$. By the triangle inequality we have

$$d_G(v,y) \leq d_G(v,x) + d_G(x,y) \leq 2d_G(v,x) \leq 2a_i$$

as well as $d_G(y,r) \leq d_G(v,r)$. By combining these inequalities we get

$$l(v) \leq (1+\epsilon)d_G(v,r) + b_{i+1} + 2a_i\,.$$

Note that by Lemma 4.8 we have $b_{i+1} + 2a_i \leq b_i$. Therefore, the desired inequality follows. We now argue that we actually may apply the induction hypothesis on the node $y$ with priority $j_y \geq i+1$ by showing that $f(y,j_y) < f(v,i)$. Simply consider the following inequalities:

$$\begin{aligned} f(y,j_y) \leq f(y,i+1) &= d_G(y,r) + b_{i+1} \\ &\leq d_G(v,r) + b_{i+1} \\ &< d_G(v,r) + b_i = f(v,i)\,. \end{aligned}$$

*Case 2.2:* The edge $(v,y)$ is not contained in $H$. Then $y$ is not in the bunch of $v$ and there must be some node $z$ of priority at least $i+2$ such that $d_G(v,z) \leq d_G(v,y)$ (where $d_G(v,y) \leq 2a_i$). Since $v$ is not stretched, we have $l(v) \leq l(z) + d_G(v,z)$. We show below that we can apply the induction hypothesis, which gives $l(z) \leq (1+\epsilon)d_G(z,r) + b_{i+2}$ because $z$ has priority at least $i+2$. By combining these inequalities we get

$$l(v) \leq (1+\epsilon)d_G(z,r) + b_{i+2} + 2a_i\,.$$

We now use the triangle inequality to obtain

$$d_G(z,r) \leq d_G(z,v) + d_G(v,r) \leq d_G(v,r) + 2a_i\,.$$

Therefore we get

$$l(v) \leq (1+\epsilon)d_G(v,r) + b_{i+2} + (4+2\epsilon)a_i$$

Since $b_{i+2} + (4+2\epsilon)a_i \leq b_i$ by Lemma 4.8, the desired inequality follows. Finally, we argue that we actually may apply the induction hypothesis on the node $z$ with priority $j_z \geq i+2$ by showing that $f(y,j_z) < f(v,j)$. We get the following chain of inequalities:

$$\begin{aligned} f(z,j_z) \leq f(z,i+2) &= d_G(z,r) + b_{i+2} \\ &\leq d_G(v,r) + d_G(z,v) + b_{i+2} \\ &\leq d_G(v,r) + 2a_i + b_{i+2} \\ &< d_G(v,r) + (4+2\epsilon)a_i + b_{i+2} \\ &\leq d_G(v,r) + b_i = f(v,i)\,. \end{aligned}$$

□

Now that we have proved the approximation guarantee of the monotone ES tree, it almost immediately follows from the techniques in [16] that there is a decremental all-pairs shortest paths algorithm with nearly the same approximation guarantee, as stated in Theorem 1.3.

**4.3 Modified Decremental SSSP Algorithm** To speed up the approximate decremental SSSP algorithm of Section 3.1 we want to run it on the emulator $H$ described in Sections 4.1 and 4.2 and not on $G$.

The main challenge is to deal with edges that are inserted into $H$. The fact that $H$ is a weighted graph is only a minor obstacle because the maximum edge weight is bounded by $\gamma$. Remember that the algorithm has parameters $\kappa$, $\delta$, and $\rho$ and that the Thorup-Zwick emulator has parameters has parameters $\epsilon$ and $\kappa$, which determine its approximation guarantee and the maximum edge weight $\gamma$.

To formulate and analyze the modified algorithm we will use the following terminology. The algorithm uses distinguished nodes called *centers*. For each center we maintain a *monotone* ES-tree on the emulator $H$. We say that a node $v$ is *contained in the monotone ES-tree of $c$ up to depth $t$*, if the distance estimate $\hat{d}(c,v)$ of the monotone ES-tree satisfies $\hat{d}(c,v) \leq (1+\epsilon)t + \epsilon\gamma$ (these values correspond to the approximation guarantee of the monotone ES-tree). By this definition we guarantee that $d_G(v,c) \leq t$ implies that $v$ is contained in the monotone ES-tree of $c$ up to depth $t$. We say that a node $v$ is *contained in the monotone ES-tree of $c$* if the distance estimate of $v$ to $c$ given by the monotone ES-tree is finite (and not $\infty$). An edge $(u,v)$ is *incident to the monotone ES-tree of $c$* if $u$ or $v$ is contained in the monotone ES-tree of $c$ and the edge $(u,v)$ exists in $H$. Furthermore we say that a node $v$ is *$t$-covered* by a center $c$ if $v$ is contained in the monotone ES-tree of $c$ up to depth $t$. Note that this definition of being $t$-covered deviates from the definition in Section 3.1.

To deal with the new setting we modify the algorithm of Section 3.1 as follows:

- Since the emulator $H$ is a *weighted* graph, we initialize each phase by computing a shortest paths tree rooted at $s$ *in the emulator* $H$ (and not in $G$) by running Dijkstra's algorithm.
- The following types of updates are possible in the emulator $H$: edge deletions, edge insertions, and edge weight increases. All of them could change the distances from $s$ in $H$. Thus, we start a new phase every time the algorithm has seen $\kappa$ updates in $H$.
- Due to the edge insertions the level of a node might sometimes decrease. Therefore we implement the priority queues of the centers by min-heaps.
- We replace, for each node $v$, the rounded level by the *delayed rounded level* $L'(v)$. Due to insertions of edges into the emulator it can happen that the rounded level $L(v)$ of a node decreases. The delayed rounded level $L'(v)$ will not undergo all these decreases. It is defined as follows: If $L(v)$ increases we set $L'(v) = L(v)$. If $L(v)$ decreases by only $\delta$ we do not change $L'(v)$. If $L(v)$ decreases by at least $2\delta$[9], we set $L'(v) = L(v)$. Thus $L(v) \leq L'(v) \leq L(v) + \delta$.
- As edges are inserted into $H$, we cannot determine the centers by sampling from the initial edges. Instead, we sample each node with probability $\frac{a20k\gamma n^{1/k}\ln^2 n}{\rho}$ for a large enough constant $a$ and use the corresponding nodes as rich centers. We also sample each node with probability $\frac{a\ln n}{\delta}$ and use the corresponding nodes as poor centers.
- Since $H$ is not purely decremental we use monotone ES-trees instead of ES-trees. To deal with the approximate nature of $H$ we also have to change the depths of these trees. For every rich center $c$ we maintain a monotone ES-tree rooted at $c$ up to depth $45\delta + 27\kappa\gamma + 28\gamma$. A poor center is active if it is not $(10\delta + 6\kappa\gamma + 6\gamma)$-covered by any rich center. For every active poor center we maintain a monotone ES-tree up to depth $5\delta + 3\kappa\gamma + 2\gamma$. Note that the distance estimates returned by monotone ES-trees are non-decreasing. Thus, as soon as a poor center becomes active it will remain active.

The pseudocode of the modified algorithm is given in Algorithm 2.

Having modified the algorithm as above, we now analyze its running time and approximation guarantee. This analysis closely follows the analysis of the original algorithm in Section 3. First, we show that the sampling process guarantees that every node is covered by a center.

LEMMA 4.9. *Every node is either $(21\delta + 12\kappa\gamma + 13\gamma)$-covered by a rich center or $\delta$-covered by an active poor center whp.*

Just as before, this lemma is needed for the correctness of the algorithm. We can state its approximation guarantee as follows.

LEMMA 4.10. *After every update processed by the algorithm the graph $T$ is a tree and, for every node $x$, $d_G(x,s) - (132\kappa\delta + 78\kappa^2\gamma + \kappa\gamma + 84\gamma) \leq l(x) \leq (1+\epsilon)d_G(x,s) + \epsilon\gamma$.*

We now show that the set of edges incident to the monotone ES-tree of an active poor center is at most $\rho$. This needs an argument involving the size of each bunch because we are sampling nodes instead of edges.

LEMMA 4.11. *Consider an active poor center $c$ and its monotone ES-tree. The number of edges ever incident to this monotone ES-tree since $c$ became active is at most $\rho$ whp.*

[9] Remember that the rounded level $L(v)$ is a multiple of $\delta$.

**Algorithm 2:** Additive approximation algorithm on emulator

```
   // After updating an edge (x,y), the corresponding update procedure has to be executed for both (x,y) and (y,x).
1  initialize()
       // First initialization and re-initialization at beginning of new phase
2      Compute the shortest-paths tree T rooted at s in H
3      foreach node v do set level l(v) to current distance between v and s in H and L(v) to ⌊l(v)/δ⌋ · δ
4      foreach node v such that L(v) has decreased by at least 2δ in line 3 do
5          L'(v) ← L(v)
6          foreach center c do set key of v in heap of c to L'(v)
7      U ← ∅
8  update(x, y)
       // Update of edge (x,y) in H (insertion, deletion, or weight increase)
9      if (x,y) is deleted and y is parent of x in tree T then add x to U
10     if there were κ updates since the beginning of the current phase then
11         initialize() // start new phase
12     else
13         foreach u ∈ U do
14             Find rich center c that (21δ + 12κγ + 13γ)-covers u or active poor center c that δ-covers u
15             Find node v with minimum delayed rounded level L'(v) in monotone ES-tree of c
16             if s is contained in monotone ES-tree of c then
17                 Make s the parent of u in T
18             else
19                 if L'(v) < l(u) − δ then
20                     Make v the parent of u in T
21                 else
22                     initialize() // start new phase
```

We already explained that the delayed rounded level does not change as often as the rounded level. We now give an upper bound on this number of changes, which will make updating the levels in the priority queues of the centers efficient.

LEMMA 4.12. *The delayed rounded level $L'(v)$ of a node $v$ changes at most $O(k\gamma^2 n^{1+1/k}\log n/\delta)$ times.*

There is one complication left that we have to consider. Due to the edge insertions the level of a node $v$ computed at the beginning of the current phase might actually underestimate the current distance between $v$ and $s$ in $H$. However, we can show that this difference is bounded by the number of insertions since the beginning of the phase.

LEMMA 4.13. *At any time, for every node $v$ it holds that $l(v) \le d_H(v,s) + 3\kappa\gamma$.*

With the help of this lemma it can be shown that when the reconnection procedure fails for a node $u$, the distance between $u$ and $s$ has increased by at least $\delta$.

LEMMA 4.14. *Let $u \in U$ be a node whose edge to its parent has been deleted in the current phase. Let $c$ be a rich center that $(21\delta + 12\kappa\gamma + 13\gamma)$-covers $u$ or an active poor center that $\delta$-covers $u$. Let $v$ be a node of minimum delayed rounded level $L'(v)$ in the monotone ES-tree of $c$ and assume that $s$ is not contained in the monotone ES-tree tree of $c$. If $L'(v) \ge l(u) - \delta$, then $d_H(u,s) \ge l(u) + \delta$.*

Again, this implies that also the distance from $s$ of other nodes increases. In particular there is a set of nodes whose sum of distances from $s$ increases by $\Omega((\delta^2/\gamma) - \delta)$ (as $\gamma$ is the maximum edge weight in $H$).

LEMMA 4.15. *If there is a node $u$ such that $d_H(u,s) \ge l(u) + \delta$, then there is a set of nodes $V'$ such that $\sum_{v\in V'}(d_H(v,s) - l(v)) = \Omega((\delta^2/\gamma) - \delta)$.*

These increases might later on be revoked due to edge insertions, but this cannot happen too often as the number of edge insertions and the distance increase per insertion are limited. Thus, we can bound the number of phases as follows.

LEMMA 4.16. *Whp the number of phases is*

$$O\left(\frac{k\gamma^2 n^{1+1/k}\log n}{\kappa} + \frac{k\gamma^3 n^{2+1/k}\log n}{\delta^2 - \gamma\delta}\right).$$

The overall running time analysis now is very similar to the original algorithm of Section 3.1 and is done with the parameter choice $k = \sqrt{\log n}/\sqrt{\log(3/\epsilon)}$, $\kappa = n^{1/5}$, $\delta = n^{3/5}$, and $\rho = n^{4/5}$.

PROPOSITION 4.1. *There is an algorithm that maintains a distance estimate $\hat{d}(v,s)$ for every node $v$ under edge deletions such that $d_G(v,s) \le \hat{d}(v,s) \le (1+\epsilon)d_G(v,s) + O(n^{4/5+1/k})$. If $\epsilon \ge 3/n^{1/4}$, its total update time is $O(k^2 n^{9/5+5/k}\log^4 n + mn^{2/k})$ in expectation where $k = \sqrt{\log n/\log(3/\epsilon)}$.*

*Proof.* We use the parameters $\kappa = n^{1/5}$, $\delta = n^{3/5}$, and $\rho = n^{4/5}$ to obtain an additive approximation. (We will later on use the inequality $\kappa \le \delta$.) We now explain how to set the parameter $k$ (which will also determine $\gamma$). For simplicity we will use the following rough bound:

$$\gamma = 2/\epsilon\,(1+2/\epsilon)^{k-2} \le (1+2/\epsilon)\,(1+2/\epsilon)^{k-2} \le (1+2/\epsilon)^k \le (3/\epsilon)^k\,.$$

The last inequality holds because $\epsilon \le 1$. In our algorithm we set $k = \sqrt{\log n/\log(3/\epsilon)}$. With this choice of $k$ we have $n^{1/k} = (3/\epsilon)^k$ and therefore also $\gamma \le n^{1/k}$ and $\delta\gamma \le n^{3/5+1/k}$. We use the technical assumption $\epsilon \ge 12/n^{1/4}$ to guarantee that $1/k \le 1/2$.

Using these values, we get the following bounds on important quantities of the algorithm:

- At any time, the number of edges of $H$ is $O(n^{1+1/k}\log n)$ (Lemma 4.1)
- The number of insertions into $H$ is $O(kn^{1+2/k}\log n)$ (Lemma 4.3)
- The number of updates on $H$ is $O(kn^{1+3/k}\log n)$ (Lemma 4.4)
- The number of phases of the algorithm is $O(kn^{4/5+4/k}\log n)$ (Lemma 4.16)[10]
- The expected number of rich centers is $O(kn^{1/5+2/k}\log^2 n)$.
- The expected number of poor centers is $O(n^{2/5}\log n)$.

The costs of our algorithm can now be bounded as follows:

(1) Computing a shortest paths tree in $H$ at the beginning of each phase takes time $O(|E(H)| + n\log n)$ with Dijkstra's algorithm where $|E(H)|$ is the current number of edges of $H$. The number of edges is $O(n^{1+1/k}\log n)$ and therefore the first term dominates the second term. Since we do this at the beginning of every phase, this takes time $O(kn^{9/5+5/k}\log^2 n)$ in total.

(2) We have to maintain the monotone ES-trees of rich centers up to depth $45\delta + 27\kappa\gamma + 28\gamma$ which is $O(\delta\gamma)$ because $\kappa \le \delta$. We bound $|E'|$ in Theorem 4.6 by the number of initial edges in $H$ plus the number of edges inserted into $H$, which is $|E'| \le O(kn^{1+2/k}\log n)$. By Theorem 4.6 maintaining one such tree takes time $O((\delta\gamma + \gamma)kn^{1+2/k}\log^2 n + kn^{1+3/k}\log n)$, where the second term comes from the number of updates in $H$. As the second term is dominated by the first term, this is $O(kn^{8/5+3/k}\log^2 n)$. Thus, the total running time of this task for all rich centers is $O(k^2 n^{9/5+5/k}\log^4 n)$.

(3) The number of edges ever incident to the monotone ES-tree of each active poor center is at most $\rho$ (Lemma 4.11). By Theorem 4.6 maintaining one ES-tree of a poor center up to depth $5\delta + 3\kappa\gamma + 2\gamma$ (which is $O(\delta\gamma)$) therefore takes time $O((\delta\gamma+\gamma)\rho\log n + kn^{1+3/k}\log n)$, which is $O(n^{7/5+1/k}\log n)$ (the second term comes from the number of updates in $H$ and is dominated by the first term). Thus, the total running time of this task for all active poor centers is $O(n^{9/5+1/k}\log^2 n)$.

(4) For every node $v$, the total time needed for updating the value of $L'(v)$ in the heap of a center is $O(k\gamma^2 n^{1+1/k}\log^2 n/\delta)$ (which is $O(kn^{2/5+3/k}\log^2 n)$) because $L(v)$ changes at most $O(k\gamma^2 n^{1+1/k}\log n/\delta)$ times (Lemma 4.12) and each update takes time $O(\log n)$. Therefore the total time needed for updating the values $L(v)$ of all nodes $v$ in the heaps of all rich centers is $O(k^2 n^{8/5+5/k}\log^4 n)$.

(5) Similarly, the time needed for maintaining the values $L(v)$ of all nodes $v$ in the heaps of all poor centers is $O(kn^{9/5+3/k}\log^3 n)$.

(6) The running time of the reconnection procedure after an update is $O(1)$ for every node in $U$. As there are at most $\kappa$ nodes in $U$ this takes time $O(\kappa) = O(n^{1/5})$ after every update in $H$ and thus time $O(kn^{6/5+3/k}\log n)$ in total.

All these running times are dominated by $O(k^2 n^{9/5+5/k}\log^4 n)$. Independently of our algorithm we have to maintain the emulator $H$ under edge deletions in $G$. By Lemma 4.2 this takes total time $O(m\gamma n^{1/k}) = O(mn^{2/k})$ in expectation. By plugging in the values for $\kappa$, $\delta$, and $\gamma$ in Lemma 4.10 we can state the approximation guarantee of our algorithm as given above. □

[10] Note that the term $\gamma\delta = n^{1/k+3/5}$ is dominated by $\delta^2 = n^{6/5}$ because $1/k \le 1/2$.

For turning the additive approximation into a multiplicative $(1+\epsilon)$-approximation we use the same standard technique as in Section 3. The above algorithm already provides a $(1+\epsilon)$-approximation for distances that are large enough. To deal with small distances, we separately run a *monotone* ES-tree rooted at $s$ up to depth $O(n^{4/5+1/k}/\epsilon)$. This leaves us with an additive error of $n^{1/k}$.

PROPOSITION 4.2. *There is an algorithm that maintains a distance estimate $\hat{d}(v,s)$ for every node $v$ under edge deletions such that $d_G(v,s) \le \hat{d}(v,s) \le (1+\epsilon)d_G(v,s) + n^{1/k}$. If $\epsilon \ge 6/n^{1/4}$, its total update time is $O(k^2 n^{9/5+5/k}\log^4 n/\epsilon + mn^{2/k})$ in expectation where $k = \sqrt{\log n/\log(6/\epsilon)}$.*

We combine this algorithm with an *exact* ES-tree up to depth $n^{1/k}/\epsilon$ on the *original* graph $G$ to get the desired $(1+\epsilon)$-approximation.

THEOREM 4.17. *There is an algorithm that maintains a distance estimate $\hat{d}(v,s)$ for every node $v$ under edge deletions such that $d_G(v,s) \le \hat{d}(v,s) \le (1+\epsilon)d_G(v,s)$. If $\epsilon \ge 12/n^{1/4}$, its total update time is $O(k^2 n^{9/5+5/k}\log^4 n/\epsilon + mn^{2/k}/\epsilon)$ in expectation where $k = \sqrt{\log n/\log(12/\epsilon)}$.*

If $\epsilon$ is a constant then the total running time of the algorithm is $O(n^{9/5+O(1/\sqrt{\log n})} + m^{O(1/\sqrt{\log n})})$ as stated in Theorem 1.1.

## 5 $(3+\epsilon)$-approximate Decremental APSP

We now prove that our techniques also lead to a faster algorithm for decremental APSP, with a $(3+\epsilon)$ approximation guarantee[11]. Recall that, as outlined in Section 1 (see Claim 1.4 and the following paragraph), we have a $\widetilde{O}(mn/\sqrt{h})$-time center cover data structure. Additionally, by Theorem 1.1, we can $(1+\epsilon)$-approximately solve decremental SSSP from $\widetilde{O}(h)$ centers in $\widetilde{O}(hn^{1.8+O(1/\sqrt{\log n})})$ total update time. The main idea is that, by using the $\widetilde{O}(mn/\sqrt{h})$-time algorithm when $h \ge (m/n^{0.8})^{2/3}$ and the $\widetilde{O}(hn^{1.8+O(1/\sqrt{\log n})})$ algorithm in other cases, we can always maintain the center cover data structure in $\widetilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$ total update time, which is faster than $\widetilde{O}(mn)$. We then plug this data structure into the framework of Roditty and Zwick [22] to solve decremental APSP with the same total update time. We now describe this idea in more detail. (For more details on the framework of Roditty and Zwick, especially how the center cover data structure plays a role, see [16].)

Let $h^* = (m/n^{0.8})^{2/3}$, $h_i = (1+\epsilon)^i$ and $\alpha = (1+\epsilon)^2/(\epsilon-\epsilon^2)$. For any integer $i \ge 0$ such that $h_i \ge h^*$, we maintain a $\widetilde{O}(mn/\sqrt{h_i})$-time center cover data structure denoted by $\mathcal{C}_i$ with parameters $\alpha$ and $h_i$. Additionally, we use the $\widetilde{O}(n^{1.8+O(1/\sqrt{\log n})})$-time algorithm to maintain $(1+\epsilon)$-approximate distances between every node and $\widetilde{O}(h^*)$ randomly selected centers, which we will call *super centers*. For each node $u$ and integer $i$, we let $c_{u,i}$ be a center in $\mathcal{C}_i$ that covers $u$ in the sense of Claim 1.4; i.e., either (i) $c_{i,u}$ is an active poor center of distance at most $n/h_i$ from $u$ and $c_{i,u}$ is the root of an ES-tree of depth $\alpha n/h_i$ or (ii) $c_{i,u}$ is a rich center of distance at most $(\alpha+1)n/h_i$ from $u$ and $c_{i,u}$ is the root of an ES-tree of depth $2\alpha n/h_i$. We also let $c'_u$ be a super center of distance at most $n/h^*$ from $u$ (which exists with high probability). We can maintain $c_{i,u}$ and $c'_u$ without an additional cost since we already maintain an ES-tree from these centers.

When we get a query for a distance between two nodes $u$ and $v$, we use

$$\hat{d}(u,v) = \min\begin{cases}\min_i d_G(u,c_{i,u}) + d_G(c_{i,u},v)\\ d_G(u,c'_u) + d_G(c'_u,v)\end{cases}$$

as an answer to the query, where $G$ is the current graph. We note the following. (i) It is possible that, for some $i$, we do not know the value of $d_G(c_{i,u},v)$ since $v$ is not contained in the ES-tree rooted at $c_{i,u}$. In this case, we treat $d_G(c_{i,u},v)$ as $\infty$. (ii) We cannot compute $d_G(u,c'_u)$ and $d_G(c'_u,v)$ exactly but we can get $(1+\epsilon)$-approximate values of them since our algorithm maintains a $(1+\epsilon)$-approximate solution of SSSP using $c'_u$ as a source. Thus, we will get a $(1+\epsilon)$-approximate solution of $\hat{d}(u,v)$. (iii) We need $O(\log n)$ time to compute $\hat{d}(u,v)$ and thus obtain an $O(\log n)$ query time. By using a standard technique we can slightly change the definition of $\hat{d}(u,v)$ to improve the query time to $O(\log\log n)$[12]. We omit the detail of this improvement here to keep the argument simple (for the details of this argument, see e.g. [22, 16]).

We end our proof with the following approximation guarantee.

CLAIM 5.1. *If $0 \le \epsilon \le 1/2$, then $d_G(u,v) \le \hat{d}(u,v) \le (3+16\epsilon)d_G(u,v)$.*

[11] To simplify the presentation, we assume in this section that $\epsilon$ is a constant such that $0 < \epsilon \le 1$. Our $O(\cdot)$ in the running time will hide $\mathrm{poly}(1/\epsilon)$; e.g., the running time of $\widetilde{O}(hn^{1.8+O(1/\sqrt{\log n})})$ is in fact $\widetilde{O}(hn^{1.8+O(\mathrm{poly}(1/\epsilon)/\sqrt{\log n})}\,\mathrm{poly}(1/\epsilon))$.

[12] In particular, we can define $i'$ to be the minimum $i$ such that the ES-tree rooted at $c_{i,u}$ contains $v$ and use $\hat{d}'(u,v) = \min\left(d_G(u,c_{i',u}) + d_G(c_{i',u},v), d_G(u,c'_u) + d_G(c'_u,v)\right)$ as the query answer. We can find $i'$, and thus compute $\hat{d}'(u,v)$, in $O(\log\log n)$ time using binary search.

*Proof.* The first inequality is simply by the triangle inequality. We now prove the second inequality.

*Case 1: When $n/h^* \le \epsilon d_G(u,v)$.* In this case, we know that $d_G(u,c'_u) \le n/h^* \le \epsilon d_G(u,v)$. It then follows that $d_G(c'_u,v) \le d_G(u,c'_u) + d_G(u,v) \le (1+\epsilon)d_G(u,v)$. Thus, $\hat{d}(u,v) \le d_G(u,c'_u)+d_G(c'_u,v) \le (1+2\epsilon)d_G(u,v)$.

*Case 2: When $n/h^* > \epsilon d_G(u,v)$.* Let $i^*$ be such that $\epsilon d_G(u,v)/(1+\epsilon) < n/h_{i^*} \le \epsilon d_G(u,v)$. Note that $h_{i^*} \ge h^*$ since $n/h^* > \epsilon d_G(u,v) \ge n/h_{i^*}$; thus, our algorithm maintains a center cover data structure $\mathcal{C}_{i^*}$. Let $c_{i^*,u}$ be the center in $\mathcal{C}_{i^*}$ covering $u$. Consider two subcases.

*Case 2.1:* If $c_{i^*,u}$ is an active poor center, then we know that $d_G(c_{i^*,u},u) \le n/h_{i^*} \le \epsilon d_G(u,v)$. As in Case 1, it follows that

$$d_G(c_{i^*,u},v) \le d_G(c_{i^*,u},u) + d_G(u,v) \le (1+\epsilon)d_G(u,v).$$

Note that our algorithm maintains an ES-tree rooted at $c_{i^*,u}$ of depth $\tau = \alpha n/h_{i^*}$. Using $n/h_{i^*} > \epsilon d_G(u,v)/(1+\epsilon)$ and $\alpha = (1+\epsilon)^2/(\epsilon-\epsilon^2)$, we have

$$\tau > \frac{1+\epsilon}{1-\epsilon} d_G(u,v) \ge (1+\epsilon)d_G(u,v).$$

Thus, this ES-tree contains $v$ and we can compute $d_G(c_{i^*,u},v)$. It follows that $\hat{d}(u,v) \le d_G(u,c_{i^*,u}) + d_G(c_{i^*,u},v) \le (1+2\epsilon)d_G(u,v)$.

*Case 2.2:* If $c_{i^*,u}$ is a rich center, then we know that $c_{i^*,u}$ is of distance at most $(\alpha+1)n/h_{i^*}$ from $u$, which can be bounded as $d_G(c_{i^*,u},u) \le (\alpha+1)n/h_{i^*} \le (\epsilon\alpha+\epsilon)d_G(u,v)$ using $n/h_{i^*} \le \epsilon d_G(u,v)$. It follows that

$$\begin{aligned} d_G(c_{i^*,u},v) &\le d_G(c_{i^*,u},u) + d_G(u,v) \\ &\le (1+\epsilon\alpha+\epsilon)d_G(u,v) \\ &= \left(1+\epsilon+\frac{(1+\epsilon)^2}{1-\epsilon}\right) d_G(u,v) \end{aligned}$$

using $\alpha = (1+\epsilon)^2/(\epsilon-\epsilon^2)$. Note that our algorithm maintains an ES-tree rooted at $c_{i^*,u}$ of depth $\tau = 2\alpha n/h_{i^*}$. Using $n/h_{i^*} > \epsilon d_G(u,v)/(1+\epsilon)$ and $\alpha = (1+\epsilon)^2/(\epsilon-\epsilon^2)$, we have

$$\begin{aligned} \tau > 2\frac{1+\epsilon}{1-\epsilon} d_G(u,v) &= \left(1+\epsilon+\frac{(1+\epsilon)^2}{1-\epsilon}\right) d_G(u,v) \\ &\ge d_G(c_{i^*,u},v)\,. \end{aligned}$$

The equation $2(1+\epsilon)/(1-\epsilon) = 1+\epsilon+(1+\epsilon)^2/(1-\epsilon)$ can easily be verified by multiplying $(1-\epsilon)/(1+\epsilon)$ on both sides. Thus, this ES-tree contains $v$. It follows that

$$\begin{aligned} \hat{d}(u,v) &\le d_G(u,c_{i^*,u}) + d_G(c_{i^*,u},v) \\ &\le (1+2\epsilon+2\epsilon\alpha)d_G(u,v) \\ &= (1+2\epsilon+2(1+\epsilon)^2/(1-\epsilon))d_G(u,v) \\ &\le (1+2\epsilon+2(1+7\epsilon))d_G(u,v) \le (3+16\epsilon)d_G(u,v) \end{aligned}$$

Note that the inequality $(1+\epsilon)^2/(1-\epsilon) \le 1+7\epsilon$ follows from $\epsilon \le 1/2$. □

We conclude the discussion of the $(3+\epsilon)$-approximate decremental APSP algorithm by restating Theorem 1.2.

THEOREM 1.2. ($(3+\epsilon)$-APPROXIMATION FOR APSP) *For any constant $0 < \epsilon < 1$, there is a $(3+\epsilon)$-approximation algorithm for decremental APSP with $O(\log\log n)$ worst-case query time and expected total update time of $\widetilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$.*

*Proof.* The correctness of the algorithm follows immediately from Claim 5.1. As usual, we run the algorithm described above with $\epsilon' = \epsilon/16$, to obtain a $(1+\epsilon)$-approximation (instead of the $(1+16\epsilon)$-approximation of Claim 5.1).

To prove the running time bound, remember that for every integer $i \ge 0$ such that $h_i \ge h^* = (m/n^{0.8})^{2/3}$, we maintain a center cover data structure $\mathcal{C}_i$ with parameters $\alpha = (1+\epsilon)^2/(\epsilon-\epsilon^2)$ and $h_i = (1+\epsilon)^i$. Maintaining $\mathcal{C}_i$ takes time $\widetilde{O}(\alpha mn/\sqrt{h_i})$ (see Section 1.2), which is $\widetilde{O}(mn/\sqrt{h^*})$ as $h_i \ge h^*$ and we assume that $\epsilon$ is a constant. As the algorithm uses only a logarithmic number of center cover data structures, they can all be maintained in time $\widetilde{O}(mn/\sqrt{h^*})$, which is $\widetilde{O}(m^{2/3}n^{3.8/3})$ by our choice of $h^*$. We also run the $(1+\epsilon)$-approximate decremental SSSP algorithm of Theorem 4.17 for $\widetilde{O}(h^*)$ randomly selected centers. This takes time $\widetilde{O}(h^*n^{1.8+O(1/\sqrt{\log n})})$ which is $\widetilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$ by our choice of $h^*$. This dominates the time needed for maintaining the center cover data structures. □

## 6 Conclusion

We presented a decremental $(1+\epsilon)$-approximate SSSP algorithm for sparse undirected unweighted graphs that is faster than the $O(mn)$ algorithm of Even and Shiloach [14] using a new center cover data structure and extending our previous lazy-update ES-tree [17]. By sparsifying graphs with the Thorup-Zwick emulator [24, 25] and extending our monotone ES-tree technique from [16], we can also use this new algorithm on dense graphs and ultimately obtain subquadratic total update time.

One direction of improving our SSSP result would be to improve the running time of our $\widetilde{O}(n^{1.8})$ algorithm for very sparse graphs with $m = O(n)$. We are not aware of any lower bound for this problem. Another direction of improving our result would be to extend it to weighted graphs. Similar to the unweighted case, decremental SSSP can be maintained in $\widetilde{O}(mn\log W)$ total

update time on weighted graphs by using Bernstein's rounding technique [6, 7] to modify the edge weights in such a way that we only have to maintain ES-trees up to depth $O(n)$ (instead of $nW$). However, unlike the unweighted case, we cannot replace the ES-tree by our lazy-update ES-tree to get a better running time. This is because the analysis of our lazy-update ES-tree relies on the following property of unweighted graphs: if the distance from the source increases for some node $v$ by $\delta$ it also has to increase by at least $\delta - 1$ for some neighbor of $v$.

For decremental APSP, it would be interesting to get a trade-off between total update time and approximation guarantee on *weighted* graphs. Currently we can achieve this for unweighted graphs using the algorithm of Bernstein and Roditty [8] ($(2k-1+\epsilon)$-approximation in $\widetilde{O}(n^{2+1/k+o(1)})$ time) and the algorithm in this paper ($(1+\epsilon, O(1/\epsilon)^k)$-approximation in $\widetilde{O}(n^{2+1/k})$ time). Both algorithms use a similar approach where we run some $\widetilde{O}(mn)$-time algorithm on a sparse emulator that approximates the distances of the original graph. Since we can also maintain such an emulator on weighted graphs, it is possible that the same approach will work on weighted graphs.

It would also be interesting to improve the running time of our $(3+\epsilon)$-approximation $\widetilde{O}(m^{2/3}n^{3.8/3})$-time algorithm for APSP. One direction for doing this is to relax the approximation guarantee for the sake of a better running time[13]. A more challenging direction is to keep the approximation ratio the same and improve the running time further.

**Acknowledgements.** We thank the reviewers of SODA 2014 for their helpful comments.

## References

[1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999. Announced at SODA, 1996.

[2] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991. Announced at SODA, 1990.

[3] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. On-line computation of minimal and maximal length paths. *Theoretical Computer Science*, 95(2):245–261, 1992.

[4] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *SODA*, pages 394–403, 2003.

[5] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. *Journal of Algorithms*, 62(2):74–92, 2007. Announced at STOC, 2002.

[6] A. Bernstein. Fully dynamic $(2+\epsilon)$ approximate all-pairs shortest paths with fast query and close to linear update time. In *FOCS*, pages 693–702, 2009.

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

[8] A. Bernstein and L. Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *SODA*, pages 1355–1365, 2011.

[9] C. Demetrescu and G. F. Italiano. Improved bounds and new trade-offs for dynamic all pairs shortest paths. In *ICALP*, pages 633–643, 2002.

[10] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, 51(6):968–992, 2004. Announced at STOC, 2003.

[11] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *Journal of Computer and System Sciences*, 72(5):813–837, 2006. Announced at FOCS, 2001.

[12] D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000. Announced at FOCS, 1996.

[13] M. Elkin. Computing almost shortest paths. *ACM Transactions on Algorithms*, 1(2):283–323, 2005. Announced at PODC, 2001.

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

[15] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006. Announced at FOCS, 2001.

[16] M. Henzinger, S. Krinninger, and D. Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the $O(mn)$ barrier and derandomization. In *FOCS*, 2013.

[17] M. Henzinger, S. Krinninger, and D. Nanongkai. Sublinear-time maintenance of breadth-first spanning tree in partially dynamic networks. In *ICALP*, 2013.

[18] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. Announced at STOC, 1994.

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

[20] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path

[13] One possible approach is to try running our $(3+\epsilon)$-approximation algorithm on the Thorup-Zwick emulator with some ES-trees replaced by monotone ES-trees. We expect this approach to at best have an approximation guarantee of $O(2^k)$ and a total update time of $\widetilde{O}(n^{2-\delta+1/k} + mn^{1/k})$ for some small constant $0 < \delta \leq 0.2/3$. It would be more interesting to get an $O(k)$-approximation guarantee with a similar total update time.

algorithms. In *COCOON*, pages 268–277, 2001.

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

[22] L. Roditty and U. Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM Journal on Computing*, 41(3):670–683, 2012. Announced at FOCS, 2004.

[23] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT*, pages 384–396, 2004.

[24] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):74–92, 2005. Announced at STOC, 2001.

[25] M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *SODA*, pages 802–809, 2006.

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

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

## A Appendix

To make the paper self-contained we provide in the following the pseudocode of the monotone ES-tree algorithm (Algorithm 3) and its running time analysis (Lemma A.1). More details can be found in [16].

LEMMA A.1. *The monotone ES-tree algorithm on $H$ has a total update time of $O((\tau+\gamma)|E'|\log n + \#up)$. Here $\#up$ is the total number of updates in $H$ and $E' \subseteq E(H)$ is the set of edges incident to nodes that are contained in the monotone ES-tree at some time (i.e., all nodes $v$ for which $l(v) \neq \infty$ at some time).*

*Proof.* Besides constant time for each update, all costs can be charged to level increases of nodes. In particular, the algorithm incurs a cost of $O(\deg(v)\log n)$ every time the level of the node $v$ increases, where $\deg(v)$ is the degree of $v$ at the time of the level increase. Let $V'$ be the set of all nodes ever contained in the monotone ES-tree, i.e., all nodes $v$ for which $l(v) \neq \infty$ at some point in time. Let $E'$ be the set of all edges ever incident to a node in $V'$. For every node $v \in V'$ we define the dynamic degree $\deg'(v)$ to be $\deg'(v) = |\{u \in V \mid (u,v) \in E'\}|$. Since $\deg(v) \leq \deg'(v)$ for every node $v \in V'$, the work charged to $v$ for each of its level increases is $O(\deg'(v)\log n)$. As the maximum finite level is bounded by $(1+\epsilon)\tau + \epsilon\gamma = O(\tau+\gamma)$, the total running time of the ES-tree algorithm is proportional to $\sum_{v\in V'}(\tau+\gamma)\deg'(v)\log n = (\tau+\gamma)\log n \sum_{v\in V'}\deg'(v) \leq 2(\tau+\gamma)|E'|\log n$. □

**Algorithm 3:** Monotone Even-Shiloach tree algorithm

```
// After updating an edge (u,v), the corresponding update procedure has to be executed for both (u,v) and (v,u).
// Internal data structures
N(x): for every node x a heap N(x) whose intended use is to store for every neighbor y of x in the current graph the value of l(y) + w(x,y) where w(x,y) is the weight of the edge (x,y) in the current graph
H: global heap used to determine order in reconnection procedure

insert(u, v)
    if l(v) + w(u,v) < l(u) then
        Make v the parent of u in the tree
        Update key of v in heap N(u) of u to l(v) + w(u,v)

increase(u, v, w)
    // Increase weight of edge (u,v) to w
    if (u,v) is a tree edge then
        Remove tree edge (u,v)
        Update key of v in heap N(u) of u
        Put u into heap H with key l(u)
        reconnect()

delete(u, v)
    increase(u, v, ∞)

reconnect(u, v)
    while heap H is not empty do
        Take node y with minimum key l(y) from heap H (and remove y from H)
        l'(y) = min_z(l(z) + w(y,z)) (can be retrieved from the heap N(y) of y)
        if l(y) ≥ l'(y) then
            z = arg min_z(l(z) + w(y,z)) (can be retrieved from the heap N(y) of y)
            Make z the parent of y in the tree
        else
            l(y) ← l'(y)
            if l(y) > (1+ε)τ + εγ then
                l(y) ← ∞
            else
                Put y into heap H with key l(y)
            for every edge (x,y) do
                update key of y in heap N(x) to l(y) + w(x,y)
            for every child x of y in the tree do
                Remove tree edge (x,y)
                Put x into heap H with key l(x)
```