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

Monika Henzinger† Sebastian Krinninger† Danupon Nanongkai‡

Abstract

We study dynamic $(1+\epsilon)$-approximation algorithms for the all-pairs shortest paths problem in unweighted undirected n-node m-edge graphs under edge deletions. The fastest algorithm for this problem is a randomized algorithm with a total update time of $\tilde{O}(mn/\epsilon)$ and constant query time by Roditty and Zwick [FOCS 2004]. The fastest deterministic algorithm is from a 1981 paper by Even and Shiloach [JACM 1981]; it has a total update time of $O(mn^2)$ and constant query time. We improve these results as follows:

(1) We present an algorithm with a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time that has an additive error of 2 in addition to the $1+\epsilon$ multiplicative error. This beats the previous $\tilde{O}(mn/\epsilon)$ time when $m = \Omega(n^{3/2})$. Note that the additive error is *unavoidable* since, even in the *static* case, an $O(n^{3-\delta})$-time (a so-called *truly subcubic*) combinatorial algorithm with $1+\epsilon$ multiplicative error cannot have an additive error less than $2-\epsilon$, unless we make a major breakthrough for Boolean matrix multiplication [Dor et al. FOCS 1996] and many other long-standing problems [Vassilevska Williams and Williams FOCS 2010].

The algorithm can also be turned into a $(2+\epsilon)$-approximation algorithm (without an additive error) with the same time guarantees, improving the recent $(3+\epsilon)$-approximation algorithm with $\tilde{O}(n^{5/2+O(\sqrt{\log(1/\epsilon)/\log n})})$ running time of Bernstein and Roditty [SODA 2011] in terms of both approximation and time guarantees.

(2) We present a deterministic algorithm with a total update time of $\tilde{O}(mn/\epsilon)$ and a query time of $O(\log\log n)$. The algorithm has a multiplicative error of $1+\epsilon$ and gives the first improved deterministic algorithm since 1981. It also answers an open

*The definite version of this article is published as: Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *SIAM Journal on Computing* (forthcoming). A preliminary version was presented at the *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS 2013)*.

†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, the University of Vienna (IK I049-N), and a Google Faculty Research Award. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506 and from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532.

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

question raised by Bernstein [STOC 2013]. The deterministic algorithm can be turned into a deterministic fully dynamic $(1+\epsilon)$-approximation with an amortized update time of $\tilde{O}(mn/(\epsilon t))$ and a query time of $\tilde{O}(t)$ for every $t \leq \sqrt{n}$.

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

Contents

1 Introduction

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

1.1 The Problem

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

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

We use the term *single-source shortest paths* (SSSP) to refer to the special case where the distance query can be done only when $x = s$ for a prespecified *source node* s. The efficiency is judged by two parameters: *query time*, denoting the time needed to answer *each* distance query, and *total update time*, denoting the time needed to process *all* edge deletions. The running time will be in terms of n, the number of nodes in the graph, and m, the number of edges *before* any deletion. We use $\tilde{O}$-notation to hide an $O(\text{poly} \log n)$ term. When it is clear from the context, we use "time" instead of "total update time," and, unless stated otherwise, the query time is $O(1)$. One of the main focuses of this problem in the literature, which is also the goal in this paper, is to *optimize the total update time* while keeping the query time and *approximation guarantees* small. We say that an algorithm provides an (α, β)-*approximation* if the distance query on nodes x and y on the current graph G returns an estimate $\delta(x, y)$ such that $d_G(x, y) \leq \delta(x, y) \leq \alpha d_G(x, y) + \beta$. We call α and β *multiplicative* and *additive errors*, respectively. We are particularly interested in the case where $\alpha = 1 + \epsilon$, for an arbitrarily small constant $\epsilon > 0$, β is a small constant, and the query time is constant or near-constant.

Previous Results. Prior to our work, the best total update time for *deterministic* decremental APSP algorithms was $\tilde{O}(mn^2)$ by one of the earliest papers in the area from 1981 by Even and Shiloach [ES81]. The fastest *exact randomized* algorithms are the $\tilde{O}(n^3)$-time algorithms by Demetrescu and Italiano [DI06] and Baswana, Hariharan, and Sen [BHS07]. The fastest *approximation* algorithm is the $\tilde{O}(mn)$-time $(1 + \epsilon, 0)$-approximation algorithm by Roditty and Zwick [RZ12]. If we insist on an $O(n^{3-\delta})$ running time, for some constant $\delta > 0$, Bernstein and Roditty [BR11] obtain an $\tilde{O}(n^{2+1/k+O(1/\sqrt{\log n})})$-time $(2k - 1 + \epsilon, 0)$-approximation algorithm, for any integer $k \geq 2$, which gives, e.g., a $(3 + \epsilon, 0)$-approximation

Reference	Total Running Time	Approximation	Deterministic?
[ES81]	$\tilde{O}(mn^2)$	Exact	Yes
This paper	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	Yes
[DI06, BHS07]	$\tilde{O}(n^3)$	Exact	No
[RZ12]	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	No
This paper	$\tilde{O}(n^{5/2}/\epsilon)$	$(1+\epsilon, 2)$	No
[BR11]	$\tilde{O}(n^{5/2+\sqrt{\log(6/\epsilon)}/\sqrt{\log n}})$	$(3+\epsilon, 0)$	No
This paper	$\tilde{O}(n^{5/2}/\epsilon)$	$(2+\epsilon, 0)$	No

Table 1: Comparisons between our and previous algorithms that are closely related. For details of these and other results see Section 1.4. All algorithms, except our deterministic algorithm, have $O(1)$ query time. Our deterministic algorithm has $O(\log \log n)$ query time.

guarantee in $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ time. All these algorithms have an $O(1)$ worst-case query time. See Section 1.4 for more detail and other related results.

1.2 Our Results

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

- The total running time of deterministic algorithms is improved from Even and Shiloach's $\tilde{O}(mn^2)$ to $\tilde{O}(mn)$ (at the cost of $(1+\epsilon, 0)$-approximation and $O(\log \log n)$ query time). This is the first improvement since 1981.
- For $m = \omega(n^{3/2})$, the total running time is improved from Roditty and Zwick's $\tilde{O}(mn/\epsilon)$ to $\tilde{O}(n^{5/2}/\epsilon)$, at the cost of an additive error of 2, which appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be treated as a multiplicative error of $O(\epsilon)$) and is *unavoidable* (as discussed below).
- Our $(2+\epsilon, 0)$-approximation algorithm improves the algorithm of Bernstein and Roditty in terms of both total update time and approximation guarantee. The multiplicative error of $2+\epsilon$ is essentially the best we can hope for, if we do not want any additive error.

To obtain these algorithms, we present two novel techniques, called *moving Even–Shiloach tree* and *monotone Even–Shiloach tree*, based on a classic technique of Even and Shiloach [ES81]. These techniques are reviewed in Section 1.3.

Improved Deterministic Algorithm. In 1981, Even and Shiloach [ES81] presented a deterministic decremental SSSP algorithm for undirected, unweighted graphs with a total update time of $O(mn)$ over all deletions. By running this algorithm from n different nodes, we get an $O(mn^2)$-time decremental algorithm for APSP. No progress on deterministic decremental APSP has been made since then. Our algorithm achieves the first improvement

over this algorithm, at the cost of a $(1+\epsilon, 0)$-approximation guarantee and $O(\log \log n)$ query time. (Note that our algorithm is also faster than the current fastest randomized algorithm [RZ12] by a $\log n$ factor.) Our deterministic algorithm also answers a question recently raised by Bernstein [Ber13] which asks for a deterministic algorithm with a total update time of $\tilde{O}(mn/\epsilon)$. As pointed out in [Ber13] and several other places, this question is important due to the fact that deterministic algorithms can deal with an *adaptive offline adversary* (the strongest adversary model in online computation [BEY98, BDBK+94]), while the randomized algorithms developed so far assume an *oblivious adversary* (the weakest adversary model) where the order of edge deletions must be fixed before an algorithm makes random choices. Our deterministic algorithm answers exactly this question. Using known reductions, we also obtain a deterministic fully dynamic $(1+\epsilon)$-approximation with an amortized running time of $\tilde{O}(mn/(\epsilon t))$ per update and a query time of $\tilde{O}(t)$ for every $t \leq n$.

Improved Randomized Algorithm. Our aim is to improve the $\tilde{O}(mn)$ running time of Roditty and Zwick [RZ12] to so-called *truly subcubic time*, i.e., $O(n^{3-\delta})$ time for some constant $\delta > 0$, a running time that is highly sought after in many problems (e.g., [VWW10, VWY09, RT13]). Note, however, that this improvement has to come at the cost of worse approximation.

Fact 1.1 ([DHZ00, VWW10])**.** *For any $\alpha \geq 1$ and $\beta \geq 0$ such that $2\alpha + \beta < 4$, there is* ***no*** *combinatorial (α, β)-approximation algorithm, not even a static one, for APSP on unweighted undirected graphs that is truly subcubic, unless we make a major breakthrough on many long-standing open problems, such as a combinatorial Boolean matrix multiplication and triangle detection.*

This fact is due to the reductions of Dor, Halperin, and Zwick [DHZ00] and Vassilevska Williams and Williams [VWW10] (see Appendix A for a proof sketch). (Roditty and Zwick [RZ11] also showed a similar fact for decremental exact SSSP. For the weighted case, lower bounds can be obtained even for noncombinatorial algorithms by assuming the hardness of APSP computation [RZ11, AVW14].) Very recently (after the preliminary version of this paper appeared), Henzinger et al. [HKN+15] showed that Fact 1.1 holds even for *noncombinatorial* algorithms assuming that there is no truly subcubic-time algorithm for a problem called *online Boolean matrix-vector multiplication*. Henzinger et al. [HKN+15] argue that refuting this assumption will imply the same breakthrough as mentioned in Fact 1.1 if the term "combinatorial algorithm" (which is not a well-defined term) is interpreted in a certain way (in particular if it is interpreted as a "Strassen-like algorithm" as defined in [BDH+12], which captures all known fast matrix multiplication algorithms). Thus, the best approximation guarantee we can expect from truly subcubic algorithms is, e.g., a multiplicative or additive error of at least 2. Our algorithms achieve essentially these *best approximation guarantees*: in $\tilde{O}(n^{5/2}/\epsilon)$ time, we get a $(1+\epsilon, 2)$-approximation, and, if we do not want any additive error, we can get a $(2+\epsilon, 0)$-approximation (see Theorem 3.22 and Corollary 3.23 for the precise statements of these results).[1] We note that, prior to our work, Bernstein and Roditty's algorithm [BR11] could achieve, e.g., a $(3+\epsilon, 0)$-approximation guarantee in $\tilde{O}(n^{5/2+O(\sqrt{1/\log n})})$ time. This result is improved by our $(2+\epsilon, 0)$-approximation

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

algorithm in terms of both time and approximation guarantees and is far worse than our $(1+\epsilon, 2)$-approximation guarantee, especially when the distance is large. Also note that the running time of our $(1+\epsilon, 2)$-approximation algorithm improves the $\tilde{O}(mn)$ one of Roditty and Zwick [RZ12] when $m = \omega(n^{3/2})$, except that our algorithm gives an additive error of 2 which is unavoidable and appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be counted as a multiplicative error of $O(\epsilon)$).

1.3 Techniques

Our results build on two previous algorithms. The first algorithm is the classic SSSP algorithm of Even and Shiloach [ES81] (with the more general analysis of King [Kin99]), which we will refer to as the *Even–Shiloach tree*. The second algorithm is the $(1+\epsilon, 0)$-approximation APSP algorithm of Roditty and Zwick [RZ12]. We actually view the algorithm of Roditty and Zwick as a *framework* which runs several Even–Shiloach trees and maintains some properties while edges are deleted. We would like to alter the Roditty–Zwick framework but doing so usually makes it hard to bound the cost of maintaining Even–Shiloach trees (as we will discuss later). Our main technical contribution is the development of new variations of the Even–Shiloach tree, called *moving Even–Shiloach tree* and *monotone Even–Shiloach tree*, which are suitable for our modified Roditty–Zwick frameworks. Since there are many other algorithms that run Even–Shiloach trees as subroutines, it might be possible that other algorithms will benefit from our new Even–Shiloach trees as well.

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

1.3.1 Monotone Even–Shiloach Tree for Improved Randomized Algorithms

The high-level idea of our randomized algorithm is to run an existing decremental algorithm of Roditty and Zwick [RZ12] on a sparse *weighted* graph that approximates the distances in the original graph, usually referred to as an *emulator* (see Section 3.1 for more detail). This approach is commonly used in the static setting (e.g., [ACI+99, DHZ00, Elk05, EP04, ABC+98, Coh98, CZ01, TZ05, Zwi02]), and it was recently used for the first time in the decremental setting by Bernstein and Roditty [BR11]. As pointed out by Bernstein and Roditty, while it is a simple task to run an existing APSP algorithm on an emulator in the static setting, doing so in the decremental setting is not easy since it will destroy the "decrementality" of the setting: when an edge in the original graph is deleted, we might

have to *insert* an edge into the emulator. Thus, we cannot run decremental algorithms on an arbitrary emulator, because from the perspective of this emulator, we are not in a decremental setting.

Bernstein and Roditty manage to get around this problem by constructing an emulator with a special property.[2] Roughly speaking, they show that their emulator guarantees that *the distance between any two nodes changes* $\tilde{O}(n)$ *times*. Based on this simple property, they show that the $(2k-1,0)$-approximation algorithm of Roditty and Zwick [RZ12] can be run on their emulator with a small running time. However, they *cannot* run the $(1+\epsilon,0)$-approximation algorithm of Roditty and Zwick on their emulator. The main reason is that this algorithm relies on a more general property of a graph under deletions: for any R between 1 and n, the distance between any two nodes changes at most R times *before it exceeds* R (i.e., it changes from 1 to R). They suggested finding an emulator with this more general property as a future research direction.

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

Our modification to the Even–Shiloach tree is as follows. Recall that the Even–Shiloach tree can maintain the distances between a specific node s and all other nodes, up to R, in $O(mR)$ total update time under edge deletions. This is because, for any node v, it has to do work $O(\deg(v))$ (the degree of v) only when the distance between s and v changes, which will happen at most R times (from 1 to R) in the decremental model. Thus, the total work on each node v will be $O(R\deg(v))$ which sums to $O(mR)$ in total. This algorithm does not perform well when there are edge insertions: one edge insertion could cause a *decrease* in the distance between s and v by as much as $\Omega(R)$, causing an additional $\Omega(R)$ distance changes. The idea of our monotone Even–Shiloach tree is extremely simple: *ignore distance decreases*! It is easy to show that the total update time of our algorithm remains the same $O(mR)$ as the classic one. The hard part is proving that it gives a good approximation when run on an emulator. This is because it does not maintain the *exact* distances on an emulator anymore. So, even when the emulator gives a good approximate distance on the original graph, our monotone Even–Shiloach tree might not. Our monotone Even–Shiloach tree does not give any guarantee for the distances in the emulator, but we can show that it still approximates the distances in the original graph. Of course, this will *not* work on any emulator; but we can show that it works on a specific type of emulator that we call *locally persevering emulators*.[3] Roughly speaking, a locally persevering emulator is an emulator

[2] In fact, their emulator is basically identical to one used earlier by Bernstein [Ber09], which is in turn a modification of a spanner developed by Thorup and Zwick [TZ05, TZ06]. However, the properties they proved are entirely new.

[3] We remark that there are other emulators that can be maintained in the decremental setting; see, e.g., [TZ05, TZ06, RZ12, Ber09, BR11, AFI06, Elk11, BKS12]. We are the first to introduce the notion of locally persevering emulators and show that there is an emulator that has this property.

where, for any "nearby"[4] nodes u and v in the original graph, either

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

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

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

Proving the Approximation Guarantee of the Monotone Even–Shiloach Tree. To illustrate why our monotone Even–Shiloach tree gives a good approximation when run on a locally persevering emulator, we sketch a result that is weaker and simpler than our main results; we show how to $(3, 0)$-approximate distances from a particular node s to other nodes. This fact easily leads to a $(3+\epsilon, 0)$-approximation $\tilde{O}(n^{5/2}/\epsilon)$-time algorithm, which gives the same approximation guarantee as the algorithm of Bernstein and Roditty [BR11] and is slightly faster and reasonably simpler. To achieve this, we use the following emulator which is a simple modification of the emulator of Dor, Halperin, and Zwick [DHZ00]: Randomly select $\tilde{\Theta}(\sqrt{n})$ nodes. At any time, the emulator consists of all edges incident to nodes of degree at most $\sqrt{n}$ and edges from each random node c to every node v of distance at most 2 from c with weight equal to the distance between v and c. When the distance exceeds 2, the edge is deleted from the emulator. It can be shown that this emulator can be maintained in $\tilde{O}(mn^{1/2}) = \tilde{O}(n^{5/2})$ time under edge deletions. Moreover, it is a $(3, 0)$-emulator with high probability, since for every edge (u, v) either

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

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

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

[4]Note that the word "nearby" will be parameterized by a parameter τ in the formal definition. So, formally, we must use the term (α, β, τ)-locally persevering emulator where α and β are multiplicative and additive approximation factors, respectively. See Section 3.1 for detail.

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

It follows that this emulator is locally persevering.[5] Now we show that when we run the monotone Even–Shiloach tree on the above emulator, it gives $(3, 0)$-approximate distances between s and all other nodes. Recall that the monotone Even–Shiloach tree maintains a distance estimate, say $\ell(v)$, between s and every node v in the emulator.[6] For every node v, the value of $\ell(v)$ is regularly updated, except that when the degree of a node drops to $\sqrt{n}$ and the resulting insertion of an edge, say (u, v), decreases the distance between v and s in the emulator; in particular, $\ell(v) > \ell(u) + w(u, v)$, where $w(u, v)$ is the weight of edge (u, v). A usual way to modify the Even–Shiloach tree for dealing with such an insertion [BR11] is to decrease the value of $\ell(v)$ to $\ell(u) + w(u, v)$. Our monotone Even–Shiloach tree will *not* do this and keeps $\ell(v)$ unchanged. In this case, we say that the node v and the edge (u, v) *become stretched*. In general, an edge (u, v) is *stretched* if $\ell(v) > \ell(u) + w(u, v)$ or $\ell(u) > \ell(v) + w(u, v)$, and a node is stretched if it is incident to a stretched edge. Two observations that we will use are

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

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

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

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

thus, assuming that $\ell(u_1)$ satisfies the claim (by induction), $\ell(v)$ will satisfy the claim as well. To prove this, observe that if the edge (v, u_1) is contained in the emulator then we know that $\ell(v) \leq \ell(u_1) + 1$ (since v is not stretched), and we are done. Otherwise, by the fact that this emulator is locally persevering, we know that there is a path $\pi = \langle v, c, u_1\rangle$ of length at most 3 in the emulator, and it is persevering (see Item (ii')). By (O2), *edges in π are not stretched*. It follows that

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

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

In Section 3, we show how to refine the above argument to obtain a $(1+\epsilon, 2)$-approximation guarantee. The first refinement, which is simple, is extending the emulator above to a $(1+\epsilon, 2)$-emulator. This is done by adding edges from every random node c to all nodes in distance at most $1/\epsilon$ from c. The next refinement, which is the main one, is the formal definition of (α, β, τ)-locally persevering emulators for some parameters α, β, and τ, and extending the proof outlined above to show that the monotone Even–Shiloach tree on such an emulator will give an $(\alpha + \beta/\tau, \beta)$-approximate distance estimate. We finally show that our simple $(1 + \epsilon, 2)$-emulator is a $(1, 2, 1/\epsilon)$-locally persevering emulator.

[5] We note that we are being vague here. To be formal, we later define the notion of (α, β, τ)-locally persevering emulator in Definition 3.2, and the emulator we just defined will be $(3, 0, 1)$-locally persevering.

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

1.3.2 Moving Even–Shiloach Tree for Improved Deterministic Algorithms

Many distance-related algorithms in both dynamic and static settings use the following *randomized argument* as an important technique: if we select $\tilde{O}(h)$ nodes, called *centers*, uniformly at random, then every node will be at distance at most n/h from one of the centers with high probability [UY91, RZ12]. This even holds in the decremental setting (assuming an oblivious adversary). Like other algorithms, the Roditty–Zwick algorithm also heavily relies on this argument, which is the only reason it is randomized. Our goal is to derandomize this argument. Specifically, for several different values of h, the Roditty–Zwick framework selects $\tilde{O}(h)$ random centers and uses the randomized argument above to argue that every node in a connected component of size at least n/h is *covered* by a center in the sense that it will always be within distance at most n/h from at least one center; we call this set of centers a *center cover*. It also maintains an Even–Shiloach tree of depth $R = O(n/h)$ from these h centers, which takes a total update time of $\tilde{O}(mR)$ for each tree and thus $\tilde{O}(hmR) = \tilde{O}(mn)$ over all trees. To derandomize the above process, we have two constraints:

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

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

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

Algorithm 1.2. *As long as there is a node v in a "big" connected component (i.e., of size at least n/h) that is not covered by any center, make v a new center.*

Algorithm 1.2 clearly guarantees the first constraint. The second constraint follows from the fact that the distance between any two centers is more than n/h. Since understanding the proof for guaranteeing the second constraint is important for understanding our *charging argument* later, we sketch it here. Let us label the centers by numbers $j = 1, 2, \ldots, h$. For a center with number j, we let B^j be a "ball" of radius $n/(2h)$; i.e., B^j is a set of nodes in distance at most $n/(2h)$ from center number j. Observe that B^j and $B^{j'}$ are disjoint for distinct centers j and j' since the distance between these centers is more than n/h. Moreover, $|B^j| \geq n/(2h)$ since every center is in a big connected component. So, the number of balls (thus the number of centers) is at most $n/(n/(2h)) = 2h$. This guarantees the second constraint. Thus, we can guarantee both constraints in the static setting.

This, however, is not enough in the dynamic setting since *after* edge deletions, some nodes in big components might not be covered anymore, and if we keep repeating Algorithm 1.2, we might have to keep creating new centers to such an extent that the second constraint is violated. The key idea that we introduce to avoid this problem is to allow a center and the Even–Shiloach tree rooted at it to *move*. We call this a *moving Even–Shiloach tree* or *moving centers* data structure. Specifically, in the moving Even–Shiloach tree, we view a root (center) s *not* as a node, but as a *token* that can be placed on any node, and the task of the moving Even–Shiloach tree is to maintain the distance between the node on which the root is placed and all other nodes, up to distance R. We allow a *move operation* where we can move the root to a new node and the corresponding Even–Shiloach tree must be

adjusted accordingly. To illustrate the power of the move operation, consider the following simple modification of Algorithm 1.2. (Later, we also have to modify this algorithm due to other problems that we will discuss next.)

Algorithm 1.3. *As long as there is a node v in a big connected component that is not covered by any center, we make it a center as follows. If there is a center in a small connected component, we move this center to v; otherwise, we open a new center at v.*

Algorithm 1.3 reuses centers and Even–Shiloach trees in small connected components[7] without violating the first constraint since nodes in small connected components do not need to be covered. The second constraint can also be guaranteed by showing that $|B^j| \geq n/(2h)$ for all j when we open a new center. Thus, by using moving Even–Shiloach trees, we can guarantee the two constraints above. We are, however, *not done yet*. This is because our new move operation also incurs a cost! *The most nontrivial idea in our algorithm is a charging argument to bound this cost.* There are two types of cost. First, the *relocation cost*, which is the cost of constructing a new breadth-first search tree rooted at the new location of the center. This cost can be bounded by $O(m)$ since we can construct a breadth-first search tree by running the static $O(m)$-time algorithm. Thus, it will be enough to guarantee that we do not move Even–Shiloach trees more than $O(n)$ times. In fact, this is already guaranteed in Algorithm 1.3 since we will *never* move an Even–Shiloach tree back to a previous node. The second cost, which is *much harder* to bound, is the *additional maintenance cost*. Recall that we can bound the total update time of an Even–Shiloach tree by $O(mR)$ because of the fact that the distance between its root (center) and each other node changes at most R times before exceeding R, by increasing from 1 to R. However, when we move the root from, say, a node u to its neighbor v, the distance between the new root v and some node, say x, might be smaller than the previous distance from u to x. In other words, *the decrementality property is destroyed*. Fortunately, observe that the distance change will be *at most one* per node when we move a tree to a neighboring node. Using a standard argument, we can then conclude that *moving a tree between neighboring nodes costs an additional distance maintenance cost of $O(m)$*. This motivates us to define the notion of *moving distance* to measure how far we move the Even–Shiloach trees in total. We will be able to bound the maintenance cost by $O(mn)$ if we can show that the total moving distance (summing over all moving Even–Shiloach trees) is $O(n)$. Bounding the total moving distance by $O(n)$ while having only $O(h)$ Even–Shiloach trees is the most challenging part in obtaining our deterministic algorithm. We do it by using a careful charging argument. We sketch this argument here. For more intuition and detail, see Section 4.

Charging Argument for Bounding the Total Moving Distance. Recall that we denote the centers by numbers $j = 1, 2, \ldots, h$. We make a few modifications to Algorithm 1.3. The most important change is the introduction of the set C^j for each center j (which is the root of a moving Even–Shiloach tree). This will lead to a few other changes. The importance of C^j is that we will "charge" the moving cost of center j to nodes in C^j; in particular, we bound the total moving distance to be $O(n)$ by showing that the moving distance of center j can be bounded by $|C^j|$, and C^j and $C^{j'}$ are disjoint for distinct centers j and j'. The other

[7] We note the detail that we need a deterministic dynamic connectivity data structure [HK01, HLT01] to implement Algorithm 1.3. The additional cost incurred is negligible.

important changes are the definitions of "ball" and "small connected component" which will now depend on C^j.

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

These new definitions might not be intuitive, but they are crucial for the charging argument. We also have to modify Algorithm 1.3 in a counterintuitive way: the most important modification is that we have to give up the nice property that the distance between any two centers is more than $n/(2h)$ as in Algorithms 1.2 and 1.3. In fact, we will *always* move a center out of a small connected component, and we will move it *as little as possible*, even though the new location could be near other centers. In particular, consider the deletion of an edge (u, v). It can be shown that there is *at most one* center j that is in a small connected component (according to the new definition), and this center j must be in the same connected component as u or v. Suppose that such a center j exists, and it is in the same connected component as u, say X. Then we will move center j to v, which is just enough to move j out of component X (it is easy to see that v is the node outside of X that is nearest to j before the deletion). We will also update C^j by adding all nodes of X to C^j. This finishes the moving step, and it can be shown that there is no center in a small connected component now. Next, we make sure that every node is covered by opening a new center at nodes that are not covered, as in Algorithm 1.2. To conclude, our algorithm is as follows.

Algorithm 1.4. *Consider the deletion of an edge (u, v). Check whether there is a center j that is in a "small" connected component X (of size less than $(n/(2h)) - |C^j|$). If there is such a j (there will be at most one such j), move it out of X to a new node which is the unique node in $\{u, v\} \setminus X$. After moving, execute the static algorithm as in Algorithm 1.2.*

To see that the total moving distance is $O(n)$, observe that when we move a center j out of component X in Algorithm 1.4, we incur a moving distance of at most $|X|$ (since we can move j along a path in X). Thus, we can always bound the total moving distance of center j by $|C^j|$. We additionally show that C^j and $C^{j'}$ are disjoint for different centers j and j'. So, the total moving distance over all centers is at most $\sum_j |C^j| \leq n$. We also have to bound the number of centers. Since we give up the nice property that centers are far apart, we cannot use the same argument to show that the sets B^j are disjoint and big (i.e., $|B^j| \geq n/(2h)$), as in Algorithms 1.3 and 1.4. However, using C^j, we can still show something very similar: $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint for distinct j and j', and $|B^j \cup C^j| \geq n/(2h)$. Thus, we can still bound the number of centers by $O(h)$ as before.

1.4 Related Work

Dynamic APSP has a long history, with the first papers dating back to 1967 [LC67, Mur67][8]. It also has a tight connection with its *static* counterpart (where the graph does not change),

[8]The early papers [LC67, Mur67], however, were not able to beat the naive algorithm where we compute APSP from scratch after every change.

which is one of the most fundamental problems in computer science: On the one hand, we wish to devise a dynamic algorithm that beats the naive algorithm where we recompute shortest paths *from scratch* using static algorithms after every deletion. On the other hand, the best we can hope for is to match the total update time of decremental algorithms to the best running time of static algorithms. To understand the whole picture, let us first recall the current situation in the static setting. We will focus on combinatorial algorithms[9] since our and most previous decremental algorithms are combinatorial. Static APSP on unweighted undirected graphs can be solved in $O(mn)$ time by simply constructing a breadth-first search tree from every node. Interestingly, this algorithm is the fastest combinatorial algorithm for APSP (despite other fast noncombinatorial algorithms based on matrix multiplication). In fact, a faster combinatorial algorithm will be a *major breakthrough*, not just because computing shortest paths is a long-standing problem by itself, but also because it will imply faster algorithms for other long-standing problems, as stated in Fact 1.1.

The fact that the best static algorithm takes $O(mn)$ time means two things: First, the naive algorithm will take $O(m^2n)$ total update time. Second, the best total update time we can hope for is $O(mn)$. A result that is perhaps the first to beat the naive $O(m^2n)$-time algorithm is from 1981 by Even and Shiloach [ES81] for the special case of SSSP. Even and Shiloach actually studied decremental connectivity, but their main data structure gives an $O(mn)$ total update time with $O(1)$ query time for decremental SSSP; this implies a total update time of $O(mn^2)$ for decremental APSP. Roditty and Zwick [RZ11] later provided evidence that the $O(mn)$-time decremental unweighted SSSP algorithm of Even and Shiloach is the fastest possible by showing that this problem is at least as hard as several natural static problems such as Boolean matrix multiplication and the problem of finding all edges of a graph that are contained in triangles. For the incremental setting, Ausiello et al. [AIMS+91] presented an $\tilde{O}(n^3)$-time APSP algorithm on unweighted directed graphs. (An extension of this algorithm for graphs with small integer edge weights is given in [AIM+92].) After that, many efficient fully dynamic algorithms have been proposed (e.g., [HKR+97, Kin99, FR06, DI06, DI02]). Subsequently, Demetrescu and Italiano [DI04] achieved a major breakthrough for the fully dynamic case: they obtained a fully dynamic deterministic algorithm for the weighted directed APSP problem with an amortized time of $\tilde{O}(n^2)$ *per update*, implying a total update time of $\tilde{O}(mn^2)$ over all deletions in the decremental setting, the same running time as the algorithm of Even and Shiloach. (Thorup [Tho04] presented an improvement of this result.) An amortized update time of $\tilde{O}(n^2)$ is essentially optimal if the distance matrix is to be explicitly maintained, as done by the algorithm of Demetrescu and Italiano [DI04], since each update operation may change $\Omega(n^2)$ entries in the matrix. Even for unweighted, undirected graphs, no faster algorithm is known. Thus, the $O(mn^2)$ total update time of Even and Shiloach *remains the best* for deterministic decremental algorithms, even on undirected unweighted graphs and if approximation is allowed.

For the case of randomized algorithms, Demetrescu and Italiano [DI06] obtained an exact decremental algorithm on weighted directed graphs with $\tilde{O}(n^3)$ total update time[10] (if

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

[10] This algorithm actually works in a much more general setting where each edge weight can assume S different values. Note that the amortized time per update of this algorithm is $\tilde{O}(Sn)$, but this holds only when there are $\Omega(n^2)$ updates (see [DI06, Theorem 10]). Also note that the algorithm is randomized with one-sided error.

weight increments are not considered). Baswana, Hariharan, and Sen [BHS07] obtained an exact decremental algorithm on unweighted directed graphs with $\tilde{O}(n^3)$ total update time. They also obtained a $(1+\epsilon, 0)$-approximation algorithm with $\tilde{O}(m^{1/2}n^2)$ total update time. In [BHS03], they improved the running time further on undirected unweighted graphs, at the cost of a worse approximation guarantee: they obtained approximation guarantees of $(3, 0)$, $(5, 0)$, $(7, 0)$ in $\tilde{O}(mn^{10/9})$, $\tilde{O}(mn^{14/13})$, and $\tilde{O}(mn^{28/27})$ time, respectively. Roditty and Zwick [RZ12] presented two improved algorithms for unweighted, undirected graphs. The first was a $(1+\epsilon, 0)$-approximate decremental APSP algorithm with constant query time and a total update time of $\tilde{O}(mn)$. This algorithm remains the current fastest. The second algorithm achieves a worse approximation bound of $(2k-1, 0)$ for any $2 \leq k \leq \log n$, but has the advantage of requiring less space ($O(m + n^{1+1/k})$). By modifying the second algorithm to work on an emulator, Bernstein and Roditty [BR11] presented the first truly subcubic algorithm which gives a $(2k-1+\epsilon, 0)$-approximation and has a total update time of $\tilde{O}(n^{1+1/k+O(1/\sqrt{\log n})})$. They also presented a $(1+\epsilon, 0)$-approximation $\tilde{O}(n^{2+O(1/\sqrt{\log n})})$-time algorithm for SSSP, which is the first improvement since the algorithm of Even and Shiloach. Very recently, Bernstein [Ber13] presented a $(1+\epsilon, 0)$-approximation $\tilde{O}(mn \log W)$-time algorithm for the directed weighted case, where W is the ratio of the largest edge weight ever seen in the graph to the smallest such weight.

We note that the $(1+\epsilon, 0)$-approximation $\tilde{O}(mn)$-time algorithm of Roditty and Zwick matches the state of the art in the static setting; thus, it is essentially tight. However, by allowing additive error, this running time was improved in the static setting. For example, Dor, Halperin, and Zwick [DHZ00], extending the approach of Aingworth et al. [ACI+99], presented a $(1, 2)$-approximation for APSP in unweighted undirected graphs with a running time of $O(\min\{n^{3/2}m^{1/2}, n^{7/3}\})$. Elkin [Elk05] presented an algorithm for unweighted undirected graphs with a running time of $O(mn^{\rho} + n^2\zeta)$ that approximates the distances with a multiplicative error of $1+\epsilon$ and an additive error that is a function of ζ, ρ, and ϵ. There is no decremental algorithm with additive error prior to our algorithm.

Subsequent Work. Independent of our work, Abraham and Chechik [AC13] developed a randomized $(1+\epsilon, 2)$-approximate decremental APSP algorithm with a total update time of $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ and constant query time. This result is very similar to one of ours, except that the running time in [AC13] is slightly more than $\tilde{O}(n^{5/2})$. After the preliminary version of this paper [HKN13a] appeared, we extended the randomized algorithm in this paper and obtained the following two algorithms for APSP [HKN14a]: (i) a $(1+\epsilon, 2(1+2/\epsilon)^{k-2})$-approximation with total time $\tilde{O}(n^{2+1/k}(37/\epsilon)^{k-1})$ for any $2 \leq k \leq \log n$ (improving the time in this paper with a higher additive error when $k \geq 3$), and (ii) a $(3+\epsilon)$-approximation with total time $\tilde{O}(m^{2/3}n^{3.8/3+O(1/\sqrt{\log n})})$ (it is faster than the algorithm in this paper for sparse graphs but causes more multiplicative error). These two algorithms heavily rely on the monotone Even–Shiloach tree introduced in this paper. In the same paper, the monotone Even–Shiloach tree was also used in combination with techniques in [HKN13b] to obtain the first subquadratic-time algorithm for approximate SSSP. Very recently, we obtained an almost linear total update time for $(1+\epsilon)$-approximate SSSP in weighted undirected graphs [HKN14b], where the monotone Even–Shiloach tree again played a central role. We also obtained the first improvement over Even–Shiloach's algorithm for single-source reachability and approximate single-source shortest paths on *directed* graphs [HKN14c].

2 Background

2.1 Basic Definitions

In the following we give some basic notation and definitions.

Definition 2.1 (Dynamic graph)**.** *A* dynamic graph $\mathcal{G}$ *is a sequence of graphs* $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ *that share a common set of nodes* V*. The set of edges of the graph* G_i *(for* $0 \leq i \leq k$*) is denoted by* $E(G_i)$*. The number of nodes of* $\mathcal{G}$ *is* $n = |V|$*, and the initial number of edges of* $\mathcal{G}$ *is* $m = |E(G_0)|$*. The set of edges ever contained in* $\mathcal{G}$ *up to time* t *(where* $0 \leq t \leq k$*) is* $E_t(\mathcal{G}) = \cup_{0 \leq i \leq t} E(G_i)$*. A dynamic weighted graph* $\mathcal{H}$ *is a sequence of weighted graphs* $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ *that share a common set of nodes* V*. For* $0 \leq i \leq k$ *and every edge* $(u, v) \in E(H_i)$*, the weight of* (u, v) *is given by* $w_i(u, v)$*.*

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

Definition 2.2 (Updates)**.** *For a dynamic graph* $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ *we say for an edge* (u, v) *that*

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

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

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

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

Definition 2.3 (Decremental graph)**.** *A* decremental graph $\mathcal{G}$ *is a dynamic graph* $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ *such that for every* $1 \leq i \leq k$ *there is exactly one edge deletion at time* i*. Note that* G_i *is the graph after the* i*-th edge deletion.*

By our definition decremental graphs are always unweighted. For a weighted version of this concept it would make sense to additionally allow edge weight increases. In a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ we necessarily have $k \leq m$ because every edge can be deleted only once.

For decremental shortest paths algorithms the total update time usually does *not* depend on the number of deletions k. This is the case because of the amortization argument typically used for these algorithms. For this reason, it will often suffice for our purposes to bound $\phi_k(\mathcal{G})$ or $|E_k(\mathcal{G})|$ by numbers that do not depend on k.

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

Definition 2.4 (Distance)**.** *The distance of a node x to a node y in a graph G is denoted by $d_G(x, y)$. If x and y are not connected in G, we set $d_G(x, y) = \infty$. In a weighted graph (H, w) the distance of x to y is denoted by $d_{H,w}(x, y)$.*

Definition 2.5. *An* (α, β)-approximate decremental all-pairs shortest paths (APSP) data structure *for a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ maintains, for all nodes x and y and all $0 \leq i \leq k$, an estimate $\delta_i(x, y)$ of the distance between x and y in G_i. After the i-th edge deletion (where $0 \leq i \leq k$), it provides the following operations:*

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

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

Similarly, we define a data structure for decremental single-source shortest paths (SSSP). We incorporate two special requirements in this definition. First, we are interested in SSSP data structures that only need to work up to a certain distance range[11] R^{d} from the source node which is specified by a parameter R^{d}. Second, we demand that the data structure tells us whenever a node leaves this distance range. The latter is a technical requirement that simplifies some of our proofs.

Definition 2.6. *An* (α, β)-approximate decremental single-source shortest paths (SSSP) data structure *with* source (or: root) node r *and* distance range parameter R^{d} *for a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ maintains, for every node x and all $0 \leq i \leq k$, an estimate $\delta_i(x, r) \in \{0, 1, \ldots, \lfloor \alpha R^d + \beta \rfloor, \infty\}$ of the distance between x and r in G_i. After the i-th edge deletion (where $0 \leq i \leq k$), it provides the following operations:*

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

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

[11] In this paper, there are two related parameters R^{d} (introduced here) representing the "distance range" of an SSSP data structure (e.g., the Even–Shiloach tree described in Section 2.2) and R^{c} (which will be introduced in Section 2.3) representing the "cover range" of the center cover data structure.

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

Definition 2.7 (Degree)**.** *We say that v* is a neighbor of *u if there is an edge (u, v) in G. The* degree *of a node u in the graph G, denoted by $\deg_G(u)$, is the number of neighbors of u in G. The* dynamic degree *of a node u in a dynamic graph $\mathcal{G}$ is $\deg_{\mathcal{G}}(u) = |\{(u, v) \mid (u, v) \in E_k(\mathcal{G})\}|$.*

Definition 2.8 (Paths)**.** *Let (H, w) be a weighted graph, and let π be a path in (H, w). The number of nodes on the path π is denoted by $|\pi|$, and the total weight of the path (i.e., the sum of the weights of its edges) is denoted by $w(\pi)$.*

Definition 2.9 (Connected component)**.** *For every graph G and every node x we denote by $\mathrm{Comp}_G(x)$ the connected component of x in G, i.e., the set of nodes that are connected to x in G.*

2.2 Decremental Shortest-Path Tree Data Structure (Even–Shiloach Tree)

The central data structure in dynamic shortest paths algorithms is the dynamic SSSP tree introduced by Even and Shiloach, in short *ES-tree*. Even and Shiloach [ES81] developed this data structure for undirected, unweighted graphs. Later on, Henzinger and King [HK95] observed that it can be adapted to work on directed graphs, and King [Kin99] gave a modification for directed, weighted graphs with positive integer edge weights. In the following we review some important properties of this data structure.

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

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

$$\ell(v) = \min_u \left(\ell(u) + w(u, v)\right) \tag{1}$$

where the minimum is over all current neighbors u of v. When we delete an edge incident to v, the value of $\ell(v)$ might change. If this happens, v *sends a message* to each of its neighbors to inform about this change, since the levels of these nodes might have to change as well. Every neighbor of v then updates its level accordingly, and if its level changes, it sends messages to its neighbors (including v), too. (See Figure 1 for an example.) An important point, which we will show soon, is that *we can implement the ES-tree in time proportional to the number of messages.* This means that when a node v's level is changed, we can bound

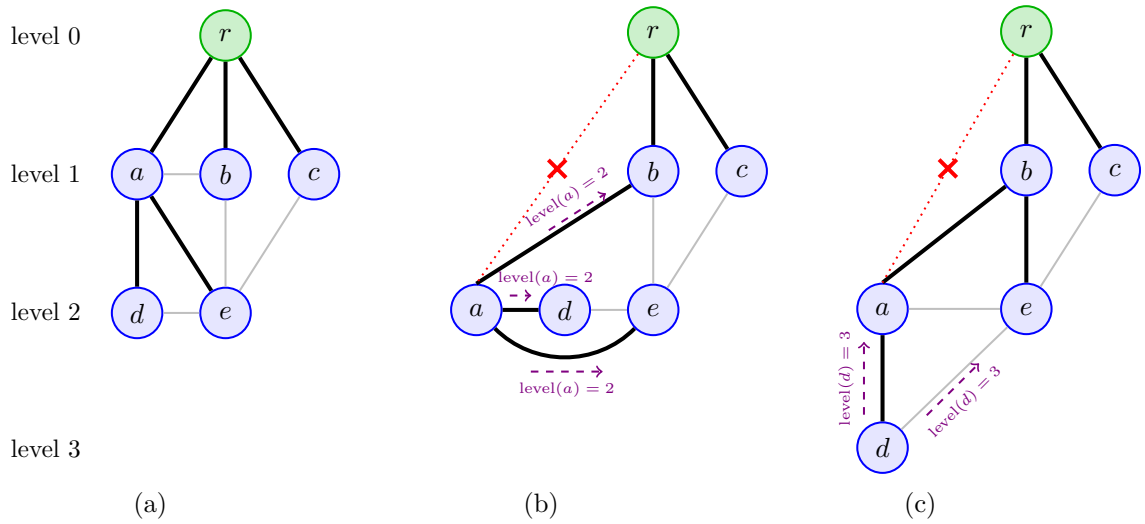


Figure 1: Example of the view of the ES-tree as nodes talking to each other. (a) The ES-tree before the edge deletion. (b) After deleting the edge (r, a), the level of the node a changes to 2. The node a sends a message to all neighbors to inform them about this change. (c) This causes the node d to change its level, and thus d sends a message to inform its neighbors. There are no other changes, so the new ES-tree is as in (c). Thus, there are 5 messages involved in constructing the new ES-tree, and Algorithm 1 shows that this process can be implemented in 5 time units.

the time we need to maintain the ES-tree by its current degree. Thus, the contribution of a node v to the running time to update the ES-tree after the i-th deletion or weight increase is $\deg_{G_i}(v)$ times its level change, i.e., $\min(R^{\mathrm{d}}, \ell_i(v)) - \min(R^{\mathrm{d}}, \ell_{i-1}(v))$ (the minimum is to avoid the case where $\ell_i(v) = \infty$). This intuitively leads to the following lemma.

Lemma 2.10 (King [Kin99])**.** *The ES-tree is an exact decremental SSSP data structure for shortest paths up to a given length R^d. It has constant query time, and in a decremental graph $\mathcal{G} = (G_i, w_i)_{0 \le i \le k}$ its total update time can be bounded by*

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

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

Recall that $\phi_k(\mathcal{G})$ is the total number of updates (deletions and weight increases). Note that when the graph is unweighted, only deletions are allowed. In this case, the $\phi_k(\mathcal{G})$ term can be ignored. Lemma 2.10 can be simplified by using two specific bounds. These bounds are in fact what we need later in this paper.

Corollary 2.11. *There is an exact decremental SSSP data structure for paths up to a given length R^d that has constant query time, and in a decremental graph $\mathcal{G} = (G_i, w_i)_{0 \le i \le k}$ with*

source node r its total update time can be bounded by

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

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

The first bound in Corollary 2.11 is because, for every node v, we can use $\deg_{G_i}(v) \leq \deg_{G_0}(v)$, and we can express the running time caused by v's level change in terms of its initial level ($\min(R^{\text{d}}, d_{G_0}(v,r))$ and its final level ($\min(R^{\text{d}}, d_{G_k}(v,r))$). We will need this bound in Section 4. The second bound follows easily from the first one and we will need it in Section 3.

Implementation. The pseudocode for achieving the above result can be found in Algorithm 1. (For simplicity we show an implementation using heaps, which causes an extra $\log n$ factor in the running time. King [Kin99] explains how to avoid heaps in order to improve the running time by a factor of $\log n$.) For every node x the ES-tree maintains a heap $N(x)$ that stores for every neighbor y of x in the current graph the value of $\ell(y) + w(x,y)$ where $w(x,y)$ is the weight of the edge (x,y) in the current graph. (Intuitively, $N(x)$ corresponds to the "knowledge" of x about its neighbors.) These data structures can be initialized in $\tilde{O}(m)$ time by running, for example, Dijkstra's algorithm (see procedure Initialize()).[12]

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

Procedure UpdateLevels() processes the nodes in Q in the order of their current level (see the while-loop starting on Line 13). In every iteration it will process y in Q with smallest $\ell(y)$ (as in Line 14). The lowest level that is possible for a node y is $\ell'(y) = \min_z(\ell(z)+w(y,z))$, the minimum of $\ell(z) + w(y,z)$ over all neighbors z of y in the current graph (following Equation (1)). Therefore every node y will repeatedly update its level to $\ell'(y)$ (unless its level already has this value); see Line 15. (An exception is when the level of a node x exceeds the desired depth R^{d}. In this case the level of x is set to ∞ and x will never be connected to the tree again. See Line 18.) If this updating rule leads to a level increase, the algorithm has to update the heap $N(x)$ of every neighbor x and put x to the heap Q (since the level of x might increase), as in the for-loop starting on Line 19 (this is equivalent to having y send a message to x in the informal description).

The running time analysis takes into account the level increases occurring in the ES-tree. It is based on the following observation: For every node x processed in the while-loop of the

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

Algorithm 1: ES-tree

// The ES-tree is formulated for weighted undirected graphs.

// Internal data structures:

- $N(u)$: for every node u a heap $N(u)$ whose intended use is to store for every neighbor v of u in the current graph the value of $\ell(v) + w(u, v)$, where $w(u, v)$ is the weight of the edge (u, v) in the current graph
- Q: global heap whose intended use is to store nodes whose levels might need to be updated

Procedure INITIALIZE()
  Compute shortest paths tree from r in (G_0, w_0) up to depth R^{d}
  foreach *node* u **do**
    Set $\ell(u) = d_{G_0}(u, r)$
    for *every edge* (u, v) **do** insert v into heap $N(u)$ of u with key $\ell(v) + w(u, v)$
Procedure DELETE(u, v)
  INCREASE(u, v, ∞)
Procedure INCREASE(u, v, $w(u, v)$)
  // Increase weight of edge (u, v) to $w(u, v)$
  Insert u and v into heap Q with keys $\ell(u)$ and $\ell(v)$, respectively
  Update key of v in heap $N(u)$ to $\ell(v) + w(u, v)$ and key of u in heap $N(v)$ to $\ell(u) + w(u, v)$
  UPDATELEVELS()
Procedure UPDATELEVELS()
  while *heap* Q *is not empty* **do**
    Take node y with minimum key $\ell(y)$ from heap Q and remove it from Q
    $\ell'(y) \leftarrow \min_z(\ell(z) + w(y, z))$
    // $\ell'(y)$ can be retrieved from the heap $N(y)$. $\arg\min_z(\ell(z) + w(y, z))$ is y's parent in the ES-tree
    if $\ell'(y) > \ell(y)$ **then**
      $\ell(y) \leftarrow \ell'(y)$
      if $\ell'(y) > R^d$ **then** $\ell(y) \leftarrow \infty$
      foreach *neighbor* x *of* y **do**
        update key of y in heap $N(x)$ to $\ell(y) + w(x, y)$
        insert x into heap Q with key $\ell(x)$ if Q does not already contain x

procedure UpdateLevels() in Algorithm 1, if the level of x increases, the algorithm has to spend time $O(\deg(x) \log n)$ updating the heaps $N(y)$ of all neighbors y of x and adding these neighbors to heap Q. If the level of x does not increase, the algorithm only has to spend time $O(\log n)$. In the second case the running time can be charged to one of the following events that causes x to be in Q: (1) a weight increase of some edge (x, y), and (2) a level increase of some neighbor of x. This leads to the result in Lemma 2.10.

2.3 The Framework of Roditty and Zwick

In the following we review the algorithm of Roditty and Zwick [RZ12] because its main ideas are the basis of our own algorithms. We will put their arguments in a certain structure that clarifies for which part of the algorithm we obtain improvements. Their algorithm is based on the following observation. Consider approximating the distance $d_G(x, y)$ for some pair of nodes x and y. For some $0 < \epsilon \leq 1$, we want a $(1 + O(\epsilon), 0)$-approximate value of $d_G(x, y)$. Assume that we know an integer p such that 2^p is a "distance guess" of $d_G(x, y)$, i.e.,

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

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

$$d_G(x, z) \leq \epsilon 2^p. \tag{3}$$

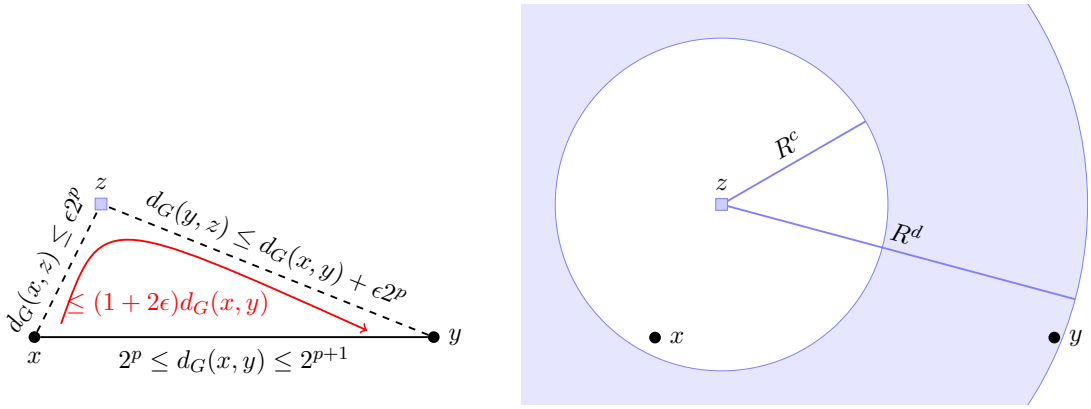
Then it follows that we can use $d_G(x, z) + d_G(z, y)$ as a $(1 + 2\epsilon)$-approximation of the true distance $d_G(x, y)$; this follows from applying the triangle inequality twice (also see Figure 2a):

$$\begin{aligned} d_G(x, y) &\leq d_G(x, z) + d_G(z, y) \\ &\leq d_G(x, z) + (d_G(z, x) + d_G(x, y)) \leq (1 + 2\epsilon) d_G(x, y)\,. \end{aligned} \tag{4}$$

Thus, under the assumption that for any x we only want to determine the distances from x to nodes y with $d_G(x, y)$ in the range from 2^p to 2^{p+1}, we only have to make sure that there is a node z that satisfies Equation (3); we call such node z a *center*. We will maintain a set U of nodes such that for every node x there is a node $z \in U$ that satisfies Equation (3). In fact, we only need this to be true for nodes x that are in a "big" connected component since if the connected component containing x is too small, then there is no node y that satisfies Equation (2). We call such U a *center cover*. The following definition states this more precisely.

Definition 2.12 (Center cover)**.** *Let U be a set of nodes of a graph G, and let R^c be a positive integer denoting the* cover range*. We say that a node x is* covered *by a node $c \in U$ in G if $d_G(x, c) \leq R^c$. We say that U is a* center cover of G with parameter R^c *if every node x that is in a connected component of size at least R^c is covered by some node $c \in U$ in G.*

One main component of Roditty and Zwick's framework as we describe it is the *center cover data structure*. This data structure maintains a center cover U as above. Furthermore, for every center $z \in U$, we will maintain the distance to every node y such that $d_G(z, y) \leq 2^{p+2}$. This will allow us to compute $d_G(x, z) + d_G(z, y)$ as an approximate value of $d_G(x, y)$ (as in Equation (4)). In general, we treat the number 2^{p+2} as another parameter of the data structure denoted by R^{d} (called distance range parameter). The values of R^{c} and R^{d} are typically closely related; in particular, $R^{\mathrm{c}} \leq R^{\mathrm{d}} = O(R^{\mathrm{c}})$. The center cover data structure is defined formally as follows (also see Figure 2b).



(a) (b)

Figure 2: (a) depicts Equations (2) to (4). (b) shows the cover range (small circle) and distance range (big circle) used by the center cover data structure (Definition 2.13).

Definition 2.13 (Center cover data structure)**.** *A* center cover data structure *with* cover range parameter R^{c} *and* distance range parameter R^{d} *for a decremental graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *maintains, for every* $0 \le i \le k$*, a set of* centers $C_i = \{1, 2, \ldots, l\}$ *and a set of nodes* $U_i = \{c_i^1, c_i^2, \ldots, c_i^l\}$ *such that* U_i *is a center cover of* G_i *with parameter* R^c*. For every center* $j \in C_i$ *and every* $0 \le i \le k$*, we call* $c_i^j \in U_i$ *the* location *of center* j *in* G_i*, and for every node* x *we say that* x *is covered by* j *if* x *is covered by* c_i^j *in* G_i*. After the* i*-th edge deletion (where* $0 \le i \le k$*), the data structure provides the following operations:*

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

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

As the update time of the data structure will depend on the number l of centers, the goal is to keep l as small as possible, preferably $l = \tilde{O}(n/R^{\mathrm{c}})$. As an example, consider the following *randomized* implementation of Roditty and Zwick [RZ12]: randomly pick a set U of $((n/R^{\mathrm{c}}) \operatorname{poly} \log n)$ nodes as the set of centers. It can be shown that, with high probability, this set will remain a center cover during all deletions. The `distance` and `findCenter` queries can be answered in $O(1)$ time by maintaining an ES-tree of depth mR^{d} for every center. The total time to maintain this data structure is thus $\tilde{O}(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. We typically set $R^{\mathrm{c}} = \Omega(R^{\mathrm{d}})$. In this case, the total time becomes $\tilde{O}(mn)$.

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

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

To see the approximation guarantee, let p^* be such that $2^{p^*} \leq d_{G_i}(x, y) \leq 2^{p^*+1}$. Observe that if $p = p^*$, then $d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is a $(1 + O(\epsilon), 0)$-approximate distance estimate (due to Equation (4)), and if $p \neq p^*$, then $d_{G_i}(z_p, x) + d_{G_i}(z_p, y) \geq d_{G_i}(x, y)$ (by the triangle inequality). Thus, $\min_p d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$ is a $(1+O(\epsilon), 0)$-approximate value of $d_{G_i}(x, y)$. The query time, which is the time to compute $\min_p d_{G_i}(z_p, x) + d_{G_i}(z_p, y)$, is $O(\log n)$.

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

Theorem 2.14 ([RZ12])**.** *Assume that for all parameters R^c and R^d such that $R^c \leq R^d$ there is a center cover data structure that has constant query time and a total update time of $T(R^c, R^d)$. Then, for every $\epsilon \leq 1$, there is a $(1 + \epsilon, 0)$-approximate decremental APSP data structure with $O(\log \log n)$ query time and a total update time of $\sum_p T(R^c_p, R^d_p)$ where $R^c_p = \epsilon 2^p$ and $R^d_p = 2^{p+2}$ (for $0 \leq p \leq \lfloor \log n \rfloor$).*

As shown before, Roditty and Zwick [RZ12] obtain a randomized center cover data structure with constant query time and a total update time of $\tilde{O}(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. By Theorem 2.14 they get a $(1 + \epsilon, 0)$-approximate decremental APSP data structure with a total update time of $\tilde{O}(mn/\epsilon)$ and a query time of $O(\log \log n)$. Note that the query time can sometimes be

reduced further to $O(1)$, and this is the case for their algorithm as well as our randomized algorithm in Section 3. This is essentially because there is a $(3,0)$-approximation randomized algorithm for APSP, which can be used to approximate p^* (we defer details to Lemma 3.19). To analyze the total update time of their data structure for k deletions, observe that

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

In Section 3 we show that we can maintain an *approximate* version of the center cover data structure in time $\tilde{O}(n^{5/2}R^{\mathrm{d}}/(\epsilon R^{\mathrm{c}}))$. Using this data structure, we will get a $(1+\epsilon,2)$-approximate decremental APSP data structure with a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time. In Section 4 we show how to maintain an exact *deterministic* center cover data structure with a total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. By Theorem 2.14 this immediately implies a deterministic $(1+\epsilon,0)$-approximate decremental APSP data structure with a total update time of $O(mn \log n)$ and a query time of $O(\log \log n)$.

3 $\tilde{O}(n^{5/2})$-Total Time $(1+\epsilon,2)$- and $(2+\epsilon,0)$-Approximation Algorithms

In this section, we present a data structure for maintaining APSP under edge deletions with multiplicative error $1+\epsilon$ and additive error 2 that has a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$. The data structure is correct with high probability. We also show a variant of this data structure with multiplicative error $2+\epsilon$ and no additive error. In doing this, we introduce the notion of a *persevering path* (see Definition 3.1) and a *locally persevering emulator* (Definition 3.2). In Section 3.1, we then present the locally persevering emulator that we will use to obtain our result. Then in Section 3.2 we explain our main technique, called the *monotone Even–Shiloach tree*, where we maintain the distances from a single node to all other nodes, up to some distance R^{d}, in a locally persevering emulator. (Recall that R^{d} is a parameter called the "distance range.") In Section 3.3 we show how approximate decremental SSSP helps in solving approximate decremental APSP. Finally, in Section 3.4, we show how to put the results in Sections 3.1 to 3.3 together to obtain the desired $(1+\epsilon,2)$- and $(2+\epsilon,0)$-approximate decremental APSP data structures.

Definition 3.1 (Persevering path)**.** *Let $\mathcal{H} = (H_i, w_i)_{0\le i\le k}$ be a dynamic weighted graph. We say that a path $\pi = \langle v_0, v_1, \ldots, v_\ell\rangle$ is* persevering up to time t *(where $t \le k$) if for all $0 \le i \le \ell - 1$,*

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

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

We now introduce the notion of a *locally persevering emulator*. An (α,β)-emulator of a dynamic graph $\mathcal{G} = (G_i)_{0\le i\le k}$ is usually another dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\le i\le k}$ with the same set of nodes as $\mathcal{G}$ that preserves the distance of the original dynamic graph; i.e., for all $i \le k$ and all nodes x and y, there is a path π_{xy} in H_i such that $d_{G_i}(x,y) \le w_i(\pi_{xy}) \le$

$\alpha d_{G_i}(x,y)+\beta$. The notion of a *locally persevering* emulator has another parameter τ. It requires the condition $d_{G_i}(x,y) \le w_i(\pi_{xy}) \le \alpha d_{G_i}(x,y)+\beta$ to hold only when $d_{G_i}(x,y) \le \tau$. More importantly, it puts an additional restriction that the path π_{xy} must be either a shortest path in G_i or a persevering path.

Definition 3.2 (Locally persevering emulator)**.** *Consider parameters $\alpha \ge 1$, $\beta \ge 0$ and $\tau \ge 1$, a dynamic graph $\mathcal{G} = (G_i)_{0\le i\le k}$, and a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\le i\le k}$. For every $i \le k$, we say that a path π in G_i is* contained *in (H_i, w_i) if every edge of π is contained in H_i and has weight* 1*. We say that $\mathcal{H}$ is an (α, β, τ)-*locally persevering emulator *of $\mathcal{G}$ if for all nodes x and y we have*

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

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

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

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

3.1 $(1, 2, \lceil 2/\epsilon \rceil)$-Locally Persevering Emulator of Size $\tilde{O}(n^{3/2})$

In the following we present the locally persevering emulator that we will use to achieve a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$ for decremental approximate APSP. Roughly speaking, we can replace the running time of $\tilde{O}(mn/\epsilon)$ by $\tilde{O}(n^{5/2}/\epsilon^2)$ because this emulator always has $\tilde{O}(n^{3/2})$ edges. However, to be technically correct, we have to use the stronger fact that the number of edges ever contained in the emulator is $\tilde{O}(n^{3/2})$, as in the following statement.

Lemma 3.3 (Existence of $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator with $\tilde{O}(n^{3/2})$ edges)**.** *For every $0 < \epsilon \le 1$ and every decremental graph $\mathcal{G} = (G_i)_{0\le i\le k}$, there is data structure that maintains a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\le i\le k}$ in $O(mn^{1/2} \log n/\epsilon)$ total time such that $\mathcal{H}$ is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator with high probability. Moreover, the number of edges ever contained in the emulator is $|E_k(\mathcal{H})| = O(n^{3/2} \log n)$, and the total number of updates in $\mathcal{H}$ is $\phi_k(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.*

We construct a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\le i\le k}$ as follows. Pick a set D of nodes by including every node independently with probability $(a \ln n)/\sqrt{n}$ for a large enough constant a. Note that the size of D is $O(\sqrt{n} \log n)$ in expectation. It is well known that by this type of sampling every node with degree more than $\sqrt{n}$ has a neighbor in D with high probability (see, e.g., [UY91, DHZ00]); i.e., D *dominates* all high-degree nodes. This is even true for every version G_i of a decremental graph $\mathcal{G} = (G_i)_{0\le i\le k}$. For every

$0 \leq i \leq k$, we define that the graph H_i contains the following two types of edges. For every node $x \in D$ and every node y such that $d_{G_i}(x, y) \leq \lceil 2/\epsilon \rceil + 1$, H_i contains an edge (x, y) of weight $d_{G_i}(x, y)$. For every node x such that $\deg_{G_i}(x) \leq \sqrt{n}$, H_i contains every edge (x, y) of G_i.[13] Note that as edges are deleted from $\mathcal{G}$, distances between nodes might increase, which in turn increases the weights of the corresponding edges in $\mathcal{H}$. When the distance between x and y in $\mathcal{G}$ exceeds $\lceil 2/\epsilon \rceil + 1$, the edge (x, y) is deleted from $\mathcal{H}$.

In the following we prove Lemma 3.3 by arguing that the dynamic graph $\mathcal{H}$ described above has the following four desired properties:

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

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

Lemma 3.4 (Locally persevering)**.** *The dynamic graph $\mathcal{H}$ described above is a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathcal{G}$ with high probability.*

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

We now argue that $\mathcal{H}$ fulfills the second part of the definition of a $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of $\mathcal{G}$. Assume that $d_{G_t}(x, y) \leq \lceil 2/\epsilon \rceil$ and that no shortest path from x to y in G_t is also contained in (H_t, w_t). Let π be an arbitrary shortest path from x to y in G_t. Since π is not contained in H_t, there must be some edge (u, v) on π such that $(u, v) \notin E(H_t)$. This can only happen if u has degree more than $\sqrt{n}$ in G_t. With high probability u has a neighbor $z \in D$ in G_t (see, e.g., [UY91, DHZ00]). Now consider any $i \leq t$. Note that $d_{G_t}(x, u) \leq d_{G_t}(x, y) \leq \lceil 2/\epsilon \rceil$ and that $d_{G_i}(x, z) \leq d_{G_t}(x, z)$ because distances never decrease in a decremental graph. By the triangle inequality we get

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

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

[13] Our construction is very similar to the classic $(1, 2)$-emulator given by Dor, Halperin, and Zwick [DHZ00]. The main difference is that we can only insert edges of limited weight into the emulator; in particular, we only have edges of weight $O(1/\epsilon)$ in the emulator. One reason for this choice is that it is not known whether the $(1, 2)$-emulator of Dor et al. can be maintained in $\tilde{O}(mn)$ time under edge deletions.

(z, y). Since both edges are persevering up to time t, also the path π' is persevering up to time t. Furthermore, π' guarantees the desired approximation:

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

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

Lemma 3.5 (Number of edges)**.** *The number of edges ever contained in the dynamic graph $\mathcal{H}$ is $|E_k(\mathcal{H})| = O(n^{3/2} \log n)$ in expectation.*

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

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

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

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

Lemma 3.6 (Number of updates)**.** *The total number of updates in the dynamic graph $\mathcal{H}$ described above is $\phi_k(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$ in expectation.*

Proof. Only the following kinds of updates appear in $\mathcal{H}$: edge insertions, edge deletions, and edge weight increases. Every edge that is inserted or deleted has to be contained in $\mathcal{H}$ at some time. Thus, we can bound the number of insertions and deletions by $|E_k(\mathcal{H})|$, which is $O(n^{3/2} \log n)$ by Lemma 3.5

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

Lemma 3.7 (Running time)**.** *The edges of the dynamic graph $\mathcal{H}$ described above can be maintained in expected total time $O(mn^{1/2} \log n/\epsilon)$ over all k edge deletions in $\mathcal{G}$.*

Proof. We use the following data structures: (A) For every node, we maintain its incident edges in $\mathcal{H}$ with a dynamic dictionary using dynamic perfect hashing [DKM+94] or cuckoo hashing [PR04]. This graph representation allows us to perform insertions and deletions of edges as well as edge weight increases. (B) For every node x, we maintain the degree of x in $\mathcal{G}$. (C) For every node $x \in D$ we maintain a (classic) ES-tree (see Section 2.2) rooted at x up to distance $\lceil 2/\epsilon \rceil + 1$.

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

We can perform each deletion, insertion, and edge weight increase in expected amortized constant time. As there are $O(n^{3/2} \log n/\epsilon)$ updates in $\mathcal{H}$ in expectation by Lemma 3.6, the expected total time for maintaining (A) is $O(n^{3/2} \log n/\epsilon)$. We need constant time per deletion in $\mathcal{G}$ to update (B) and thus time $O(m)$ in total. Maintaining the ES-tree takes total time $O(m/\epsilon)$ for each node in D (see Section 2.2). Since in expectation there are $O(n^{1/2} \log n)$ nodes in D, the expected total time for maintaining (A) is $O(mn^{1/2} \log n/\epsilon)$ in total.

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

3.2 Maintaining Distances Using Monotone Even–Shiloach Tree

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

Lemma 3.8 (Monotone ES-tree + Locally Persevering Emulator)**.** *For every distance range parameter R^d, every source node r, and every decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ with an (α, β, τ)-locally persevering emulator $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$, the monotone ES-tree on $\mathcal{H}$ is an $(\alpha + \beta/\tau, \beta)$-approximate decremental SSSP data structure for $\mathcal{G}$. It has constant query time and a total update time of*

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

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

Note that we need to modify the ES-tree because although the input graph undergoes only edge deletions, the emulator might have to undergo some edge *insertions*. If we straightforwardly extend the ES-tree to handle insertions, we will have to keep the level of any node y at $\ell(y) = \min_z(\ell(z) + w(y,z))$ as in Line 15 of Algorithm 1. This might cause the level $\ell(y)$ of some node y in the ES-tree to *decrease*. This *destroys the monotonicity* of levels of nodes, which is the key to guaranteeing the running time of the ES-tree, as shown in Section 2.2. The monotone ES-tree is a variant that insists on keeping the nodes' levels monotone (thus the name); it never decreases the level of any node.

Implementation of Monotone ES-Tree. Our monotone ES-tree data structure is a modification of the ES-tree, which always maintains the level $\ell(x)$ of every node x in a shortest paths tree rooted at r up to depth R^{d}, as presented in Section 2.2. Algorithm 2 shows the pseudocode of the monotone ES-tree. Our modification can deal with edge insertions, but does this in a *monotone* manner: it will never decrease the level of any node. In doing so, it will lose the property of providing a shortest paths tree of the underlying dynamic graph, which in our case is the emulator $\mathcal{H}$. However, due to special properties of the emulator, we can still guarantee that the level provided by the monotone ES-tree is an approximation of the distance in the original decremental graph $\mathcal{G}$. The distance estimate provided for a node x is the level of x in the monotone ES-tree.

The overall algorithm now is as follows (see Algorithm 2 for details). We initialize the monotone ES-tree by computing a shortest paths tree in the emulator H_0 up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. For every node x in this tree we set $\ell(x) = d_{H_0}(x, r)$, and for every other node x we set $\ell(x) = \infty$. Starting with these levels, we maintain an ES-tree rooted at r up to depth $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ on the graph $\mathcal{H}$. This ES-tree alone cannot deal with edge insertions and edge weight increases. Our additional procedure that is called after the insertion of an edge (u, v) only updates the value of v in the heap $N(u)$ of u to $\ell(v) + w(u, v)$. In particular, the level of u is not changed after such an insertion.

Order of Updates. Before we start analyzing our algorithm we clarify a crucial detail about the order of updates in the locally persevering emulator $\mathcal{H}$. Consider an edge deletion in the graph G_i that results in the graph G_{i+1}. In the emulator $\mathcal{H}$, it might be the case that several updates are necessary to obtain (H_{i+1}, w_{i+1}) from (H_i, w_i). There could be several insertions, edge weight increases, and edge deletions at once.[14] Our algorithm will process these updates (using the monotone ES-tree) in a specific order: First, it processes the edge insertions, one after the other. Afterward, we process the edge deletions and edge weight increases (also one after the other). This order is crucial for the correctness of our algorithm.

Analysis. We first argue about the correctness of the monotone ES-tree and afterward argue about its running time. In the following we let $\ell_i(u)$ be the level of u in the monotone ES-tree after it has processed the i-th edge deletion in $\mathcal{G}$ (which could mean that it has processed a whole series of insertions, weight increases, and deletions of the emulator $\mathcal{H}$). Remember that (H_i, w_i) denotes the emulator after all updates caused by the i-th deletion in $\mathcal{G}$. We say that an edge (u, v) is *stretched* if $\ell_i(u) \neq \infty$ and $\ell_i(u) > \ell_i(v) + w_i(u, v)$. We say

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

Algorithm 2: Monotone ES-tree

// The algorithm is like the usual ES-tree (Algorithm 1) with three modifications:

1. The algorithm runs on $\mathcal{H} = (H_0, H_1, \ldots)$ instead of $\mathcal{G}$.
2. The depth of the tree is $(\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$ instead of R^{d}
3. There are additional procedures for the insertion of edges and edge weight increases.

// Procedures DELETE() and INCREASE() are the same as before.
// Line numbers in the form i^* indicate lines that are different from Algorithm 1. Blue color marks the changes.

```
1  Procedure INITIALIZE()
2* |  Compute shortest paths tree from r in (H_0, w_0) up to depth (α + β/τ)R^d + β
3  |  foreach node u do
4* |  |  Set ℓ(u) = d_{H_0}(u, r)
5  |  |  for every edge (u, v) do insert v into heap N(u) of u with key ℓ(v) + w(u, v)

6  Procedure INSERT(u, v, w(u, v))
7  |  Insert v into heap N(u) with key ℓ(v) + w(u, v) and u into heap N(v) with key
   |  ℓ(u) + w(u, v)

8  Procedure UPDATELEVELS()
9  |  while heap Q is not empty do
10 |  |  Take node y with minimum key ℓ(y) from heap Q and remove it from Q
11 |  |  ℓ'(y) ← min_z(ℓ(z) + w(y, z))
   |  |  // min_z(ℓ(z) + w(y, z)) can be retrieved from the heap N(y).
   |  |     arg min_z(ℓ(z) + w(y, z)) is y's parent in the ES-tree.
12 |  |  if ℓ'(y) > ℓ(y) then
13 |  |  |  ℓ(y) ← ℓ'(y)
14* |  |  |  if ℓ'(y) > (α + β/τ)R^d + β then ℓ(y) ← ∞
15 |  |  |  foreach neighbor x of y do
16 |  |  |  |  update key of y in heap N(x) to ℓ(y) + w(x, y)
17 |  |  |  |  insert x into heap Q with key ℓ(x) if Q does not already contain x
```

that a node u is *stretched* if it is incident to an edge (u, v) that is stretched. Note that for a node u that is not stretched we either have $\ell_i(u) = \infty$ or $\ell_i(u) \leq \ell_i(v) + w_i(u, v)$ for every edge $(u, v) \in E(H_i)$. Our analysis uses four simple observations about the algorithm. (Recall that a tree edge is an edge between any node y and its parent as in Line 11 of Algorithm 2; i.e., it is an edge (y, z') for some node $z' = \arg\min_z(\ell(z) + w(y, z))$.)

Observation 3.9. *The following holds for the monotone ES-tree:*

(1) The level of a node never decreases.

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

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

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

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

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

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

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

We now prove that the monotone ES-tree provides an $(\alpha + \beta/\tau, \beta)$-approximation of the true distance if it runs on an (α, β, τ)-locally persevering emulator. We use an inductive argument to show that, after having processed the i-th deletion of an edge in $\mathcal{G}$, the level of every node x is a $(\alpha + \beta/\tau, \beta)$-approximation of the distance of x to the root, i.e., $d_{G_i}(x, r) \leq \ell_i(x) \leq (\alpha + \beta/\tau)d_{G_i}(x, r) + \beta$. The intuition why this should be correct is as follows: If the monotone ES-tree gives the desired approximation before a deletion in $\mathcal{G}$ and the deletion does not cause an edge in $\mathcal{H}$ to become stretched, then the structure of the

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

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

Lemma 3.10. *Let $0 < i \leq k$ and assume that $\ell_{i-1}(x') \leq (\alpha + \beta/\tau) \cdot d_{G_{i-1}}(x', r) + \beta$ for every node x' with $\ell_{i-1}(x') \neq \infty$. Then $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$ for every stretched node x.*

Proof. Here we need the assumption that the monotone ES-tree sees the updates in the emulator caused by a single edge deletion in a specific order, namely such that all edge insertions can be processed before the edge weight increases and edge deletions. Since x is stretched, there must have been a previous insertion of an edge (x, y) incident to x such that x is stretched since the time this edge was inserted (see Observation 3.9(2)). Let $\ell'(x)$ denote the level of x after the insertion of (x, y) has been processed. By Observation 3.9, nodes do not change their level as long as they are stretched, and therefore $\ell_i(x) = \ell'(x)$.

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

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

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

In order to prove the approximation guarantee for nonstretched nodes, we have to exploit the properties of the (α, β, τ)-locally persevering emulator $\mathcal{H}$. In the classic ES-tree the level of two nodes differs by at most the weight of a path connecting them—modulo some technical conditions that arise for ES-trees of limited depth. In the monotone ES-tree this is only true for persevering paths (see Lemma 3.12). Before we can show this we need an even simpler property of the monotone ES-tree: If two nodes are connected by an edge that is not stretched, then their levels differ by at most the weight of the edge connecting them. Again, in the classic ES-tree this holds for any edge.

Lemma 3.11. *Consider any $0 \leq i \leq k$ and any $(x, y) \in E(H_i)$. We have*

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

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

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

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

Lemma 3.12. *For every path π from a node x to a node z that (1) is persevering up to time i and (2) has the property that $\ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^d + \beta$, we have $\ell_i(x) \leq \ell_i(z) + w_i(\pi)$.*

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

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

By the definition of persevering paths, every edge (u, v) on π has always existed in $\mathcal{H}$ since the beginning. Therefore the edge (x, y) has never been inserted which means that (x, y) is not stretched by Observation 3.9(2). Since levels and edge weights are nondecreasing we have $\ell_{i-1}(z) + w_{i-1}(\pi) \leq \ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau)R^{\mathrm{d}} + \beta$. By the induction hypothesis for $i - 1$ this implies that $\ell_{i-1}(x) \leq \ell_{i-1}(z) + w_{i-1}(\pi) \neq \infty$. We therefore may apply Lemma 3.11 and get that $\ell_i(x) \leq \ell_i(y) + w_i(x, y) \leq \ell_i(z) + w_i(\pi)$. □

Using the property above, we would ideally like to do the following: Split a shortest path from x to the root r into subpaths of length $\leq \tau$ and replace each subpath by a persevering path such that the length of each subpath and the persevering path by which it is replaced are approximately the same. Repeated applications of the inequality of Lemma 3.12 would then allow us to bound the level of x. However, this approach alone does not work because the definition of a locally persevering emulator does not always guarantee the existence of a persevering path. Instead of a persevering path, the locally persevering emulator might also provide us with a shortest path of G_i that is contained in the current emulator H_i. In principle this is a nice property because a shortest path is even better than an approximate

shortest path. But the problem now is that nodes on this path could be stretched and only for nonstretched nodes can the difference in levels of two nodes be bounded by the weight of the edge between them. We can resolve this issue by induction on the distance to r, which allows us to use the contained path only partially.

Lemma 3.13 (Correctness)**.** *For every node x and every $0 \leq i \leq k$, $\ell_i(x) \geq d_{G_i}(x, r)$, and if $d_{G_i}(x, r) \leq R^d$, then $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$.*

Proof. We start with a proof of the first inequality, $\ell_i(x) \geq d_{G_i}(x, r)$. Consider the (weighted) path π from x to the root r in the monotone ES-tree. Recall that the parent of node v is a node $u = \arg\min_z(\ell(z) + w(y, z))$ as in Line 11 in Algorithm 2. For every edge (u, v) on this path, where v is the parent of u, we have $\ell_i(u) \geq \ell_i(v) + w_i(u, v)$ by Observation 3.9(4). By repeated applications of this inequality for every edge on π we get $\ell_i(x) \geq w_i(\pi) + \ell_i(r) = w_i(\pi)$ (since the level of the root r is always 0). Since π is a path in H_i we have $w_i(\pi) \geq d_{G_i}(x, r)$ because a locally persevering emulator never underestimates the true distance by definition.

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

The claim is clearly true if x is the root node r itself. If $i \geq 1$, then note that $d_{G_{i-1}}(x, r) \leq d_{G_i}(x, r) \leq R^{\mathrm{d}}$, and therefore, by the induction hypothesis for $i - 1$, we have $\ell_{i-1}(x) \neq \infty$. Therefore we may apply Lemma 3.10, which means that the desired inequality holds if x is stretched. Thus, from now on we assume that $x \neq r$ and that x is not stretched. We distinguish two cases.

Case 1: Consider first the case that there is a shortest path from x to r in G_i such that its first edge (x, y) is contained in (H_i, w_i). Note that $d_{G_i}(y, r) < d_{G_i}(x, r)$. Therefore we may apply the induction hypothesis, and by doing so we get $\ell_i(y) \leq (\alpha + \beta/\tau) d_{G_i}(y, r) + \beta$. We now want to argue that $\ell_i(x) \leq \ell_i(y) + w_i(x, y)$ by applying Lemma 3.11. The edge (x, y) is contained in (H_i, w_i) with weight $w_i(x, y) = d_{G_i}(x, y)$, and thus

$$\begin{aligned}
\ell_i(y) + w_i(x, y) = \ell_i(y) + d_{G_i}(x, y) &\leq (\alpha + \beta/\tau) \cdot d_{G_i}(y, r) + \beta + d_{G_i}(x, y) \\
&\leq (\alpha + \beta/\tau) \cdot (d_{G_i}(x, y) + d_{G_i}(y, r)) + \beta \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta \qquad (1) \\
&\leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta\,. \qquad (2)
\end{aligned}$$

Remember that (x, y) is not stretched and if $i \geq 1$, then $\ell_{i-1}(x) \neq \infty$ (as argued above). Using (2) we may now apply Lemma 3.11 and, together with (1), get that

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

as desired.

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

that there is a path π from x to z in (H_i, w_i) that is persevering up to time i such that $w_i(\pi) \leq \alpha d_{G_i}(x,z) + \beta$.

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

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

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

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

The last equation follows from the definition of z.

In both cases we have $\ell_i(z) + w_i(\pi) \leq (\alpha + \beta/\tau) \cdot R^{\mathrm{d}} + \beta$. Since π is persevering up to time i, we may apply Lemma 3.12 and get the following approximation guarantee:

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

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

Lemma 3.14 (Running Time)**.** *For k deletions in $\mathcal{G}$, the monotone ES-tree has a total update time of $O(\phi_k(\mathcal{H}) \log n + |E_k(\mathcal{H})| \cdot ((\alpha + \beta/\tau)R^d + \beta) \log n)$, where $E_k(\mathcal{H})$ is the set of all edges ever contained in $\mathcal{H}$ up to time k.*

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

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

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

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

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

Since the monotone ES-tree is only maintained up to depth $(\alpha+\beta/\tau)R^{\mathrm{d}}+\beta$, at most $(\alpha+\beta/\tau)R^{\mathrm{d}}+\beta$ level increases are possible for every node. Thus, the total update time of the monotone ES-tree is $O(\phi_k(\mathcal{H})\log n+\sum_{x\in U}\deg_{\mathcal{H}}(x)((\alpha+\beta/\tau)R^{\mathrm{d}}+\beta)\log n)$. Since $\sum_{x\in U}\deg_{\mathcal{H}}(x)\leq 2|E_{\mathcal{H}}(U)|$, the total update time is $O(\phi_k(\mathcal{H})+E_{\mathcal{H}}(U)((\alpha+\beta/\tau)R^{\mathrm{d}}+\beta)\log n)$. □

Eliminating the $\log n$-factor. The factor $\log n$ in the running time of Lemma 3.14 comes from using a heap Q and, for every node u, a heap $N(u)$. We now want to avoid using these heaps and only charge $O(\deg_{\mathcal{H}}(u))$ to every level increase of a node u and time $O(1)$ to every update in $\mathcal{H}$. King [Kin99] explained how to eliminate the $\log n$-factor for the classic ES-tree. However, we cannot use the same modified data structures as King because of the possibility of insertions and edge weight increases.

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

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

$$\ell(v)+w(u,v)\leq\ell(u)\,. \tag{5}$$

Therefore we maintain a counter $c(u)$ for every node u such that $c(u)=|\{v\mid\ell(v)+w(u,v)\leq\ell(u)\}|$.[15] If the counters are correctly maintained, we can simply check whether $c(u)$ is 0 to determine whether the level of u has to increase (which replaces Lines 11 and 12 of Algorithm 2). For a node u and its neighbor v the status of Inequality (5) only changes (i.e., the inequality starts or stops being satisfied) in the following cases:

- The level of u or the level of v increases.
- The weight of the edge (u, v) increases.

[15] The idea of maintaining this kind of counter has previously been used by Brim et al. [BCD+11] in the context of mean-payoff games.

- The edge (u, v) is inserted (thus v becomes a neighbor of u).
- The edge (u, v) is deleted (thus v stops being a neighbor of v).

Note that for two nodes u and v we can check whether they satisfy Inequality (5) in constant time. Thus, we can efficiently maintain the counters as follows:

- Every time we update an edge (u, v) (by an insertion, deletion, or weight increase), we check in constant time whether Inequality (5) holds before the update and whether it holds after the update. Then we increase or decrease $c(v)$ and $c(u)$ if necessary. These operations take constant time, which we charge to the update in $\mathcal{H}$.
- Every time $\ell(u)$ increases, we recompute $c(u)$. This takes time $O(\deg_{\mathcal{H}}(u))$. Furthermore, for every neighbor v of u, we check in constant time whether Inequality (5) holds before the update and whether it holds after the update. Then we increase or decrease $c(v)$ if necessary. This takes constant time for every neighbor of u and thus time $O(\deg_{\mathcal{H}}(u))$ for all of them. We can charge the running time $O(\deg_{\mathcal{H}}(u))$ to the level increase of u.

Having explained how to maintain the counters, the remaining running time analysis is the same as in Lemma 3.14. The improved running time can therefore be stated as follows.

Lemma 3.15 (Improved Running Time)**.** *For k deletions in $\mathcal{G}$, the monotone ES-tree can be implemented with a total update time of $O(\phi_k(\mathcal{H}) + |E_k(\mathcal{H})| \cdot ((\alpha + \beta/\tau)R^d + \beta))$, where $E_k(\mathcal{H})$ is the set of all edges ever contained in $\mathcal{H}$ up to time k.*

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

We can achieve this property as follows. For every node u we maintain a list $L(u)$ of nodes. *Every time* a node u and one of its neighbors v start to satisfy Inequality (5), v is appended to $L(u)$. Note that it is *not* always the case that u and all nodes v in the list $L(u)$ satisfy Inequality (5). We just have the guarantee that they satisfied it at some previous point in time. However, the converse is true: If u and its neighbor v currently satisfy Inequality (5), then v is contained in $L(u)$. Using the same argument as above for maintaining the counters, the running time for appending nodes to the lists is paid for by charging $O(1)$ to every update in $\mathcal{H}$ and $O(\deg_{\mathcal{H}}(u))$ to every level increase of a node u.

We can now decide whether the level of a node u has to increase as follows (this replaces Lines 11 and 12 of Algorithm 2). Look at the first node v in the list $L(u)$. If u and v *still* satisfy Inequality (5), the level of u does not have to increase. Otherwise, we retrieve and remove the first element from the list until we find a node v such that u and v satisfy Inequality (5). If no such node v can be found in the list, then the list will be empty after this process and we know that the level of u has to increase. Otherwise, the first node in the list $L(u)$ serves as the parent of u in the tree. The constant running time for reading and removing the first node can be charged to the previous appending of this node to $L(u)$.

Note that the list $L(u)$ of each node u might require a lot of space because some nodes might appear several times. If we want to save space, we can do the following. For every

node u we maintain a set $S(u)$ that stores for every neighbor of u whether it is contained in $L(u)$. Every time we add or remove a node from $L(u)$ we also add or remove it from $S(u)$. Before adding a node to $L(u)$ we additionally check whether it is already contained in $S(u)$ and thus also in $L(u)$. We implement $S(u)$ with a dynamic dictionary using dynamic perfect hashing [DKM+94] or cuckoo hashing [PR04]. This data structure needs time $O(1)$ for look-ups and expected amortized time $O(1)$ for insertions and deletions. Thus, the running time bound of Lemma 3.15 will still hold in expectation. Furthermore, for every node u, the space needed for $L(u)$ and $S(u)$ is bounded by $O(\deg_{\mathcal{H}}(u))$. However, this solution is no longer deterministic.

3.3 From Approximate SSSP to Approximate APSP

In the following, we show how a combination of approximate decremental SSSP data structures can be turned into an approximate decremental APSP data structure. We follow the ideas of Roditty and Zwick [RZ12], who showed how to obtain approximate APSP from *exact* SSSP. We remark that one can obtain an efficient APSP data structure from this reduction, if the running time of the (approximate) SSSP data structure depends on the distance range that it covers in a specific way.

We first define an approximate version of the center cover data structure and show how such a data structure can be obtained from an approximate decremental SSSP data structure by marginally worsening the approximation guarantee. We slightly modify the notions of a center cover and a center cover data structure we gave in Section 2.3, where we reviewed the algorithmic framework of Roditty and Zwick [RZ12]. The main idea behind their APSP data structure is to maintain $\log n$ instances of center cover data structures such that the instance p can answer queries for the approximate distance of two nodes x and y if the distance between them is in the range from 2^p to 2^{p+1}. Arbitrary distance queries can then be answered by performing a binary search over the instances to determine p. We will follow this approach using approximate instead of exact data structures.

Definition 3.16 (Approximate center cover)**.** *Let U be a set of nodes in a graph G, let R^c be a positive integer, the* cover range*, and let $\alpha \geq 1$ and $\beta \geq 0$. We say that a node x is (α, β)-*covered *by a node $c \in U$ in G if $d_G(x, c) \leq \alpha R^c + \beta$. We say that U is an (α, β)-*approximate center cover of G with parameter R^c *if every node x that is in a connected component of size at least R^c is (α, β)-covered by some node $c \in U$ in G.*

Definition 3.17. *An (α, β)-*approximate center cover data structure *with* cover range parameter R^c *and* distance range parameter R^d *for a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ maintains, for every $0 \leq i \leq k$, a set of* centers *$C_i = \{1, 2, \ldots, l\}$ and a set of nodes $U_i = \{c_i^1, c_i^2, \ldots, c_i^l\}$ such that U_i is an (α, β)-approximate center cover of G_i with parameter R^c. For every center $j \in C_i$ and every $0 \leq i \leq k$, we call c_i^j the* location *of center j in G_i and for every node x we say that x is (α, β)-covered by j in G_i if x is (α, β)-covered by c_i^j in G_i. After the i-th edge deletion (where $0 \leq i \leq k$), the data structure provides the following operations:*

- Delete(u, v)*: Delete the edge (u, v) from G_i.*
- Distance(j, x)*: Return an estimate $\delta_i(c_i^j, x)$ of the distance between the location c_i^j of center j and the node x such that $\delta_i(c_i^j, x) \leq \alpha d_{G_i}(c_i^j, x) + \beta$, provided that*

$d_{G_i}(c_i^j, x) \leq R^d$. *If* $d_{G_i}(c_i^j, x) > R^d$, *then either return* $\delta_i(c_i^j, x) = \infty$ *or return* $\delta_i(c_i^j, x) \leq \alpha d_{G_i}(c_i^j, x) + \beta$.

- FindCenter(x)*: If* x *is in a connected component of size at least* R^c*, then return a center* j *(with current location* c_i^j*) such that* $d_{G_i}(x, c_i^j) \leq \alpha R^c + \beta$*. If* x *is in a connected component of size less than* R^c*, then either return* $\perp$ *or return a center* j *such that* $d_{G_i}(x, c_i^j) \leq \alpha R^c + \beta$.

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

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

Lemma 3.18 (Approximate SSSP implies approximate center cover)**.** *Let* R^c *and* R^d *be parameters such that* $R^c \leq R^d$*. If there are* (α, β)*-approximate decremental SSSP data structures with distance range parameters* R^c *and* R^d *for some* $\alpha \geq 1$ *and* $\beta \geq 0$ *that have constant query times and total update times of* $T(R^c)$ *and* $T(R^d)$*, respectively (where* $T(R^d)$ *is* $\Omega(n)$*), then there is an* (α, β)*-approximate center cover data structure that is correct with high probability and has constant query time and an expected total update time of* $O((T(R^d)n \log n)/R^c)$.

Proof. Let $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph. It is well known (see, for example, [UY91] and [RZ12]) that, by random sampling, we can obtain a set $U = \{c^1, c^2, \ldots, c^l\}$ of expected size $O(n \log n / R^c)$ that is a center cover of G_i for every $i \leq k$ with high probability. Clearly, every center cover is also an (α, β)-approximate center cover. Thus, U is an (α, β)-approximate center cover of G_i for every $0 \leq i \leq k$. Throughout all deletions, the set $C = \{1, 2, \ldots, l\}$ will serve as the set of centers and each center j will always be located at the same node c^j.

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

The set S_x can be implemented by using an array of size $|C| = O((n \log n)/R^c)$ for every node x. We initialize S_x in time $O((n \log n)/R^c)$ as follows: For every center j, we query $\delta_0(x, c^j)$ and insert j into S_x if $\delta_0(x, c^j) \leq \alpha R^c + \beta$. Since $\delta_0(x, c^j) \leq \alpha d_{G_0}(x, c^j) + \beta$, this includes every center j such that $d_{G_0}(x, c^j) \leq R^c$. To maintain the sets of centers, we do the following after every deletion. Remember that for every center j, the first SSSP data structure with source c^j returns every node x such that $\delta_i(x, c^j) \leq \alpha R^c + \beta$ and $\delta_{i+1}(x, c^j) > \alpha R^c + \beta$. For every such node x we remove j from S_x. Note that every center j with $\delta_t(x, c^j) > \alpha R^c + \beta$ (for $0 \leq t \leq k$) can safely be removed from S_x because $\delta_t(x, c^j) > \alpha R^c + \beta$ implies $d_{G_t}(x, c^j) > R^c$ and $d_{G_i}(x, c^j) \geq d_{G_t}(x, c^j)$ for all $i \geq t$. We

can charge the running time for maintaining the sets of centers to the delete operations in the SSSP data structures. Thus, this running time is already included in the total update time stated above. For every node x, no center is ever added to S_x after the initialization. Thus, in the array representing S_x, we can maintain a pointer to the leftmost center time proportional to the size of the array, which is $|C| = O((n \log n)/R^c)$.

We now show how to perform the operations of an approximate center cover data structure, as specified in Definition 3.17, in constant time. Let i be the index of the last deletion. Given a center j and a node x, we answer a query for the distance of x to c^j by returning $\delta_i(x, c^j)$ from the second SSSP data structure of c^j, which gives an (α, β)-approximation of the true distance. Given a node x, we answer a query for finding a nearby center by returning any center j in the set of centers S_x of x. If S_x is empty, we return $\perp$. Note that for every center j in S_x we know that $d_{G_i}(x, c^j) \leq \alpha R^c + \beta$, as required, because $d_{G_i}(x, c^j) \leq \delta_i(x, c^j)$. If x is in a connected component of size at least R^c, we can ensure that we find a center j in S_x because, by our random choice of centers, we have $d_{G_i}(x, c^j) \leq R^c$ for some center j with high probability. If $d_{G_i}(x, c^j) \leq R^c$, then S_x contains j. □

We now show why the approximate center cover data structure is useful. If one can obtain an approximate center cover data structure, then one also obtains an approximate decremental APSP data structure with slightly worse approximation guarantee. The proof of this observation follows Roditty and Zwick [RZ12]. In their algorithm, Roditty and Zwick keep a set of nodes U (which we call centers) such that every node (that is in a sufficiently large connected component) is "close" to some node in U. To be able to efficiently find a close center for every node, they maintain, for every node, the nearest node in the set of centers. However, it is sufficient to return *any* center that is close.

Lemma 3.19 (Approximate center cover implies approximate APSP)**.** *Assume that for all parameters R^c and R^d such that $R^c \leq R^d$ there is an (α, β)-approximate center cover data structure that has constant query time and a total update time of $T(R^c, R^d)$. Then, for every $0 < \epsilon \leq 1$, there is an $(\alpha + 2\epsilon\alpha^2, 2\beta + 2\alpha\beta)$-approximate decremental APSP data structure with $O(\log \log n)$ query time and a total update time of $\hat{T} = \sum_{p=0}^{\lfloor \log n \rfloor} T(R_p^c, R_p^d)$, where $R_p^c = \epsilon 2^p$ and $R_p^d = \alpha\epsilon 2^p + \beta + 2^{p+1}$ (for $0 \leq p \leq \lfloor \log n \rfloor$).*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A query time of $O(\log n)$ is immediate as we can simply return the minimum of all distance estimates $\hat{\delta}_i^p(x,y)$. A query time of $O(\log\log n)$ is possible because of the following idea: If $d_{G_i}(x,y) \neq \infty$, it is sufficient to find the minimum index p such that $\hat{\delta}_i^p(x,y) \neq \infty$. This minimum index can be found by performing binary search over all $\log n$ possible indices. Furthermore, the query time can be reduced to $O(1)$ if there is a second (α',β')-approximate decremental APSP data structure with constant query time for some constants α' and β'.

We first compute the distance estimate $\delta'_i(x,y)$ of the second data structure for which we know that $d_{G_i}(x,y) \in [\delta'_i(x,y)/\alpha' - \beta', \delta'_i(x,y)]$. Now there is only a constant number of indices p such that $\{2^p, \dots, 2^{p+1}\} \cap [\delta'_i(x,y)/\alpha' - \beta', \delta'_i(x,y)] \neq \emptyset$. For every such index we compute $\hat{\delta}^p_i(x,y)$ and return the minimum distance estimate obtained by this process. □

Finally, we show how to obtain an approximate decremental APSP data structure from an approximate decremental SSSP data structure if the approximation guarantee is of the form $(\alpha + \epsilon, \beta)$. In that case we can avoid the worsening of the approximation guarantee of Lemma 3.18.

Lemma 3.20. *Assume that for some $\alpha \geq 1$ and $\beta \geq 0$, every $0 < \epsilon \leq 1$, and all $0 \leq R^d$ there is an $(\alpha + \epsilon, \beta)$-approximate decremental SSSP data structure with distance range parameter R^d that has constant query time and a total update time of $T'(R^d, \epsilon)$. Then there is an $(\alpha + \epsilon, \beta)$-approximate decremental APSP data structure with a query time of $O(\log \log n)$ and a total update time of*

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

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

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

Proof. By combining Lemma 3.18 with Lemma 3.19 the approximate decremental SSSP data structure implies that there is an $(\hat{\alpha}, \hat{\beta})$-approximate decremental APSP data structure where $\hat{\alpha} = (\alpha + \hat{\epsilon}) + 2\hat{\epsilon}(\alpha + \hat{\epsilon})^2$ and $\hat{\beta} = 2\beta + 2(\alpha + \hat{\epsilon})\beta$. This APSP data structure has a query time of $O(\log \log n)$ and a total update time of

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

By Lemma 3.19 the query time can be reduced to $O(1)$ if, for some constants α' and β', there is an (α', β')-approximate decremental APSP data structure with constant query time and a total update time of $\hat{T}$.

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

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

and

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

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

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

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

3.4 Putting Everything Together: $\tilde{O}(n^{5/2})$-Total Time Algorithm for $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-Approximate APSP

In the following we show how the monotone ES-tree of Lemma 3.8 together with the locally persevering emulator of Lemma 3.3 can be used to obtain $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-approximate decremental APSP data structures with $\tilde{O}(n^{5/2}/\epsilon^2)$ total update time. These results are direct consequences of the previous parts of this section. We first show how to obtain a $(1+\epsilon, 2)$-approximate decremental SSSP data structure. Using Lemma 3.20 we then immediately obtain a $(1+\epsilon, 2)$-approximate decremental APSP data structure.

Corollary 3.21 ($(1+\epsilon, 2)$-approximate monotone ES-tree)**.** *Given the $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator $\mathcal{H}$ of Lemma 3.3, there is a $(1+\epsilon, 2)$-approximate decremental SSSP data structure for every distance range parameter R^d that is correct with high probability, and has constant query time and an expected total update time of $O(n^{3/2} \log n/\epsilon + n^{3/2} R^d \log n)$, where the time for maintaining $\mathcal{H}$ is not included.*

Proof. Let $\mathcal{G} = (G_i)_{0 \le i \le k}$ be a decremental graph and let $\mathcal{H}$ be the $(1, 2, \lceil 2/\epsilon \rceil)$-locally persevering emulator of Lemma 3.3. By Lemma 3.8 there is an approximate decremental SSSP data structure for every source node r and every distance range parameter R^{d}. Let $\delta_i(x, r)$ denote the estimate of the distance between x and r provided after the i-th edge deletion in $\mathcal{G}$. By Lemma 3.8 we have $d_{G_i}(x, c) \le \delta_i(x, c)$, and furthermore, if $d_{G_i}(x, c) \le R^{\mathrm{d}}$, then

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

By Lemma 3.3, the number of edges ever contained in the emulator is $|E_k(\mathcal{H})| = O(n^{3/2} \log n)$ and the total number of updates in $\mathcal{H}$ is $\phi_k(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$. Therefore, by Lemma 3.8, the total update time of the approximate decremental SSSP data structure is

$$\begin{aligned} &O(\phi_k(\mathcal{H}) + |E_k(\mathcal{H})| \cdot ((\alpha + \beta/\tau) R^{\mathrm{d}} + \beta)) \\ &\qquad = O((n^{3/2} \log n)/\epsilon + (n^{3/2} \log n) \cdot ((1+\epsilon) R^{\mathrm{d}} + 2)) \\ &\qquad\qquad = O((n^{3/2} \log n)/\epsilon + n^{3/2} R^{\mathrm{d}} \log n)\,. \end{aligned}$$ □

Theorem 3.22 (Main result of Section 3: Randomized $(1+\epsilon, 2)$-approximation with truly-subcubic total update time)**.** *For every $0 < \epsilon \le 1$, there is a $(1+\epsilon, 2)$-approximate decremental APSP data structure with constant query time and an expected total update time of $O((n^{5/2} \log^3 n)/\epsilon)$ that is correct with high probability.*

Proof. We set $\hat{\epsilon} = \epsilon/18$. Let $\mathcal{H}$ denote the $(1, 2, 2/\hat{\epsilon})$-locally persevering emulator of Lemma 3.3. The total update time for maintaining $\mathcal{H}$ is $O(mn^{1/2} \log n/\epsilon)$. Since $m \le n^2$ this

is within the claimed total update time. By Corollary 3.21 we can use $\mathcal{H}$ to maintain, for every distance range parameter R^{d}, a $(1+\hat{\epsilon}, 2)$-approximate decremental SSSP data structure that has constant query time and a total update time of $T(R^{\mathrm{d}}) = O((n^{3/2}\log n)/\epsilon + n^{3/2}R^{\mathrm{d}}\log n)$.

Using $\alpha = 1$ and $\beta = 2$, it follows from Lemma 3.20 that there is a $(1+\epsilon, 2)$-approximate decremental APSP data structure whose total update time is proportional to

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

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

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

The query time of the APSP data structure provided by Lemma 3.20 can be reduced to $O(1)$. The reason is that Bernstein and Roditty [BR11] provide, for example, a $(5+\epsilon', 0)$-approximate decremental APSP data structure for some constant ϵ'. The total update time of this data structure is

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

which is well within $O(n^{5/2})$. □

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

Corollary 3.23 (Randomized $(2+\epsilon, 0)$-approximation with truly-subcubic total update time)**.** *For every $0 < \epsilon \le 1$, there is a $(2+\epsilon, 0)$-approximate decremental APSP data structure with constant query time and an expected total update time of $O((n^{5/2}\log^3 n)/\epsilon)$ that is correct with high probability.*

Proof. By using the data structure of Theorem 3.22 we can, after the i-th edge deletion in a decremental graph $\mathcal{G} = (G_i)_{0\le i\le k}$ and for all nodes x and y, query for a distance estimate $\delta_i(x, y)$ in constant time that satisfies

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

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

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

If $d_{G_i}(x, y) < 2$, then we actually have $d_{G_i}(x, y) \le 1$ because G_i is an unweighted graph. A distance of 1 simply means that there is an edge connecting x and y in G_i. Since the

adjacency matrix of $\mathcal{G}$ is maintained anyway, we can find out in constant time whether $d_{G_i}(x,y) = 1$. By setting, for all nodes x and y,

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

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

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

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

Theorem 4.1 (Main result of Section 4: Deterministic $O((mn\log n)/\epsilon)$ total update time)**.** *For every $0 < \epsilon \leq 1$, there is a deterministic $(1+\epsilon,0)$-approximate decremental APSP data structure with a total update time of $O((mn\log n)/\epsilon)$ and a query time of $O(\log\log n)$.*

Using known reductions, we show in Section 4.3 that this decremental algorithm implies a deterministic fully dynamic algorithm with an amortized running time of $\tilde{O}(mn/(\epsilon t)$ per update and a query time of $\tilde{O}(t)$ for every $t \leq n$.

The main task in proving Theorem 4.1 is to design a deterministic version of the *center cover data structure* (see Section 2.3) with a total deterministic update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$ and constant query time. Once we have this data structure, Theorem 4.1 directly follows as a corollary from Theorem 2.14. Note that we cannot use the same idea as in [RZ12] to reduce the query time from $O(\log\log n)$ to $O(1)$. This would require a *deterministic* (α,β)-approximate decremental APSP data structure for some constants α and β with *constant* query time and a total update time of $O((mn\log n)/\epsilon)$. To the best of our knowledge such a data structure has not yet been developed.

Recall that R^{c} and R^{d} are the *coverage range* and *distance range* parameters where (a) we want every node (in a connected component of size at least R^{c}) to be within distance of at most R^{c} from some center, and (b) we want to maintain the distance from each center to every node within distance at most R^{d}. Roditty and Zwick [RZ12], following an argument of Ullman and Yannakakis [UY91], observed that making each node a center independently at random with probability $(a\ln n)/R^{\mathrm{c}}$, where a is a constant and $1 \leq R^{\mathrm{c}} \leq n$, gives a set C of centers such that with probability at least $1 - n^{-(a-1)}$ the conditions of a center cover with parameter R^{c} are fulfilled by C in the initial graph and the expected size of C is $O((n\log n)/R^{\mathrm{c}})$. The randomized decremental APSP algorithm of [RZ12] simply chooses a large enough value of a so that with high probability C not only fulfills the center cover properties with parameter R^{c} in the initial graph but continues to fulfill them in all the $O(n^2)$ graphs generated during $O(n^2)$ edge deletions. This is only possible because it is assumed that the "adversary" that generates the deletions is oblivious, i.e., does not know the location

of the centers. The main challenge for the *deterministic* algorithm is to *dynamically adapt* the location and number of the centers so that (i) the center cover properties with size R^c continue to hold, while the graph is modified, and (ii) the total cost incurred is $O(mn)$. Once we have such a data structure, we can use the approach of Roditty and Zwick, as discussed in Section 2.3, to obtain an algorithm for maintaining decremental approximate shortest paths.

The *new feature* of our deterministic center cover data structure is that it sometimes *moves* centers to avoid opening too many centers (which are expensive to maintain). As we described in Section 1, the key technique behind the new data structure is what we call a *moving Even–Shiloach tree*. We note that the moving ES-tree is actually a concept rather than a new implementation: we implement it in a straightforward way by building a new ES-tree every time we have to move it. However, analyzing the total update time needs new insights and a careful charging argument. To separate the analysis of the moving ES-tree from the charging argument, we describe the data structure in two parts:

(1) First, in Section 4.1, we give the *moving centers* data structure that can answer DISTANCE and FINDCENTER queries, but needs to be told *where* to move a center, when a center has to be moved. This data structure is basically an implementation of *several* moving ES-trees.[16]

(2) Then, in Section 4.2, we show how to determine when a center (with a moving ES-tree rooted at it) has to be moved and, if so, where it has to move.

Combining these two parts gives the center cover data structure.

4.1 A Deterministic Moving Centers Data Structure (`MovingCenter`)

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

Definition 4.2 (Moving centers data structure)**.** *A* moving centers data structure *with* cover range parameter R^c *and* distance range parameter R^d *for a decremental graph* $\mathcal{G} = (G_i)_{0 \le i \le k}$ *maintains, for every* $0 \le i \le k$*, a set of* centers $C_i = \{1, 2, \ldots, l\}$ *and a set of nodes* $U_i = \{c_i^1, c_i^2, \ldots, c_i^l\}$*. For every center* $j \in C$ *and every* $0 \le i \le k$*, we call* $c_i^j \in U_i$ *the* location *of center* j *in* G_i*. After the* i*-th edge deletion (where* $0 \le i \le k$*), the data structure provides the following operations:*

- DELETE(u, v)*: Delete the edge* (u, v) *from* G_i*.*
- OPEN(x)*: Open a new center at node* x *and return the ID of the opened center for later use.*
- MOVE(j, x)*: Move the center* j *from its current location* c_i^j *to node* x*.*

[16] Later on we want to use the moving centers data structure, and not directly the moving ES-trees, because we will need an additional operation which is not directly provided by the ES-trees (in particular, the FINDCENTER operation defined in Section 4.1).

- DISTANCE(j, x): *Return the distance $d_{G_i}(c_i^j, x)$ between the location c_i^j of center j and the node x, provided that $d_{G_i}(c_i^j, x) \leq R^d$. If $d_{G_i}(c_i^j, x) > R^d$, then return ∞.*
- FINDCENTER(x): *Return a center j (with location c_i^j) such that $d_{G_i}(x, c_i^j) \leq R^c$. If no such center exists, return $\perp$.*

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

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

In the rest of Section 4.1 we use the following notation: The decremental graph $\mathcal{G}$ undergoes a sequence of k edge deletions. By G_i we denote the graph after the i-th deletion (for $0 \leq i \leq k$). Each deletion in the graph is reported to the moving centers data structure by a delete operation. By C_i we denote the set of centers at the time of the i-th delete operation, and by c_i^j we denote the location of center $j \in C_i$ at the time of the i-th delete operation.

Definition 4.3 (Moving distance (D_{move}))**.** *The* total moving distance*, denoted by D_{move}, is defined as $D_{move} = \sum_{0 \leq i < k} \sum_{j \in C_i} d_{G_i}(c_i^j, c_{i+1}^j)$.*

The main result of this section is that we can maintain a moving centers data structure in $O(m(N_{\text{open}}R^{\text{d}} + D_{\text{move}}))$ time, as in Proposition 4.4 below. The data structure is actually very simple: We maintain an ES-tree of depth at most R^{d} at every node for which we open a center and for every node to which we move a center. Note that our algorithm treats an ES-tree at each such node as a new tree, regardless of whether we open a center or move a center there. While the algorithm can naively treat each ES-tree as a new one, the analysis cannot: If we do so, we will get a total update time of $O((N_{\text{open}} + N_{\text{move}})mR^{\text{d}})$, where N_{move} is the total number of move-operations (since maintaining each ES-tree takes $O(mR^{\text{d}})$ total update time). Instead, we bound the cost incurred by the move-operation based on how far a center is moved, i.e., the moving distance D_{move}. This argument allows us to replace the unfavorable term $N_{\text{move}}mR^{\text{d}}$ by $D_{\text{move}}m$. The deterministic center cover data structure of Section 4.2 will generate a sequence of center open and move requests so that $D_{\text{move}} = O(n)$. For simplifying the analysis, we state the following result under a technical assumption, which will always be fulfilled by the intended use of the moving centers data structure in Section 4.2.[17]

Proposition 4.4 (Main result of Section 4.1: deterministic moving centers data structure)**.** *Let R^c and R^d be parameters such that $R^c \leq R^d$. Under the assumption that between two consecutive delete operations there can be at most one open or delete operation for each center, there is a moving centers data structure with a total deterministic update time of*

[17] Without this assumption the total update time will be $O((N_{\text{open}}R^{\text{d}} + N_{\text{move}} + D_{\text{move}})m)$, where N_{open} is the number of open-operations, N_{move} is the number of move operations, and D_{move} is the total moving distance.

$O((N_{open}R^d + D_{move})m)$, where N_{open} is the number of open-operations and D_{move} is the total moving distance.[18] *The data structure can answer each query in constant time.*

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

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

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

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

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

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

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

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

[18] Note that the total moving distance might be ∞ if, after some deletion i, a center j is moved from c_i^j to c_{i+1}^j such that there is no path between c_i^j and c_{i+1}^j in G_i. In this case our analysis cannot bound the total update time of the moving centers data structure.

$\sum_{0\le i<k} \deg_{G_0}(x) \cdot d_{G_i}(c_i^j, c_{i+1}^j)$ additional time will be spent updating x in the ES-tree of center j.

Consider the $(i+1)$-th edge deletion and let $j \in C_{i+1}$ be a center. By Corollary 2.11, the total time for processing this deletion in the ES-tree of center j is

$$T(i+1,j) = \sum_{x\in V} \deg_{G_0}(x) \cdot \left(\min\left(d_{G_{i+1}}(x, c_{i+1}^j), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x, c_{i+1}^j), R^{\mathrm{d}}\right)\right) \tag{6}$$

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

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

Therefore we get

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

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

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

Consider now a center j and the o_j-th edge deletion. By (6) we can bound the running time $T(o_j, j)$ as follows:

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

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

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

By summing up this quantity over all centers and switching the order of the double sum, we arrive at the following total time:

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

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

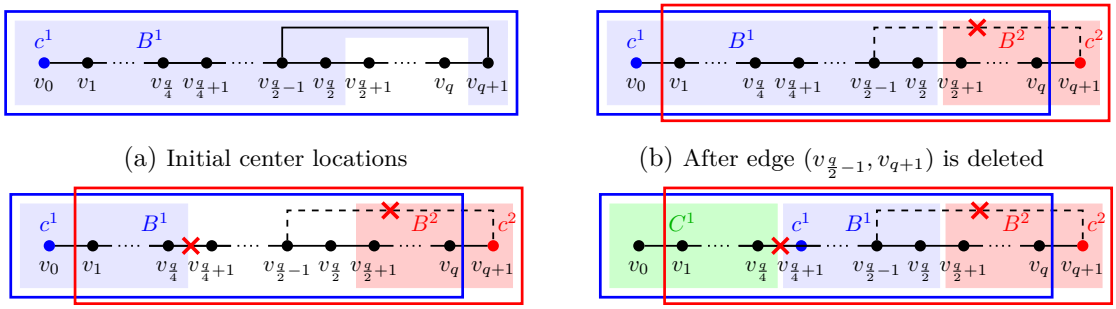
4.2 A Deterministic Center Cover Data Structure (`CenterCover`)

In this section, we present a deterministic algorithm for maintaining the center cover data structure CenterCover, as defined in Definition 2.13. That is, for parameters R^{c} and R^{d}, we show that we can maintain a set of centers with the following two properties. First, all nodes in a connected component of size at least R^{c} are *covered* by some center; i.e., each of them is in distance at most R^{c} to some center. Second, for every center, the distance to every node up to distance R^{d} is maintained. This section is devoted to proving the following.

Proposition 4.5 (Main result of Section 4.2)**.** *For every cover range parameter R^c and every distance range parameter R^d such that $R^c \le R^d$, there is a center cover data structure with a total* deterministic *update time of $O(mnR^d/R^c)$ and constant query time.*

4.2.1 High-Level Ideas

Our algorithm will internally use the moving centers data structure from Section 4.1 (called MovingCenter). It has to determine how to open and move centers in a way that ensures that at any time every node in a connected component of size at least R^{c} is covered by some center,;i.e., its distance to the nearest center is at most R^{c}. At a high level, our algorithm is very simple (see Figure 3 for an example; note that $q = R^{\mathrm{c}}$): For each center j, it maintains two sets B^j and C^j, where B^j is always defined to be the set of nodes whose distance to



(a) Initial center locations

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

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

(d) After moving c^1

Figure 3: Example of our algorithm for maintaining the center cover data structure using the moving centers data structure, as in Proposition 4.5. We use $q = R^{\mathrm{c}}$ and, for any j, we let c^j denote the location of center j. Boxes filled with colors show sets B^j and C^j. (a) shows a possible initial location of center c^1. This makes $B^1 = \{v_0, \ldots, v_{q/2}\} \cup \{v_{q+1}\}$ and $C^1 = \emptyset$. All nodes are covered by center c^1. (b) shows what our algorithm does when edge $(v_{q/2-1}, v_{q+1})$ is deleted. In this case, v_{q+1} is not covered by c^1 anymore, so we open a center c^2 at v_{q+1}. (c) shows what B^1 will look like after edge $(v_{q/4}, v_{q/4+1})$ is deleted, if we do not move center c^1. In particular, $|B^1 \cup C^1| < q/2$. (d) shows what our algorithm will do after edge $(v_{q/4}, v_{q/4+1})$ is deleted to maintain the largeness property (i.e., to make sure that $|B^1 \cup C^1| \geq q/2$): it moves nodes $v_0, \ldots, v_{q/4}$ from B^1 to C^1 and moves the first center from v_0 to $v_{q/4+1}$.

center j is at most $R^{\mathrm{c}} - |C^j|$. Initially, the algorithm sets $C^j = \emptyset$ and chooses a set of centers such that all sets B^j are disjoint (see Figure 3a). The sets C^j will never decrease during the algorithm. After an edge deletion, if some node in a large connected component (size $\geq R^{\mathrm{c}}$) that is no longer covered by any center (e.g., v_{q+1} in Figure 3b), then the algorithm simply opens a new center at that node. However, before doing so it has to check whether $|B^j \cup C^j| < R^{\mathrm{c}}/2$ for some existing center j. (For example, after edge $(v_{q/4}, v_{q/4+1})$ is deleted as in Figure 3c, $|B^1 \cup C^1| = (q/4 + 1) < q/2 = R^{\mathrm{c}}/2$.) If this is the case for center j, it will add all nodes of B^j to C^j and move the center j to the end-node of the deleted edge that is in a different connected component than the old location of j. As we will show, the nodes in B^j at the new location are *not* contained in $B^{j'}$ for any center $j' \neq j$; i.e., the invariant that all sets B^j are disjoint remains valid. For example, in Figure 3d, the algorithm puts nodes $v_0, \ldots, v_{q/4}$ to C^1 and moves c^1 to node $v_{q/4+1}$, which is the end-node of the deleted edge $(v_{q/4}, v_{q/4+1})$ that is in a connected component different from center c^1.

We now give the intuition behind this algorithm and its analysis before going into detail. Recall from Proposition 4.4 that opening and maintaining a center together costs $O(mR^{\mathrm{c}})$ time in total, and a move-operation incurs a total time of $O(m)$ per one unit moving distance. So, to get the desired $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$ total time bound, we will make sure that our algorithm uses a limited number of open-operations and a limited moving distance; in particular, we will make sure that

$$N_{\mathrm{open}} = O(n/R^{\mathrm{c}}) \quad \text{and} \quad D_{\mathrm{move}} = O(n).$$

To guarantee that we open at most $O(n/R^{\mathrm{c}})$ centers, we imagine that each node holds a coin at the beginning of the algorithm, which it can give to at most one center during the algorithm, and we require that each center must receive at least $R^{\mathrm{c}}/2$ coins from some

nodes in the end. Clearly, this will automatically ensure that at most $2n/R^{\mathrm{c}}$ centers will be opened. Since the graph keeps changing, it is hard to say which node should give a coin to which center at the beginning. Instead, our algorithm will maintain two sets for each center j: the set B^j of *borrowed* nodes from which center j has borrowed coins that it might have to return, and the set C^j of *collected* nodes from which center j has collected coins that it will never return. After all edge deletions, j will hold the coins of all nodes in $B^j \cup C^j$. Our algorithm will maintain $B^j \cup C^j$ with two properties:

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

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

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

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

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

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

(P2) (Shrinking) We never add any node to $B^j \cup C^j$. (For example, $B^1 \cup C^1$ in Figure 3a is a subset of $B^1 \cup C^1$ in Figure 3d.)

These two properties are sufficient to guarantee the disjointness property because if two sets $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint at the beginning (by Property (P1)), they will remain disjoint if we never add a node to them (by Property (P2)). The shrinking property (Property (P2)) can be checked simply by observing the behavior of the algorithm (see Lemma 4.12 for details). To show the initial-disjointness property (Property (P1)), we use the fact that j is of distance at least R^{c} from other centers when j is opened, which implies that $B^j \cap B^{j'} = \emptyset$. Additionally, we will prove that $C^{j'}$ contains only nodes in connected components of size less than R^{c}, whereas any new center j is opened in a connected component of size at least R^{c}. This implies that $B^j \cap C^{j'} = \emptyset$ when j is opened.

Finally, for the confinement property, just observe that before the algorithm moves a center j, it puts all nodes in the connected component containing the center j to C^j and moves j to a node outside of this connected component. For example, in Figure 3d the algorithm puts nodes $v_1, \ldots, v_{q/4}$ to C^1 before moving the first center through $v_1, \ldots, v_{q/4}$ to $v_{q/4+1}$.

4.2.2 Algorithm Description

Our algorithm is outlined in Algorithm 3. For each center j, the algorithm maintains its location c^j, which could change over time since centers can be moved. In addition, it also maintains the set C^j and the number r^j, which are set to $\emptyset$ and $R^{\mathrm{c}}/2$, respectively, when center j is opened. The intended value of r^j is $r^j = R^{\mathrm{c}}/2 - |C^j|$, and the algorithm always updates r^j in a way that this is ensured. The algorithm also uses the moving centers data structure (denoted by MovingCenter and explained in Section 4.1) to maintain the distance between each center j to other nodes in the graph, up to distance R^{d}. This helps us to implement CenterCover.FindCenter and CenterCover.Distance queries in a straightforward way: the algorithm just invokes the same queries from the moving centers data structure.

Initially, on G_0 (i.e., before the graph changes), our algorithm initializes the moving centers data structure by opening centers in a greedy manner: as long as there is a node x that is not covered by any center, it opens a center at x. This process will also be used every time an edge is deleted, to make sure that every node remains covered by a center. Procedure CenterCover.GreedyOpen proceeds as follows. For every node x, it checks whether x is *not covered*; this is the case if CenterCover.FindCenter(x) returns $\perp$ and the size of the connected component containing x is at least R^{c} (we refer the reader to Lemma 4.21 for how to compute the size of this component). If x is not covered, the algorithm opens a center at x, stores the index j of this new center, and initializes the values of C^j, r^j, and c^j, as in Line 6. This completes the GreedyOpen procedure.

The main work of Algorithm 3 lies in the Delete operation, since it has to make sure that all nodes are still covered by some centers after the deletion. Procedure CenterCover.Delete proceeds as follows. Let us assume that the $(i+1)$-th edge (u, v) is deleted from G_i, and let G_{i+1} denote the resulting graph. First, the procedure checks whether there is any center j that is in a large component in G_i and in a small connected component in G_{i+1}; i.e., the size of the connected component of c^j is at least r^j in G_i and less than r^j in G_{i+1} (see Line 16 of Algorithm 3). Next, if such a center j in a small connected component exists (in fact, we will show that there exists at most one such j; see Lemma 4.15), we will *move* j to a different component and update the values of C^j, r^j, and c^j. It is crucial in our analysis that j must be moved carefully. In particular, we will move j to either u or v, depending on which node is in a *different* component from c^j, the current location of j. (Note that one of u and v will be in the same connected component as j and the other will be in a different component.) We use a variable $y \in \{u, v\}$ to refer to the new location to which we move center j (see Line 18). We then update the values of C^j, r^j, and c^j. In particular, we put *all* nodes in the connected component that previously contained center j (before we move it to y) into C^j and update r^j to $R^{\mathrm{c}}/2 - |C^j|$ and c^j to y. Then we report the move of center j to y to the moving centers data structure. Afterward, we report the deletion of the edge (u, v) to the moving centers data structure so that it updates the

Algorithm 3: `CenterCover` (Deterministic Center Cover Data Structure)

// Given a decremental graph $\mathcal{G} = (G_i)_{0 \le i \le k}$ and integers R^{c} and R^{d}, this data structure maintains a set of centers such that every node (that is in a connected component of size at least R^{c}) has distance at most R^{c} to at least one center and we can query the distance between a center and a node if their distance is at most R^{d} (otherwise, we will get ∞ in return). It allows four operations: Initialize, Delete, FindCenter and Distance, as defined in Definition 2.13.

Procedure CenterCover.GreedyOpen()
  Let G_i denote the current graph
  foreach *node* x **do**
    // The if-statement checks if x is *not* covered by a center and the size of the connected component containing it is larger than R^{c}. See Lemma 4.21 for the implementation detail.
    if FindCenter$(x) = \perp$ **and** $|\mathrm{Comp}_{G_i}(x)| \ge R^c$ **then**
      // tell moving centers data structure to open new center at x. Let j be the index of this center.
      $j \leftarrow$ MovingCenter.Open(x)
      Set $C^j \leftarrow \emptyset$, $r^j \leftarrow R^{\mathrm{c}}/2$, and $c^j \leftarrow x$

// Parameters: Initial version G_0 of decremental graph $\mathcal{G}$, integers R^{c} and R^{d}.
Procedure CenterCover.Initialize(G_0, R^c, R^d)
  // Initialize the moving centers data structure (see Definition 4.2)
  MovingCenter.Initialize(G_0, R^c, R^d)
  GreedyOpen()
Procedure CenterCover.FindCenter(v) // Parameter: Node v.
  return MovingCenter.FindCenter(v)
// Parameters: Center index j and node v.
Procedure CenterCover.Distance(j, v)
  return MovingCenter.Distance(j, v)
// Parameter: $(i+1)$-th deleted edge (u, v).
Procedure CenterCover.Delete(u, v)
  Let G_i denote the graph before deleting (u, v) and let G_{i+1} denote the graph afterwards.
  // Find a center j for which the connected component containing it becomes smaller than r^j. See Lemma 4.22 for how to find such a center j. (Actually, there will be at most one such center, see Lemma 4.15.)
  Find a center j such that $|\mathrm{Comp}_{G_{i+1}}(c^j)| < r^j$.
  if *such a center* j *exists* **then**
    // Move j to either u or v depending on who is in a different connected component than c^j.
    if u *and* c^j *are* not *connected in* G_{i+1} **then** $y \leftarrow u$ **else** $y \leftarrow v$
    Set $C^j \leftarrow C^j \cup \mathrm{Comp}_{G_{i+1}}(c^j)$, $r^j \leftarrow r^j - |\mathrm{Comp}_{G_{i+1}}(c^j)|$, and $c^j \leftarrow y$
    MovingCenter.Move(j, y)
  // Report edge deletion to moving centers data structure (Definition 4.2).
  MovingCenter.Delete(u, v)
  GreedyOpen()

distances between centers and nodes to the new distances in G_{i+1}. Finally, we execute the CenterCover.GreedyOpen procedure to make sure that every node remains covered: if there is a node x that is not covered, we open a center at x. This completes the deletion operation.

4.2.3 Analysis

The correctness of Algorithm 3 is immediate. As the procedure GreedyOpen is called after every edge deletion, every node in a connected component of size at least R^c will always be covered. In the following we analyze the running time of Algorithm 3.

Our main task is to bound the running time of the moving centers data structure internally used by the algorithm. In particular we want to use the running time bound stated in Proposition 4.4 which requires us to bound the number N_{open} of open-operations performed by the algorithm and the total moving distance D_{move}. As outlined in Section 4.2.1 we assign to each center j the set $B^j \cup C^j$. The set C^j contains all nodes of connected components in which the center j once was located, as shown in Algorithm 3. The set B^j is the set of all nodes that are at distance at most r^j from the center j in the current graph. We first show that the sets $B^j \cup C^j$ fulfill two properties: disjointness and largeness. Disjointness says that for all centers $j \neq j'$ the sets $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint. Largeness says that the set $B^j \cup C^j$ has size at least $R^c/2$ for each center j. Using these two properties, we will prove that there are at most $N_{\text{open}} = O(n/R^c)$ open-operations and that the total moving distance is $D_{\text{move}} = O(n)$. These bounds will then allow us to obtain a total update time of $O(mnR^d/R^c)$ for the moving centers data structure used by Algorithm 3. Afterward we will show that all other operations of the algorithm can also be carried out within this total update time. To make our arguments precise we will use the following notation.

Definition 4.6. *Let $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ be a decremental graph for which Algorithm 3 maintains a center cover data structure. The graph undergoes a sequence of k deletions, and by G_i we denote the graph after the i-th deletion. We use the following notation:*

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

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

Observation 4.7. *Let x be a node, let $i \leq k$, and let B' be the set $B' = \{y \in V \mid d_i(x,y) \leq r\}$ for some integer r. If $|\operatorname{Comp}_i(x)| < r$, then $\operatorname{Comp}_i(x) = B'$. Furthermore, $|\operatorname{Comp}_i(x)| < r$ if and only if $|B'| < r$.*

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

Observation 4.8. *For every center j and every $i \leq k$, $r_i^j = R^c/2 - |C_i^j|$.*

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

Observation 4.9. *For every center j and every $i \leq k$, $|\operatorname{Comp}_i(x)| < R^c$ for every node $x \in C_i^j$.*

Proof. For every node x that is put into C_i^j after the i-th edge deletion, we have $|\operatorname{Comp}_i(x)| < r_i^j$. Since the size of the connected component of x never increases in a decremental graph and $r_i^j \leq R^c$ for all $i \leq k$ by Observation 4.8, the claim is true. □

Observation 4.10. *For every center j and every $i \leq k$, the sets B_i^j and C_i^j are disjoint.*

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

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

Lemma 4.11 (Initial disjointness)**.** *When the algorithm opens a center j after the i-th edge deletion, the set $B_i^j \cup C_i^j$ is disjoint from the set $B_i^{j'} \cup C_i^{j'}$ for every other center $j \neq j'$.*

Proof. Let j be the center that is opened, and let $j' \neq j$ be an existing center. The algorithm sets $C_i^j = \emptyset$, and therefore we only have to argue that B_i^j and $B_i^{j'} \cup C_i^{j'}$ are disjoint. Note that c_i^j is in a connected component of size at least R^c, because otherwise the algorithm would not have opened a center at c_i^j. Observe that the set B_i^j is contained in the connected component of c_i^j. By Observation 4.9 all nodes of $C_i^{j'}$ are in a connected component of size

less than R^c, and therefore $B_i^j \cap C_i^{j'} = \emptyset$. We now argue that $B_i^j \cap B_i^{j'} = \emptyset$. Suppose that there is some node x contained in both B_i^j and $B_i^{j'}$. By the definition of B_i^j and $B_i^{j'}$ we get $d_i(c_i^j, x) \leq r_i^j = R^c/2 - |C_i^j| \leq R^c/2$ as well as $d_i(c_i^{j'}, x) \leq R^c/2$. By the triangle inequality we get

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

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

Lemma 4.12 (Shrinking property)**.** *For every center j and every $i < k$, we have $B_i^j \cup C_i^j \subseteq B_{i+1}^j \cup C_{i+1}^j$.*

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

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

The observation needed for proving the shrinking property is $B_{i+1}^j \cup A \subseteq B_i^j$. From this observation we get $B_{i+1}^j \cup C_{i+1}^j = B_{i+1}^j \cup C_i^j \cup A \subseteq B_i^j \cup C_i^j$, as desired. We first prove $A \subseteq B_i^j$ and then $B_{i+1}^j \subseteq B_i^j$. Let B' be the set $B' = \{z \in V \mid d_{i+1}(x, z) \leq r_i^j\}$. Since $|A| < r_i^j$ we get $A \subseteq B'$ by Observation 4.7, and since $d_i(x, z) \leq d_{i+1}(x, z)$ for every node z we have $B' \subseteq B_i^j$. Now $A \subseteq B'$ and $B' \subseteq B_i^j$, and we may conclude that $A \subseteq B_i^j$.

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

$$d_i(x, z) \leq d_i(x, v) + d_i(v, z) \leq d_i(x, v) + d_{i+1}(v, z) \leq |A| + r_{i+1}^j = r_i^j\,.$$ □

Lemma 4.13 (Disjointness)**.** *Algorithm 3 maintains the following invariant: For all centers $j \neq j'$ and every $i \leq k$, $B_i^j \cup C_i^j$ is disjoint from $B_i^{j'} \cup C_i^{j'}$.*

Proof. By Lemma 4.11 the invariant holds after the initialization. Now consider the $(i+1)$-th edge deletion. Let $j \neq j'$ be two different existing centers. By the induction hypothesis $B_i^j \cup C_i^j$ and $B_i^{j'} \cup C_i^{j'}$ are disjoint. Since $B_{i+1}^j \cup C_{i+1}^j \subseteq B_i^j \cup C_i^j$ and $B_{i+1}^{j'} \cup C_{i+1}^{j'} \subseteq B_i^{j'} \cup C_i^{j'}$ by Lemma 4.12, also $B_{i+1}^j \cup C_{i+1}^j$ and $B_{i+1}^{j'} \cup C_{i+1}^{j'}$ are disjoint. Now let j be an existing center and let j' be a center that is opened in the procedure CenterCover.GreedyOpen

(called at the end of the procedure CenterCover.Delete). By Lemma 4.11 we also have that $B^j_{i+1} \cup C^j_{i+1}$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1}$ are disjoint. This shows that, for *all* centers j and j' such that $j \neq j'$, $B^j_{i+1} \cup C^j_{i+1}$ and $B^{j'}_{i+1} \cup C^{j'}_{i+1}$ are disjoint. □

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

Observation 4.14. *Let (u, v) be the $(i+1)$-th deleted edge. If $|\operatorname{Comp}_i(c^j_i)| \geq r^j_i$ and $|\operatorname{Comp}_{i+1}(c^j_i)| < r^j_i$, then $u \in B^j_i$ and $v \in B^j_i$.*

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

Lemma 4.15 (Uniqueness of center to move)**.** *Let (u, v) be the $(i+1)$-th deleted edge. If $|B^j_i| \geq r^j_i$ for every center j, then there is at most one center j such that $|\operatorname{Comp}_{i+1}(c^j_i)| < r^j_i$ and, in G_{i+1}, either u is connected to c^j_i (and v is disconnected from c^j_i) or v is connected to c^j_i (and u is disconnected from c^j_i).*

Proof. Let j be a center such that $|\operatorname{Comp}_{i+1}(c^j_i)| < r^j_i$. As $|B^j_i| \geq r^j_i$, we also have $|\operatorname{Comp}_i(c^j_i)| \geq r^j_i$ by Observation 4.7. The size of the connected component of c^j_i can only decrease if the deletion of (u, v) disconnects at least one node from $\operatorname{Comp}_i(c^j)$. For this to happen, u and v must be connected to c^j_i in G_i, and furthermore one of these nodes (either u or v) must be disconnected from c^j_i in G_{i+1} while the other node stays connected to c^j_i.

Now suppose that there are two centers $j \neq j'$ such that $|\operatorname{Comp}_i(c^j_i)| \geq r^j_i$ and $|\operatorname{Comp}_{i+1}(c^j_i)| < r^j_i$, and $|\operatorname{Comp}_i(c^{j'}_i)| \geq r^{j'}_i$ and $|\operatorname{Comp}_{i+1}(c^{j'}_i)| < r^{j'}_i$. By Observation 4.14, we get $u \in B^j_i$ and $u \in B^{j'}_i$, which contradicts the disjointness property of Lemma 4.13. We conclude that there cannot be two such centers $j \neq j'$. □

Lemma 4.16. *For every center j and every $i \leq k$, Algorithm 3 maintains the invariant $|B^j_i| \geq r^j_i$.*

Proof. We first argue that the invariant holds for every center j that we open at some node x in the greedy open procedure after the i-th deletion. The algorithm only opens the center if x is in a connected component of size at least R^c. Since $r^j_i = R^c/2 - |C^j_i| \leq R^c$ (Observation 4.8) we have $|\operatorname{Comp}_i(x)| \geq r^j_i$. Therefore we get $|B^j_i| \geq r^j_i$ by Observation 4.7.

We now show that the invariant is maintained for all centers that have already been opened before we delete the $(i+1)$-th edge (u, v). Consider first the case that $|\operatorname{Comp}_{i+1}(c_i^j)| \geq r_i^j$. In that case the center j will not be moved and we have $C_{i+1}^j = C_i^j$, $B_{i+1}^j = B_i^j$, and $r_{i+1}^j = r_i^j$. Since $|\operatorname{Comp}_{i+1}(c_{i+1}^j)| \geq r_{i+1}^j$ we get $|B_{i+1}^j| \geq r_{i+1}^j$, as desired by Observation 4.7.

Now consider the case that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. Since the invariant holds for i, Lemma 4.15 applies, and thus we can be sure that the algorithm will move center j from node x to node y (where either $y = u$ or $y = v$). Remember that we have $c_i^j = x$, $c_{i+1}^j = y$, and $r_{i+1}^j = r_i^j - |\operatorname{Comp}_{i+1}(x)|$ in that case. Since x and y were connected in G_i but are no longer connected in G_{i+1} we get $\operatorname{Comp}_{i+1}(y) = \operatorname{Comp}_i(x) \setminus \operatorname{Comp}_{i+1}(x)$. Due to $\operatorname{Comp}_{i+1}(x) \subseteq \operatorname{Comp}_i(x)$ it follows that

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

By Observation 4.7, the fact that $|\operatorname{Comp}_{i+1}(y)| \geq r_{i+1}^j$ implies that $|B_{i+1}^j| \geq r_{i+1}^j$ as desired. □

Lemma 4.17 (Largeness)**.** *For every center j and every $i \leq k$, Algorithm 3 maintains the invariant $|B_i^j \cup C_i^j| \geq R^c/2$.*

Proof. By Observation 4.10, B_i^j and C_i^j are disjoint, and by Observation 4.8 we have $r_i^j = R^c/2 - |C_i^j|$. By Lemma 4.16 we have $|B_i^j| \geq r_i^j$. Therefore we get the desired bound as follows:

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

where the inequality above follows from Lemma 4.16. □

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

Lemma 4.18 (Number of open-operations)**.** *Over all edge deletions, Algorithm 3 performs $O(n/R^c)$ open-operations in its internal moving centers data structure.*

Proof. Let C_k denote the set of centers after all k deletions. Note that moving a center does not change the number of centers. Therefore, the size of C_k is equal to the total number of centers opened. Due to the disjointness property (Lemma 4.13) the sets $B_k^j \cup C_k^j$ after all k edge deletions are disjoint for all centers j. When we sum up over all these sets we do not count any node twice. Therefore we get

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

By the largeness property (Lemma 4.17) every set $B_k^j \cup C_k^j$ has size at least $R^c/2$, i.e., $|B_k^j \cup C_k^j| \geq R^c/2$. We now combine both inequalities and get

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

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

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

Lemma 4.19 (Confinement)**.** *For every move of center j from c_i^j to c_{i+1}^j after the $(i+1)$-th edge deletion, let π_i^j be the set of nodes on a shortest path from c_i^j to c_{i+1}^j in G_i. Then, for every center j and every $0 \leq i < k$, $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$.*

Proof. Let (u, v) be the $(i+1)$-th deleted edge. Consider the situation that the algorithm moves some center j from c_i^j to c_{i+1}^j. By the rules of the algorithm for moving centers we have either $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Due to Observation 4.14 we have $c_{i+1}^j \in B_i^j$, which means that $d_i(c_i^j, c_{i+1}^j) \leq r_i^j$.

Now let π_i^j be a shortest path from c_i^j to c_{i+1}^j in G_i. All nodes in π_i^j, except for c_{i+1}^j, are connected to c_i^j in G_{i+1} since the edge (u, v) only appears as the last edge on the shortest path due to $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Therefore we have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq \mathrm{Comp}_{i+1}(c_i^j)$. Since $C_{i+1}^j = C_i^j \cup \mathrm{Comp}_{i+1}(c_i^j)$, we get $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j$. We also have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq B_i^j$ because $d_i(c_i^j, c_{i+1}^j) \leq r_i^j$. Since B_i^j and C_i^j are disjoint (Observation 4.10), also $\pi_i^j \setminus \{c_{i+1}^j\}$ and C_i^j are disjoint. It therefore follows that $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$. □

Lemma 4.20 (Total moving distance)**.** *The total moving distance of the moving centers data structure used by Algorithm 3 is $D_{move} = O(n)$.*

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

Consider the situation that the algorithm moves a center j from c_i^j to c_{i+1}^j after the $(i+1)$-th edge deletion and let π_i^j be a shortest path from c_i^j to c_{i+1}^j in G_i as in Lemma 4.19. The shortest path π_i^j consists of $d_i(c_i^j, c_{i+1}^j)$ many edges and $d_i(c_i^j, c_{i+1}^j) + 1$ many nodes. Therefore we get $d_i(c_i^j, c_{i+1}^j) = |\pi_i^j \setminus \{c_{i+1}^j\}|$. By Lemma 4.19 we have $\pi_i^j \setminus \{c_{i+1}^j\} \subseteq C_{i+1}^j \setminus C_i^j$ for every center j and every $0 \leq i < k$. The value of the set C^j after all edge deletions is given by C_k^j for every center j. By the disjointness property (Lemma 4.13) we have $\sum_{j \in C} |C_k^j| = \left|\bigcup_{j \in C} C_k^j\right|$. We now obtain the bound $D_{\mathrm{move}} \leq n$ as follows:

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

□

Implementation Details. Before we finish this section we clarify two implementation details of Algorithm 3 and argue that they can be carried out within the total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$.

There are two places in the algorithm where we have to compute the sizes of connected components. First, in the procedure GreedyOpen, we have to check for every node that is

not covered by any center whether it is in a connected component of size at least R^c. Second, in the procedure DELETE, we have to check whether the size of the connected component of some center j drops below r^j. So far we have not explained explicitly how to carry out these steps. If we could obtain the size of the connected component deterministically in linear time, the running time analysis we have given so far would suffice. Remember that the moving centers data structure internally maintains an ES-tree for every center. Thus, it would seem intuitive to use the ES-trees for counting the number of nodes in the current component of each center. However, we do not report edge deletions to the moving centers data structure immediately. Therefore it is not clear how to use these ES-trees to determine the size of the connected components of a centers.

Instead, we do the following. In parallel to our own algorithm we use the deterministic (fully) dynamic connectivity data structure of Henzinger and King [HK01].[19] This data structure can answer queries of the form "are the nodes x and y connected?" in constant time. Its amortized time per deletion is $O(n^{1/3} \log n)$. Thus, its total update time over all deletions is $O(mn^{1/3} \log n)$. Trivially, this data structure allows us to compute the size of the connected component of a node x in time $O(n)$: We simply iterate over all nodes and count how many of them are connected to x. We now explain how to perform the two tasks listed above using the dynamic connectivity data structure.

Lemma 4.21 (Detail of Line 4 in Algorithm 3)**.** *Given a dynamic connectivity data structure with constant query time, performing the check in the if-condition of Line 4 takes time $O((n + N_{open})n)$ over all deletions, where N_{open} is the total number of open-operations.*

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

Lemma 4.22 (Detail of Line 16 of Algorithm 3)**.** *Given a dynamic connectivity data structure with constant query time, we can, after the $(i+1)$-th deletion, find a center j such that $|\text{Comp}_{i+1}(c_i^j)| < r_i^j$ if it exists in time $O(n)$.*

Proof. Let (u, v) be the $(i+1)$-th deleted edge. For every center j such that $|\text{Comp}_{i+1}(c_i^j)| < r_i^j$, we have $u \in B_i^j$ by Observation 4.14. Moreover, by the disjointness property (Lemma 4.13), there can only be at most one center j such that $u \in B_i^j$. The algorithm for finding this center now is simple: We find a center j such that $u \in B_i^j$, which is unique if it exists; then we compute the size of the connected component containing c_i^j using the dynamic

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

connectivity data structure [HK01]. In particular, we iterate over all centers in time $O(n)$ to find a candidate center j such that $d_i(u, c_i^j) \leq r^j$ (i.e., $u \in B_i^j$) (as argued above, at most one such center exists). We can determine the distance $d_i(u, c_i^j)$ in constant time by querying the moving centers data structure. For the candidate center j we now have to check whether $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. We determine the size of $\operatorname{Comp}_{i+1}(c_i^j)$ in time $O(n)$ by using the dynamic connectivity data structure with constant query time. Thus, the running time for this algorithm is $O(n)$ per deletion. □

Total Update Time. Now we state the total update time of Algorithm 3. The bounds on the number of centers opened and the total moving distance of the centers allow us to bound the running time of the moving centers data structure used by the algorithm.

Theorem 4.23. *The deterministic center cover data structure of Algorithm 3 has constant query time and a total update time of $O(mnR^d/R^c)$.*

Proof. By Proposition 4.4 the moving centers data structure internally used by Algorithm 3 has constant query time and a total deterministic update time of $O(N_{\text{open}}mR^{\text{d}} + D_{\text{move}}m)$, where N_{open} is the total number of open-operations and D_{move} is the total moving distance. Algorithm 3 delegates all queries to the moving centers data structure and therefore also has constant query time. By Lemma 4.18 the number of open-operations is $O(n/R^{\text{c}})$, and by Lemma 4.20 the total moving distance is $O(n)$. Therefore the total update time of the moving centers data structure is

$$O(N_{\text{open}}mR^{\text{d}} + D_{\text{move}}m) = O(mnR^{\text{d}}/R^{\text{c}} + mn) = O(mnR^{\text{d}}/R^{\text{c}})$$

because $R^{\text{c}} \leq R^{\text{d}}$. As argued in Lemma 4.21 and **??**, all other operations of the algorithm can be implemented within a total update time of $O(mnR^{\text{d}}/R^{\text{c}})$. Therefore the claimed running time follows. □

4.3 Deterministic Fully Dynamic Algorithm

There is a well-known reduction by Henzinger and King [HK95] for converting a decremental algorithm into a fully dynamic algorithm. A similar approach has been used by Roditty and Zwick [RZ12], using the decremental algorithm we derandomized above as the starting point. In the following we sketch the deterministic fully dynamic algorithm implied by Theorem 4.1. The fully dynamic algorithm allows two update operations: deleting an arbitrary set of edges and inserting a set of edges touching a node v, called the center of the insertion.

Theorem 4.24. *For every $0 < \epsilon \leq 1$ and every $t \leq \sqrt{n}$ there is a deterministic fully dynamic $(1+\epsilon, 0)$-approximate APSP data structure with amortized update time $O(mn/(\epsilon t))$ and query time $\tilde{O}(t)$.*

Proof. The algorithm works in phases. After each t update operations we start a new phase. At the beginning of each phase we re-initialize the decremental algorithm of Theorem 4.1. We report to this algorithm all future deletions of the phase, but no insertions. For all nodes u and v let $\delta_1(u, v)$ denote the $(1+\epsilon)$-approximate distance estimate obtained by the decremental algorithm. Additionally, after every update in the graph, we do the following: Let I denote the set of centers of all insertions that so far happened in the current phase.

For every $v \in I$, we compute the shortest paths from v to all nodes in the current graph, i.e., where all insertions and deletions are considered. We use Dijkstra's algorithm for this task and denote by $\delta_2(u, v)$ the distance from u to v computed in this way.

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

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

We remark that the fully dynamic result of Roditty and Zwick [RZ12] is still a bit stronger. Their trade-off is basically the same, but it holds for a larger range of t, namely, $t \leq m^{1/2-\delta}$ for every fixed $\delta > 0$. The reason is that they use randomization not only for the decremental algorithm but also for some other part of the fully dynamic algorithm.

5 Conclusion

We obtained two new algorithms for solving the decremental approximate APSP algorithm in unweighted undirected graphs. Our first algorithm provides a $(1 + \epsilon, 2)$-approximation and has a total update time of $\tilde{O}(n^{5/2}/\epsilon)$ and constant query time. The main idea behind this algorithm is to run an algorithm of Roditty and Zwick [RZ12] on a sparse dynamic emulator. In particular, we modify the central shortest paths tree data structure of Even and Shiloach [ES81, Kin99] to deal with edge insertions in a monotone manner. Our approach is conceptually different from the approach of Bernstein and Roditty [BR11], who also maintain an ES-tree in a sparse dynamic emulator. The sparsification techniques used here and at other places only work for undirected graphs. Using a new sampling technique, we recently obtained a $(1 + \epsilon, 0)$-approximation for decremental SSSP in *directed* graphs with constant query time and a total update time of $o(mn)$ [HKN14c].

Our second algorithm provides a $(1 + \epsilon, 0)$-approximation and has a *deterministic* total update time of $O(mn \log n/\epsilon)$ and constant query time. We obtain it by derandomizing the algorithm of [RZ12] using a new amortization argument based on the idea of relocating ES-trees.

Our results directly motivate the following directions for further research. It would be interesting to extend our derandomization technique to other randomized algorithms. In particular, we ask whether it is possible to derandomize the *exact* decremental APSP algorithm of Baswana, Hariharan, and Sen [BHS07] (total update time $\tilde{O}(n^3)$).

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

Maybe the most important open problem in this field is a faster APSP algorithm for the fully dynamic setting. The fully dynamic algorithm of Demetrescu and Italiano [DI04] provides exact distances and takes time $\tilde{O}(n^2)$ per update, which is essentially optimal. Is it possible to get a faster fully dynamic algorithm that still provides a good approximation—for example a $(1+\epsilon)$-approximation?

References

[ABC+98] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. "Near-Linear Time Construction of Sparse Neighborhood Covers". In: *SIAM Journal on Computing* 28.1 (1998). Announced at FOCS'93, pp. 263–277 (cit. on p. 7).

[AC13] Ittai Abraham and Shiri Chechik. "Dynamic Decremental Approximate Distance Oracles with $(1+\epsilon, 2)$ stretch". In: *CoRR* abs/1307.1516 (2013) (cit. on p. 15).

[ACI+99] Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. "Fast Estimation of Diameter and Shortest Paths (Without Matrix Multiplication)". In: *SIAM Journal on Computing* 28.4 (1999). Announced at SODA'96, pp. 1167–1181 (cit. on pp. 7, 15).

[AFI06] Giorgio Ausiello, Paolo Giulio Franciosa, and Giuseppe F. Italiano. "Small Stretch Spanners on Dynamic Graphs". In: *Journal of Graph Algorithms and Applications* 10.2 (2006). Announced at ESA'05, pp. 365–385 (cit. on p. 8).

[AIMS+91] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. "Incremental Algorithms for Minimal Length Paths". In: *Journal of Algorithms* 12.4 (1991). Announced at SODA'90, pp. 615–638 (cit. on p. 14).

[AIM+92] Giorgio Ausiello, Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. "On-Line Computation of Minimal and Maximal Length Paths". In: *Theoretical Computer Science* 95.2 (1992), pp. 245–261 (cit. on p. 14).

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

[BCD+11] Lubos Brim, Jakub Chaloupka, Laurent Doyen, Raffaella Gentilini, and Jean-François Raskin. "Faster algorithms for mean-payoff games". In: *Formal Methods in System Design* 38.2 (2011). Announced at MEMICS'09 and GAMES'09, pp. 97–118 (cit. on p. 37).

[BDBK+94] Shai Ben-David, Allan Borodin, Richard M. Karp, Gábor Tardos, and Avi Wigderson. "On the Power of Randomization in On-Line Algorithms". In: *Algorithmica* 11.1 (1994). Announced at STOC'90, pp. 2–14 (cit. on p. 6).

[BDH+12] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. "Graph Expansion and Communication Costs of Fast Matrix Multiplication". In: *Journal of the ACM* 59.6 (2012). Announced at SPAA'11, 32:1–32:23 (cit. on p. 6).

[Ber09] Aaron Bernstein. "Fully Dynamic $(2 + \epsilon)$ Approximate All-Pairs Shortest Paths with Fast Query and Close to Linear Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2009, pp. 693–702 (cit. on p. 8).

[Ber13] Aaron Bernstein. "Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2013, pp. 725–734 (cit. on pp. 2, 6, 15).

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998, pp. I–XVIII, 1–414 (cit. on p. 6).

[BHS03] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Maintaining All-Pairs Approximate Shortest Paths Under Deletion of Edges". In: *Symposium on Discrete Algorithms (SODA)*. 2003, pp. 394–403 (cit. on p. 15).

[BHS07] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. "Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths". In: *Journal of Algorithms* 62.2 (2007). Announced at STOC'02, pp. 74–92 (cit. on pp. 4, 5, 15, 64).

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. "Fully Dynamic Randomized Algorithms for Graph Spanners". In: *ACM Transactions on Algorithms* 8.4 (2012). Announced at ESA'04 and SODA'08, 35:1–35:51 (cit. on p. 8).

[BR11] Aaron Bernstein and Liam Roditty. "Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions". In: *Symposium on Discrete Algorithms (SODA)*. 2011, pp. 1355–1365 (cit. on pp. 1, 4–10, 15, 45, 64).

[Coh98] Edith Cohen. "Fast Algorithms for Constructing t-Spanners and Paths with Stretch t". In: *SIAM Journal on Computing* 28.1 (1998). Announced at FOCS'93, pp. 210–236 (cit. on p. 7).

[CZ01] Edith Cohen and Uri Zwick. "All-Pairs Small-Stretch Paths". In: *Journal of Algorithms* 38.2 (2001). Announced at SODA'97, pp. 335–353 (cit. on p. 7).

[DHZ00] Dorit Dor, Shay Halperin, and Uri Zwick. "All-Pairs Almost Shortest Paths". In: *SIAM Journal on Computing* 29.5 (2000). Announced at FOCS'96, pp. 1740–1759 (cit. on pp. 1, 6, 7, 9, 15, 26, 27, 69).

[DI02] Camil Demetrescu and Giuseppe F. Italiano. "Improved Bounds and New Trade-Offs for Dynamic All Pairs Shortest Paths". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2002, pp. 633–643 (cit. on p. 14).

[DI04] Camil Demetrescu and Giuseppe F. Italiano. "A New Approach to Dynamic All Pairs Shortest Paths". In: *Journal of the ACM* 51.6 (2004). Announced at STOC'03, pp. 968–992 (cit. on pp. 14, 65).

[DI06] Camil Demetrescu and Giuseppe F. Italiano. "Fully dynamic all pairs shortest paths with real edge weights". In: *Journal of Computer and System Sciences* 72.5 (2006). Announced at FOCS'01, pp. 813–837 (cit. on pp. 4, 5, 14).

[DKM+94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. "Dynamic Perfect Hashing: Upper and Lower Bounds". In: *SIAM Journal on Computing* 23.4 (1994). Announced at FOCS'88, pp. 738–761 (cit. on pp. 29, 39).

[Elk05] Michael Elkin. "Computing Almost Shortest Paths". In: *ACM Transactions on Algorithms* 1.2 (2005). Announced at PODC'01, pp. 283–323 (cit. on pp. 7, 15).

[Elk11] Michael Elkin. "Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners". In: *ACM Transactions on Algorithms* 7.2 (2011). Announced at ICALP'07, 20:1–20:17 (cit. on p. 8).

[EP04] Michael Elkin and David Peleg. "$(1+\epsilon, \beta)$-Spanner Constructions for General Graphs". In: *SIAM Journal on Computing* 33.3 (2004). Announced at STOC'01, pp. 608–631 (cit. on p. 7).

[ES81] Shimon Even and Yossi Shiloach. "An On-Line Edge-Deletion Problem". In: *Journal of the ACM* 28.1 (1981), pp. 1–4 (cit. on pp. 1, 4, 5, 7, 14, 18, 64).

[FR06] Jittat Fakcharoenphol and Satish Rao. "Planar graphs, negative weight edges, shortest paths, and near linear time". In: *Journal of Computer and System Sciences* 72.5 (2006). Announced at FOCS'01, pp. 868–889 (cit. on p. 14).

[HK01] Monika R. Henzinger and Valerie King. "Maintaining Minimum Spanning Forests in Dynamic Graphs". In: *SIAM Journal on Computing* 31.2 (2001). Announced at ICALP'97, pp. 364–374 (cit. on pp. 12, 62, 63).

[HK95] Monika R. Henzinger and Valerie King. "Fully Dynamic Biconnectivity and Transitive Closure". In: *Symposium on Foundations of Computer Science (FOCS)*. 1995, pp. 664–672 (cit. on pp. 18, 63).

[HKN13a] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *Symposium on Foundations of Computer Science (FOCS)*. 2013, pp. 538–547 (cit. on p. 15).

[HKN13b] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks". In: *International Colloquium on Automata, Languages and Programming (ICALP)*. 2013, pp. 607–619 (cit. on p. 15).

[HKN14a] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "A Subquadratic-Time Algorithm for Decremental Single-Source Shortest Paths". In: *Symposium on Discrete Algorithms (SODA)*. 2014, pp. 1053–1072 (cit. on p. 15).

[HKN14b] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time". In: *Symposium on Foundations of Computer Science (FOCS)*. 2014, pp. 146–155 (cit. on p. 15).

[HKN14c] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Sublinear-Time Decremental Algorithms for Single-Source Reachability and Shortest Paths on Directed Graphs". In: *Symposium on Theory of Computing (STOC)*. 2014, pp. 674–683 (cit. on pp. 15, 64).

[HKNrt] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. "Dynamic Approximate All-Pairs Shortest Paths: Breaking the $O(mn)$ Barrier and Derandomization". In: *SIAM Journal on Computing* (forthcoming). Announced at FOCS'13 (cit. on p. 1).

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

[HKR+97] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. "Faster Shortest-Path Algorithms for Planar Graphs". In: *Journal of Computer and System Sciences* 55.1 (1997). Announced at STOC'94, pp. 3–23 (cit. on p. 14).

[HLT01] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *Journal of the ACM* 48.4 (2001). Announced at STOC'98, pp. 723–760 (cit. on p. 12).

[Kin99] Valerie King. "Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs". In: *Symposium on Foundations of Computer Science (FOCS)*. 1999, pp. 81–91 (cit. on pp. 7, 14, 18–20, 37, 64).

[LC67] P.S. Loubal and Bay Area Transportation Study Commission. *A Network Evaluation Procedure*. Bay Area Transportation Study Commission, 1967 (cit. on p. 13).

[Mur67] John D. Murchland. *The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph*. Tech. rep. LBS-TNT-26. London Business School, Transport Network Theory Unit, 1967 (cit. on p. 13).

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004). Announced at ESA'01, pp. 122–144 (cit. on pp. 29, 39).

[RT13] Liam Roditty and Roei Tov. "Approximating the Girth". In: *ACM Transactions on Algorithms* 9.2 (2013). Announced at SODA'11, 15:1–15:13 (cit. on p. 6).

[RZ11] Liam Roditty and Uri Zwick. "On Dynamic Shortest Paths Problems". In: *Algorithmica* 61.2 (2011). Announced at ESA'04, pp. 389–401 (cit. on pp. 6, 14).

[RZ12] Liam Roditty and Uri Zwick. "Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs". In: *SIAM Journal on Computing* 41.3 (2012). Announced at FOCS'04, pp. 670–683 (cit. on pp. 1, 4–8, 11, 15, 22–24, 39–41, 46, 63, 64).

[Tho04] Mikkel Thorup. "Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles". In: *Scandinavian Workshop on Algorithm Theory (SWAT)*. 2004, pp. 384–396 (cit. on p. 14).

[TZ05] Mikkel Thorup and Uri Zwick. "Approximate Distance Oracles". In: *Journal of the ACM* 52.1 (2005). Announced at STOC'01, pp. 74–92 (cit. on pp. 7, 8).

[TZ06] Mikkel Thorup and Uri Zwick. "Spanners and emulators with sublinear distance errors". In: *Symposium on Discrete Algorithms (SODA)*. 2006, pp. 802–809 (cit. on p. 8).

[UY91] Jeffrey D. Ullman and Mihalis Yannakakis. "High-Probability Parallel Transitive-Closure Algorithms". In: *SIAM Journal on Computing* 20.1 (1991). Announced at SPAA'90, pp. 100–125 (cit. on pp. 11, 26, 27, 40, 46).

[VWW10] Virginia Vassilevska Williams and Ryan Williams. "Subcubic Equivalences between Path, Matrix and Triangle Problems". In: *Symposium on Foundations of Computer Science (FOCS)*. 2010, pp. 645–654 (cit. on pp. 1, 6, 69).

[VWY09] Virginia Vassilevska, Ryan Williams, and Raphael Yuster. "All Pairs Bottleneck Paths and Max-Min Matrix Products in Truly Subcubic Time". In: *Theory of Computing* 5.1 (2009). Announced at STOC'07, pp. 173–189 (cit. on p. 6).

[Zwi02] Uri Zwick. "All Pairs Shortest Paths using Bridging Sets and Rectangular Matrix Multiplication". In: *Journal of the ACM* 49.3 (2002). Announced at FOCS'98, pp. 289–317 (cit. on p. 7).

A Proof of Fact 1.1

Due to a reduction by Dor, Halperin, and Zwick [DHZ00], a combinatorial[20] algorithm for APSP, even a $(2-\epsilon, 0)$-approximation or $(1+\epsilon, 1)$-approximation one,[21] with running time $O(n^{3-\delta})$, for any $\delta > 0$, will imply a combinatorial algorithm for *Boolean matrix multiplication* with the same running time, another breakthrough result. Further, due to Vassilevska Williams and Williams [VWW10, Theorem 1.3], the $O(n^{3-\delta})$-time combinatorial algorithm will imply breakthrough results for a few other problems. Since combinatorial dynamic algorithms can be used to solve static APSP, the same argument applies. In particular, the additive error of 2 in our $(1+\epsilon, 2)$-approximation algorithm is unavoidable if we wish to get an $O(n^{1-\delta})$ running time (a so-called *truly subcubic* time) and keep a small multiplicative error of $1+\epsilon$. For the same reason, a multiplicative error of 2 in our $(2+\epsilon, 0)$-approximation algorithm is also unavoidable. Similarly, the running time of our deterministic algorithm cannot be improved further unless we allow larger additive or multiplicative errors.

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

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