Dynamic Approximate All-Pairs Shortest Paths: Breaking the $\tilde{O}(mn)$ 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 $\tilde{O}(mn)$ and constant query time by Roditty and Zwick [33] (FOCS 2004). The fastest deterministic algorithm is from a 1981 paper by Even and Shiloach [23]; 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 $\tilde{O}(n^{5/2})$ and constant query time that has an additive error of two in addition to the $1+\epsilon$ multiplicative error. This beats the previous $\tilde{O}(mn)$ 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 [19] and many other long-standing problems [39].

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 $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ running time of Bernstein and Roditty [12] (SODA 2011) in terms of both approximation and time guarantees.

(2) We present a deterministic algorithm with a total update time of $\tilde{O}(mn)$ and a query time of $O(\log\log n)$. The algorithm has a multiplicative error of $1+\epsilon$ and gives the first improved deterministic algorithm since 1981. It also answers an open question raised by Bernstein in his STOC 2013 paper [11].

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.

*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 Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 317532.

†Division of Mathematical Sciences, Nanyang Technological University, Singapore 637371. Work partially done while at University of Vienna, Faculty of Computer Science, Austria.

Contents

List of Theorems

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 additions. 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 vertex x and vertex y in the current graph G, denoted by $d_G(x, y)$.

We use the term *single-source shortest paths* (SSSP) to refer to the special case where the distance query can be done only when $x = s$, for a pre-specified *source node* s. The efficiency is judged by two parameters: *query time* denoting the time needed to answer *each* distance query, and *total update time* denoting the time needed to process *all* edge deletions. The running time will be in terms of n, the number of nodes in the graph, and m, the number of edges *before* any deletion. We use $\tilde{O}$-notation to hide an $O(\text{poly} \log n)$ 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 $\delta(x, y)$ 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* algorithms was $\tilde{O}(mn^2)$ by one of the earliest papers in the area from 1981 by Even and Shiloach [23]. The fastest *exact randomized* algorithms are the $\tilde{O}(n^3)$-time algorithms by Demetrescu and Italiano [18] and Baswana, Hariharan, and Sen [7]. The fastest *approximation* algorithm is the $\tilde{O}(mn)$-time $(1 + \epsilon, 0)$-approximation algorithm by Roditty and Zwick [33]. If we insist on an $O(n^{3-\delta})$ running time, for some constant $\delta > 0$, Bernstein and Roditty [12] obtain an $\tilde{O}(n^{2+1/k+O(1/\sqrt{\log n})})$-time $(2k - 1 + \epsilon, 0)$-approximation algorithm, for any integer $k \geq 2$, which gives, e.g., a $(3 + \epsilon, 0)$-approximation guarantee in $\tilde{O}(n^{5/2+O(1/\sqrt{\log n})})$ time. All these algorithms have an $O(1)$ worst case query time. See Section 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

Reference	Total Running Time	Approximation	Deterministic?
Even and Shiloach [23]	$\tilde{O}(mn^2)$	Exact	Yes
This paper	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	Yes
Demetrescu and Italiano [18] and Baswana, Hariharan, and Sen [7]	$\tilde{O}(n^3)$	Exact	No
Roditty and Zwick [33]	$\tilde{O}(mn/\epsilon)$	$(1+\epsilon, 0)$	No
This paper	$\tilde{O}(n^{5/2}/\epsilon^2)$	$(1+\epsilon, 2)$	No
Bernstein and Roditty [12]	$\tilde{O}(n^{5/2+\sqrt{\log(6/\epsilon)}/\sqrt{\log n}})$	$(3+\epsilon, 0)$	No
This paper	$\tilde{O}(n^{5/2}/\epsilon^2)$	$(2+\epsilon, 0)$	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.

runs in $\tilde{O}(n^{5/2}/\epsilon^2)$ 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 $\tilde{O}(mn)$ to $\tilde{O}(n^{5/2})$, at the cost of an additive error of two, which appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be treated as a multiplicative error of $O(\epsilon)$) and is *unavoidable* (as discussed below).
- Our $(2+\epsilon, 0)$-approximation algorithm improves the algorithm of Bernstein and Roditty in terms of both total update time and approximation guarantee. The multiplicative error of $2+\epsilon$ is essentially the best we can hope for, if we do not want any additive error.

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 [23]. These techniques are reviewed in Section 1.3.

Improved Deterministic Algorithm. In 1981, Even and Shiloach [23] presented a deterministic decremental SSSP algorithm for undirected, unweighted graphs with a total update time of $O(mn)$ over all deletions. This was the first dynamic problem studied in theoretical computer science [12]. 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 [33] by a $\log n$ factor.) Our deterministic algorithm also answers a question recently raised by Bernstein [11] which asks for a deterministic algorithm with a total update time of $\tilde{O}(mn/\epsilon)$. As pointed out in [11] 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 [13, 9]) 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.

Improved Randomized Algorithm. Our aim is to improve the $\tilde{O}(mn)$ running time of Roditty and Zwick [33] to so-called *truly subcubic time*, i.e., $O(n^{3-\delta})$ time for some constant $\delta > 0$, a running time that is highly sought of in many problems (e.g., [39, 38, 31][1]). Note, however, that this improvement has to come at the cost of worse approximation:

Fact 1.1 ([19, 39])**.** *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 [19] and Vassilevska Williams and Williams [39] (see Appendix A for a proof sketch). (Roditty and Zwick [32] also showed a similar fact for decremental SSSP.) Thus, the best approximation guarantee we can expect from combinatorial truly subcubic algorithms is, e.g., a multiplicative or additive error of at least two. Our algorithms achieve essentially these *best approximation guarantees*: in $\tilde{O}(n^{5/2})$ 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.[2] We note that, prior to our work, Bernstein and Roditty's algorithm [12] can achieve, e.g., a $(3+\epsilon, 0)$-approximation guarantee in $\tilde{O}(n^{5/2+O(\sqrt{1/\log n})})$ time. This result is improved by our $(2+\epsilon, 0)$-approximation algorithm in terms of both time and approximation guarantee, 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 [33] when $m = \omega(n^{3/2})$, except that our algorithm gives an additive error of two which is unavoidable and appears only when the distance is $O(1/\epsilon)$ (since otherwise it could be counted as a multiplicative error of $O(\epsilon)$).

1.3 Techniques

Our results build on two previous algorithms. The first algorithm is the classic SSSP algorithm of Even and Shiloach [23] (with the more general analysis of King [28]), which we will refer to as *Even-Shiloach tree*. The second algorithm is the $(1+\epsilon, 0)$-approximation APSP algorithm of Roditty and Zwick [33]. We actually view the algorithm of Roditty and Zwick as a *framework* which runs several Even-Shiloach trees and maintains some properties while edges are deleted. We wish to alter the Roditty-Zwick framework but doing so usually makes it hard to bound the cost of maintaining Even-Shiloach trees (as we will discuss later). Our main technical contribution is the development of new variations of the Even-Shiloach tree, called *moving Even-Shiloach tree* and *monotone Even-Shiloach tree*, which are suitable for our modified Roditty-Zwick frameworks. Since there are many other algorithms that run Even-Shiloach trees as subroutines, it might be possible that other algorithms will benefit from our new Even-Shiloach trees as well.

Review of Even-Shiloach Tree. The Even-Shiloach tree has two parameters: a root (or source) node s and the range (or depth) R. It maintains a breadth-first search 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 at most R times before it exceeds R (i.e., from 1 to R). This property heavily relies on the "decrementality" of the model, i.e., the distance between two nodes

[1] Bernstein [11] also recently mentioned a few future directions towards this running time in the decremental setting.

[2] We note that there is still some room to eliminate the ϵ-terms. But nothing beyond this is likely to be possible.

never decreases, 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 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 [33] on an *emulator*, a sparser *weighted* graph that approximates the distances in the original graph (see Section 3.1 for more detail). This approach is commonly used in the static setting (e.g., [1, 19, 20, 22, 5, 14, 15, 35, 40]), and it was recently used for the first time in the decremental setting by Bernstein and Roditty [12]. 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[3]. 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 [33] can be run on their emulator with a small running time. However, they *cannot* run the $(1+\epsilon,0)$-approximation algorithm of Roditty and Zwick on their emulator. The main reason is that this algorithm relies on a more general property of a graph under deletions: for any R between 1 and n, the distance between any two nodes changes at most R times *before it exceeds* R (i.e., it changes from 1 to R). They suggested to find an emulator with this more general property as a future research direction.

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

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

[3] In fact, their emulator is basically identical to one used earlier by Bernstein [10], which is in turn a modification of a spanner developed by Thorup and Zwick [35, 36]. However, the properties they proved are entirely new.

on an emulator anymore. So, even when the emulator gives a good approximate distance on the original graph, our monotone Even-Shiloach tree might not. Our monotone Even-Shiloach tree does not give any guarantee for the distances in the emulator, but we can show that it still approximates the distances in the original graph. Of course, this will *not* work on any emulator; but we can show that it works on a specific type of emulators that we call *locally persevering emulators*.[4] Roughly speaking, a locally persevering emulator is an emulator where, for any "nearby"[5] nodes u and v in the original graph, either

(1) there is a shortest u-v path in the original graph that 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 $\tilde{O}(n^{5/2})$ 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 $\tilde{O}(n^{5/2})$-time $(1+\epsilon, 2)$-approximation algorithm. To turn this algorithm into a $(2+\epsilon, 0)$-approximation, we observe that we can check if two nodes are of distance one easily; thus, we only have to use our $(1+\epsilon, 2)$-approximation algorithm to answer a distance query when the distance between two nodes is at least two. In this case, the additive error of two can be treated as a multiplicative factor.

Proving the Approximation Guarantee of the Monotone Even-Shiloach Tree. To illustrate why our monotone Even-Shiloach tree gives a good approximation when run on a locally persevering emulator, we sketch a result that is weaker and simpler than our main results; we show how to $(3, 0)$-approximate distances from a particular node s to other nodes. This fact easily leads to a $(3+\epsilon, 0)$-approximation $\tilde{O}(n^{5/2})$-time algorithm, which gives the same guarantee as the algorithm of Bernstein and Roditty [12], and is slightly faster and reasonably simpler. To achieve this, we modify the emulator of Dor et al. [19]: Randomly select $\tilde{O}(\sqrt{n})$ 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 their distance. 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: 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] We remark that there are other emulators that can be maintained in the decremental setting, e.g., [35, 36, 33, 10, 12, 2, 21, 8]. We are the first to introduce the notion of locally persevering emulators and show that there is an emulator that has this property.

[5] 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) never decrease.

It follows that this emulator is locally persevering.[6] 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[7]. 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 [12] 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 real 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 v-s path $\langle v, u_1, u_2, \ldots, s\rangle$ 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 $P = \langle v, c, u_1\rangle$ of length at most three in the emulator, and it is persevering (see (ii')). By (O2), *edges in P 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 of 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 emulator 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 $(\alpha+1/\tau, \beta)$-approximate distance. We finally show that our simple $(1+\epsilon, 2)$-emulator is a $(1+\epsilon, 2, 1/\epsilon)$-locally persevering emulator.

[6] 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 define will be $(3, 0, 1)$-locally persevering.

[7] 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 [37, 33]. This even holds in the decremental setting (assuming an oblivious adversary). Like other algorithms, the Roditty-Zwick framework also heavily relies on this argument, which is the only reason why it is randomized. Our goal is to derandomize this argument. Specifically, for several different values of h, the Roditty-Zwick framework selects $\tilde{O}(h)$ 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 each center number j, we let B^j be a "ball" of radius $n/2h$; i.e., B^j is a set of nodes of 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.

However, *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 the point that the second constraint is violated. The key idea that we introduce to avoid this problem is to allow a center and the Even-Shiloach tree rooted at it to *move*. We call this a *moving Even-Shiloach tree* or *moving center* data structure. Specifically, in the moving Even-Shiloach tree, we view a root (center) s *not* as a node, but as a *token* that can be placed on any node, and the task of the moving Even-Shiloach tree is to maintain the distance between the node that the root is placed on and all other nodes, up to distance R. We allow a *move operation* where we can move the root to a new node and the corresponding Even-Shiloach tree must be adjusted accordingly. To illustrate the power of the move operation, consider the following simple modification of Algorithm 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 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 a 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 the same 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 to nodes in C^j; in particular, we bound the total moving distance to be $O(n)$ by showing that the moving distance of center j can be bounded by $|C^j|$, and C^j and $C^{j'}$ are disjoint for distinct centers j and j'. The other important changes are the definitions of "ball" and "small connected component" which will now depend on C^j.

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

These new definitions might not be intuitive, but they are crucial for the charging argument. We also have to modify Algorithm 1.3 in a counter-intuitive way: the most important modification is that we have to give up the nice property that the distance between any two center 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 if there is a center j that is in a "small" connected component X (of size less than $(n/2h) - |C^j|$). If there is such a j (there will be at most one such j), move it out of X to a new node which is the unique node in $\{u, v\} \setminus X$. After moving, execute the static algorithm as in Algorithm 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 Algorithm 1.3 and Algorithm 1.4. However, using C^j, we can still show something very similar: We can show that $B^j \cup C^j$ and $B^{j'} \cup C^{j'}$ are disjoint for distinct j and j', and that $|B^j \cup C^{j'}| \geq n/2h$. 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 [29, 30][8]. It also has a tight connection with its *static* counterpart (where the graph does not change), which is one of the most fundamental problems in computer science: On the one hand, we wish to devise a dynamic algorithm that beats the naive algorithm where we recompute shortest paths *from scratch* using static algorithms after every deletion. On the other hand, the best we can hope for is to match 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 non-combinatorial algorithms based on matrix multiplication). In fact, a faster combinatorial algorithm will be a *major breakthrough*, not just because computing shortest paths is a long-standing problem by itself, but also because it will imply faster algorithms for other long-standing problems, as stated in Fact 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 that beats the naive $O(m^2n)$-time algorithm is from 1981 by Even and Shiloach [23], for the special case of SSSP. They showed an $O(mn)$ total update time with $O(1)$ query time; this implies a total update time of $O(mn^2)$ for APSP. Roditty and Zwick [32] later provided evidence that the $O(mn)$-time decremental unweighted SSSP

[8]The early papers [29, 30], however, were not able to beat the naive algorithm where we compute APSP from scratch after every change.

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

algorithm of Even and Shiloach is the fastest possible by showing that this is at least as hard as several natural static problems such as Boolean matrix multiplication and the problem of finding all edges of a graph that are contained in triangles. For the incremental setting, Ausiello, Italiano, Marchetti-Spaccamela, and Nanni [3] 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 [4].) After that, many efficient fully-dynamic algorithms have been proposed (e.g., [26, 28, 24, 18, 16]). Subsequently, Demetrescu and Italiano [17] achieved a major breakthrough for the fully dynamic case: they obtained a fully dynamic deterministic algorithm for the 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 [34] 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 [17], 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 [18] obtained an exact decremental algorithm on weighted directed graphs with $\tilde{O}(n^3)$ total update time[10] (if weight increments are not considered). Baswana, Hariharan, and Sen [7] 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 [6], 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 [33] 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 achieved a worse approximation bound of $(2k-1, 0)$, for any $2 \leq k \leq \log n$, which has the advantage of requiring less space ($\tilde{O}(mn + n^{1/k})$). By modifying the second algorithm to work on an emulator, Bernstein and Roditty [12] 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 [11] 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 [19], extending the approach of Aingworth et al. [1], 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 [20] 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.

[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 [18, Theorem 10]). Also note that the algorithm is randomized with one-sided error.

2 Background

2.1 Graph-theoretic Definitions

In the following we fix some basic notation and definitions that will be used in the rest of this paper.

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* $m = |E(G_0)|$*. 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* $(x, y) \in E(H_i)$*, the weight of* (x, y) *is given by* $w_i(x, y)$*.*

Definition 2.2 (Decremental graph)**.** *A* decremental graph $\mathcal{G}$ *is a dynamic graph* $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ *such that for every* $0 \leq i < k$ *the graph* G_{i+1} *is the result of deleting exactly one edge from the graph* G_i*. Note that* G_i *is the graph after the* i*-th edge deletion. A* decremental weighted graph $\mathcal{H}$ *is a dynamic weighted graph* $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ *such that for every* $0 \leq i < k$ *the weighted graph* (H_{i+1}, w_{i+1}) *is either the result of deleting exactly one edge from the weighted graph* (H_i, w_i) *or the result of increasing the weight of exactly one edge in the weighted graph* (H_i, w_i)*.*

We now 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 changes from G_{i-1} to G_i. In the case of a decremental graph $\mathcal{G}$ the algorithm will for example be informed which edge was deleted. After the initialization phase and after each update phase, the algorithm has to be able to answer queries. In our case, these queries will usually be distance queries and the algorithm will answer them in constant or near-constant time. The *total update time* of the algorithm is the total time spent for processing the initialization and *all* k updates. Note that 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 typical amortization argument used for these algorithms.

Definition 2.3 (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 2.4 (Degree)**.** *The degree of a node* x *in the graph* G *is denoted by* $\deg_G(x)$*. The maximum degree of a node* x *in a dynamic graph* $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ *is denoted by* $\deg_{\mathcal{G}}(x) = \max_{0 \leq i \leq k} \deg_{G_i}(x)$*. We say that* v is a neighbor of u *if there is an edge* (u, v) *in* G*.*

Definition 2.5 (Paths)**.** *Let* $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ *be a weighted directed graph and let* P *be a path in* (H_i, w_i) *for some* $i \leq k$*. The number of nodes on the path* P *is denoted by* $|P|$ *and the total weight of the path (i.e., the sum of the weights of its edges) is denoted by* $w_i(P)$*.*

Definition 2.6 (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 single-source shortest paths tree introduced by Even and Shiloach, short *ES-tree*. Even and Shiloach [23] developed this data structure for undirected, unweighted graphs and later on, King [28] 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 for a given root node r and a given distance range parameter R^{d}. The data structure can handle arbitrary edge weight increases. Edge deletions can be done by setting the weight to ∞. The pseudocode can be found in Algorithm 1. The data structure consists of a shortest paths tree up to distance R^{d} rooted at r and for every node x a label $\ell(x)$, called the *level* of x. The level of x corresponds to the distance of x to the root r in the tree. The shortest paths tree contains nodes whose distance to the root r in the current graph is at most R^{d}. All other nodes are not connected to the tree and their level is set to ∞. Furthermore, for every node x the algorithm 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. These data structures can be initialized by running, for example, Dijkstra's algorithm.

Every weight increase of an edge (u, v) possibly affects the nodes in the subtree of of v. Nodes in the subtree that are affected in a certain way lose their current tree edge. The algorithm tries to reconnect these nodes to the tree in a way that is similar to Dijkstra's algorithm. 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.Therefore every node y that loses a tree edge to its parent will repeatedly update its level to $\ell'(y)$ (unless its level already has this value). If this updating rule leads to a level increase, the algorithm has to update the heap $N(x)$ of every node x that has an edge to y and also has to delete the tree edges to the children of y.

The algorithm uses a heap H to process the updates of the nodes in the order of their current level. By this coordination process it can be guaranteed that if $\ell'(y) = \ell(y)$, then every neighbor z providing the minimal level $\ell'(y) = \ell(z) + w(y, z)$ can be used to reconnect to the tree. A special case is the situation when the level of a node x exceeds R^{d}. In this case the level of x is set to ∞ and x will never be connected to the tree again.

In line 12 of Algorithm 1, the reader might expect the equation $\ell(y) = \ell(z) + w(y, z)$ instead of the inequality $\ell(y) \geq \ell(z) + w(y, z)$. In fact, in Algorithm 1 this inequality can only hold with equality. The reason for stating the algorithm like this is our modification in Section 3.2, called *monotone ES-tree*, where we add a procedure for inserting edges. In the monotone ES-tree algorithm it might indeed happen that the difference in levels $\ell(y) - \ell(z)$ between a node y and its parent z in the tree is more than the weight of the edge (y, z).

The running time analysis takes into account the level increases occurring in the algorithm. It is based on the following observation for every node x processed in the while-loop of the procedure RECONNECT in Algorithm 1. If the level of x increases, the algorithm has to spend time $O(\deg(x) \log n)$ for updating the heaps of all neighbors and adding its children to the heap H. 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 either be charged to the level increase of the previous parent of x in the tree or, if the edge of x to its parent was the deleted edge (u, v), it can be charged to the deletion of the edge (u, v). For simplicity we have stated the algorithm using heaps. King [28] explains how to avoid heaps in order to improve the running time by a factor of $\log n$. Using this modification, the algorithm spends time $O(\deg(x))$ per level increase of a node x.

Lemma 2.7 ([28]). *Algorithm 1 maintains shortest paths in a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ to a root node r up to distance R^d. It has constant query time and its total update time for k delete*

Algorithm 1 ES-tree algorithm

```
1: procedure DELETE(u, v)
2:     RECONNECT(u, v, ∞)
3: procedure INCREASE(u, v, w'(u, v))
4:     Remove tree edge (u, v)
5:     Update value of v in heap N(u) of u to ℓ(v) + w'(u, v)
6:     Put u into heap H with value ℓ(u)
7:     RECONNECT( )
8: procedure RECONNECT( )
9:     while heap H is not empty do
10:        Take node y with minimal value ℓ(y) from heap H (and remove y from H)
11:        ℓ(y) = min_z(ℓ(z) + w(y, z)) (can be retrieved from the heap N(y) of y)
12:        if ℓ(y) ≥ ℓ'(y) then
13:            z = arg min_z(ℓ(z) + w(y, z)) (can be retrieved from the heap N(y) of y)
14:            Make z the parent of y in the tree
15:        else
16:            ℓ(y) ← ℓ'(y)
17:            if ℓ(y) > R^d then
18:                ℓ(y) ← ∞
19:            else
20:                Put y into heap H with value ℓ(y)
21:            for every edge (x, y) do update value of y in heap N(x) to ℓ(y) + w(x, y)
22:            for every child x of y in the tree do
23:                Remove tree edge (x, y)
24:                Put x into heap H with value ℓ(x)
```

or increase operations is

$$O\left(\sum_{0\leq i<k}\sum_{x\in V}\deg_{G_i}(x)(\ell_{i+1}(x)-\ell_i(x))\right)$$

where, for $0 \leq i \leq k$, $\ell_i(x)$ is the level of x after the algorithm has processed the i-th edge deletion or edge weight increase.

As a simple corollary of this result we can express the running time of the algorithm in terms of the initial and the final level of every node.

Corollary 2.8. *There is a decremental exact SSSP algorithm that maintains shortest paths in a decremental graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$ to a root node r up to distance R^d. It has constant query time and its total update time for k delete operations is*

$$O\left(\sum_{x\in V}\deg_{G_0}(x)\cdot\Big(\min(R^d,d_{G_k}(x,r))-\min(R^d,d_{G_0}(x,r))\Big)\right)$$

where $d_{G_0}(x,r)$ is the initial distance of r to x and $d_{G_k}(x,r)$ is the distance of x to r after all k edge deletions.

The observation above is simple but we will use it in a powerful way in Section 4. It will allows to bound the running time of Even-Shiloach trees that are moved to a different root nodes over time.

2.3 The Framework of Roditty and Zwick

In the following we review the algorithm of Roditty and Zwick [33] 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. For every graph G, all nodes x, y, and z, and every $\epsilon \leq 1$, such that $2^p \leq d_G(x,y) \leq 2^{p+1}$ and $d_G(x,z) \leq \epsilon 2^p$ we have $d_G(x,y) \leq d_G(x,z) + d_G(z,y) \leq (1+2\epsilon)d_G(x,y)$ and $d_G(z,y) \leq 2^{p+2}$. Thus, we can use $d_G(x,z)+d_G(z,y)$ as a $(1+2\epsilon)$-approximation of the real distance $d_G(x,y)$. Under the assumption that all distances we are interested in are in the range from 2^p to 2^{p+1} we only have to maintain a set U of nodes such that for every node x there is a node $z \in U$ such that $d_G(x,z) \leq \epsilon 2^p$. We call such a set of nodes U a *center cover* (see Definition 2.10 below). Furthermore, for every node $z \in U$ and for every node y such that $y \leq 2^{p+2}$ the distance $d_G(z,y)$ has to be known. We have summarized these requirements in a data structure.

Definition 2.9 (Center cover)**.** *Let U be a set of nodes of a graph G and let R^c be a positive integer, the* cover range*. We say that a node x is* covered *by a node $c \in U$ 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 $c \in U$.*

Definition 2.10 (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 $\mathcal{G} = (G_i)_{0\leq i\leq k}$ maintains a set of* centers $C = \{1,2,\ldots,l\}$ *and a set of nodes $U = \{c^1, c^2, \ldots, c^l\}$ such that, for every $0 \leq i \leq k$, U is a center cover of G_i with parameter R^c. For every center $j \in C$ we call $c^j \in U$ the* location *of center j and for every node x we say that x is covered by j if x is covered by c^j. The data structure provides the following update operations:*

- INITIALIZE(G_0, R^c, R^d): *Initialize the data structure*
- DELETE(u, v): *Delete the edge* (u, v) *from the graph*

The data structure provides the following query operations, applied on the current graph G_i *after the* i*-th edge deletion:*

- DISTANCE(j, x): *This procedure returns the distance between the location* c^j *of center* j *and the node* x*, provided that* $d_{G_i}(c^j, x) \leq R^d$*. If* $d_{G_i}(c^j, x) > R^d$*, then the procedure returns* ∞*.*
- FINDCENTER(x) *If* x *is in a connected component of size at least* R^c*, this procedure returns a center* j *(with location* c^j*) such that* $d_{G_i}(x, c^j) \leq R^c$*. If* x *is in a connected component of size less than* R^c*, then this procedure either returns* $\perp$ *or it returns a center* j *(with location* c^j*) such that* $d_{G_i}(x, c^j) \leq R^c$*.*

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 $l = \tilde{O}(n/R^{\mathrm{c}})$. In any deterministic center cover data structure with $o(n)$ centers the location of centers has to change over time as an adversary can delete all edges adjacent to the centers, making all non-center nodes uncovered. For the randomized algorithm of Roditty and Zwick this is not necessary because they can initially pick a set of nodes that will remain a center cover during all deletions with high probability.

Given a center cover data structure, an approximate decremental APSP algorithm is obtained as follows. Consider $\log n$ layers where the p-th layer is responsible for the distance range from 2^p to 2^{p+1}. For each layer maintain a center cover data structure using the parameters $R^{\mathrm{c}} = \epsilon 2^p$ and $R^{\mathrm{d}} = 2^{p+2}$. To answer a query for the distance between x and y one only has to find the layer providing the minimal distance estimate. This can be done in time $O(\log \log n)$ by a binary search for the right index p.

Theorem 2.11 ([33])**.** *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* $T(n, m, R^c, R^d)$*. Then, for every* $\epsilon \leq 1$*, there is a* $(1+\epsilon, 0)$*-approximate decremental APSP algorithm with* $O(\log \log n)$ *query time and a total update time of* $\sum_p T(n, m, R_p^c, R_p^d)$ *where* $R_p^c = \epsilon 2^p$ *and* $R_p^d = 2^{p+2}$ *(for* $0 \leq p \leq \log n$*).*

Roditty and Zwick [33] 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 2.11 they get a $(1+\epsilon, 0)$-approximate decremental APSP algorithm with a total update time of $\tilde{O}(mn)$ and a query time of $O(\log \log n)$.[11] To analyze the total update time of their algorithm for k deletions observe that

$$\sum_{p=0}^{\log n} \tilde{O}(mnR_p^{\mathrm{d}}/R_p^{\mathrm{c}}) = \sum_{p=0}^{\log n} \tilde{O}(mn2^{p+1}/(2^p\epsilon)) = \sum_{p=0}^{\log n} \tilde{O}(mn/\epsilon) = \tilde{O}(mn \log n/\epsilon) = \tilde{O}(mn/\epsilon)\,.$$

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 algorithm with a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$ and constant query time. In Section 4 we show how to maintain an exact *deterministic* center cover data structure with total update time $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. By Theorem 2.11 this immediately implies a deterministic $(1+\epsilon, 0)$-approximate decremental APSP algorithm with a total update time of $O(mn \log n)$ and a query time of $O(\log \log n)$.

[11] They can even reduce the query time to $O(1)$ by using a second $(3, 0)$- approximate decremental APSP algorithm.

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

In this section, we present a dynamic algorithm for maintaining all-pairs shortest paths under edge deletions with multiplicative error $1 + \epsilon$ and additive error 2 that has a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$. The algorithm is correct with high probability. We also show a variant of this algorithm 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 show the locally persevering emulator that we will use to obtain our result. Then, in Section 3.2 we explain our main technique, called *monotone Even-Shiloach tree*, where we maintain the distances from a single node to all other nodes, up to some distance R^c, in a locally persevering emulator. Finally, in Section 3.3, we show how to put the results in Sections 3.1 and 3.2 together to obtain the desired $(1 + \epsilon, 2)$- and $(2, 0)$-approximation algorithms.

Definition 3.1 (Persevering path)**.** *Let $\mathcal{H} = (H_i, w_i)_{0\leq i\leq k}$ be a dynamic weighted graph. We say that a path P is* persevering up to time t *(where $t \leq k$) if for every edge (u, v) on P and, for all $0 \leq i \leq t$, $(u, v) \in E(H_i)$ and, for all $0 \leq i < t$, $w_i(u, v) \leq w_{i+1}(u, v)$. In other words, edges in P always exist in $\mathcal{H}$ up to time t and their weights never decrease.*

We now introduce the notion of a *locally persevering emulator*. To motivate the definition, we note that an (α, β)-emulator of a dynamic graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$ is usually used to refer to another dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\leq i\leq k}$ over the same set of nodes that preserves the distance of the original dynamic graph, i.e., for all $i \leq k$ and all nodes x and y, there is a path P in H_i such that $d_{G_i}(x, y) \leq w_i(P) \leq \alpha d_{G_i}(x, y) + \beta$. The notion of a *locally persevering* emulator puts an additional restriction on such paths. In particular, this concerns nodes x and y such that $d_{G_i}(x, y) \leq \tau$ for some parameter τ. We demand that the path P must be either a shortest path in G_i or a persevering path.

Definition 3.2 (Locally persevering emulator)**.** *Consider parameters $\alpha \geq 1$, $\beta \geq 0$ and $\tau \geq 1$, a dynamic graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$, and a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0\leq i\leq k}$. For every $i \leq k$, we say that a path P in G_i is* contained *in (H_i, w_i) if every edge of P 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 every $0 \leq i \leq k$ and for all nodes x and y we have $d_{G_i}(x, y) \leq d_{H_i,w_i}(x, y)$ and if $d_{G_i}(x, y) \leq \tau$, then additionally one of the following holds: either a shortest path P from x to y in G_i is contained in (H_i, w_i), or there is a path P' from x to y in $\mathcal{H}$ that is persevering up to time i and satisfies $w_i(P') \leq \alpha d_{G_i}(x, y) + \beta$.*

As a last preliminary step, we introduce a measure of how much a dynamic graph changes over time. We will use this measure to quantify how often an emulator changes its edges. This will be important because the running time we obtain for our modification of ES-trees will depend on this number.

Definition 3.3. *Let $\mathcal{H} = (H_i, w_i)_{0\leq i\leq k}$ be a dynamic weighted graph. For every pair of nodes x and y we define $\phi(x, y, \mathcal{H})$ to be the total number of insertions, deletions, and edge weight increases of an edge (x, y) over all graphs (H_i, w_i). We define $\phi(\mathcal{H}) = \sum_{x,y\in V} \phi(x, y, \mathcal{H})$ to be the* number of changes in $\mathcal{H}$.

3.1 $(1 + \epsilon, 2, 2/\epsilon)$-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 $m = \tilde{O}(n^{3/2})$ edges.

However, to be technically correct, we have to use the stronger fact that the sum of the "maximum degrees" of all nodes is $\tilde{O}(n^{3/2})$, as in the following statement.

Lemma 3.4 (Existence of $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator of size $\tilde{O}(n^{3/2})$)**.** *For every $\epsilon \leq 1$ and every decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$, there is data structure that can maintain a dynamic weighted graph $\mathcal{H} = (H_i, w_i)_{0 \leq i \leq k}$ in $O(mn^{1/2} \log n/\epsilon)$ total time such that $\mathcal{H}$ is a $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator with high probability. Moreover, the sum of the maximum degrees of nodes in $\mathcal{H}$ is $\sum_{x \in V} \deg_{\mathcal{H}}(x) = O(n^{3/2} \log n)$ and the total number of changes in $\mathcal{H}$ is $\phi(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.*

We construct an emulator $\mathcal{H}$ as follows. Pick a set D of $10\sqrt{n} \log n$ nodes uniformly at random. We call the nodes in D *dominators*. For every $i \leq k$, the graph H_i contains the following two types of edges. For every dominator $x \in D$ and every node y such that $d_{G_i}(x, y) \leq 2/\epsilon + 1$, H_i contains an edge 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.[12]

In the following we argue that the dynamic graph $\mathcal{H}$ described above has the desired properties. We start by bounding the initial number of edges of $\mathcal{H}$ and the number of edges that are ever inserted into $\mathcal{H}$. These are the most important quantities for the analysis of the emulator.

Observation 3.5. *The number of edges in H_0 is $|E(H_0)| = O(n^{3/2} \log n)$.*

Proof. We can charge each edge in H_0 either to a dominator or to a node with degree at most $\sqrt{n}$. For every dominator $x \in D$ there might be $O(n)$ edges adjacent to x in H_0. Since there are $O(n^{1/2} \log n)$ many dominators, the number of edges charged to dominators is $O(n^{3/2} \log n)$. For node with degree at most $\sqrt{n}$ there are $O(n^{1/2})$ 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

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

Observation 3.6. *The number of edges inserted into $\mathcal{H}$ over all deletions in $\mathcal{G}$ is $O(n^{3/2})$.*

Proof. Every time the degree of a node x changes from $\deg_{G_i} > n^{1/2}$ to $\deg_{G_{i+1}} = n^{1/2}$ (for some $0 \leq i < k$) we insert all $n^{1/2}$ edges adjacent to x in G_{i+1} into H_{i+1}. In the decremental setting it can happen at most once for every node that the degree of a node drops to $n^{1/2}$. Therefore at most $n^{3/2}$ edges are inserted in total. $\square$

In order to prove Lemma 3.4 we have to prove four properties of $\mathcal{H}$:

- $\mathcal{H}$ is a $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator of $\mathcal{G}$.
- $\mathcal{G}$ can be maintained in total time $O(mn^{1/2} \log n/\epsilon)$ for k deletions in $\mathcal{G}$.
- The sum of the maximum degrees in $\mathcal{H}$ is $O(n^{3/2} \log n)$.
- The number of changes in $\mathcal{H}$ is $\phi(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.

We prove these four properties in the following.

[12] For readers who are familiar with the emulator literature, we note that our construction is very similar to the classic $(1, 2)$-emulator given by Dor, Halperin, and Zwick [19]. 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 is that the $(1, 2)$-emulator of Dor et al. cannot be maintained in $\tilde{O}(mn)$ time under edge deletions.

Lemma 3.7 (Locally persevering)**.** *The dynamic graph $\mathcal{H}$ described above is a $(1+\epsilon, 2, 2/\epsilon)$-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 $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator of $\mathcal{G}$. Assume that $d_{G_t}(x,y) \leq 2/\epsilon$ and no shortest path from x to y in G_t is also contained in (H_t, w_t). Let P be an arbitrary shortest path from x to y in G_t. Since P is not contained in H_t, there must be some edge (u,v) on P such that $(u,v) \notin E(H_t)$. This can only happen if u has degree more than $\sqrt{n}$ in G_t. We now note the fact that with high probability u has a neighbor $z \in D$ in G_t (see, e.g., [37, 19]); i.e., there is an edge (u,z) in G_t of u to a dominator z. Now consider any $i \leq t$. Note that $d_{G_i}(x,z) \leq d_{G_t}(x,z)$ because distances never increase in the decremental setting. By the triangle inequality we get

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

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. For the same reason H_i 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 P' in H_t consisting of the edges (x,z) and (y,z). Since both edges are persevering up to time t, also the path P' is persevering up to time t. Furthermore, P' guarantees the desired approximation:

$$\begin{aligned} w_t(P') = w_t(x,z) + w_t(z,y) &= d_{G_t}(x,z) + d_{G_t}(z,y) \\ &\leq d_{G_t}(x,u) + d_{G_t}(u,z) + d_{G_t}(z,u) + d_{G_t}(u,y) \\ &= d_{G_t}(x,u) + d_{G_t}(u,y) + 2 \\ &= d_{G_t}(x,y) + 2 \leq (1+\epsilon) d_{G_t}(x,y) + 2\,. \end{aligned}$$

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 $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator of $\mathcal{G}$. □

Lemma 3.8 (Running time)**.** *The dynamic graph $\mathcal{H}$ described above can be maintained in total time $O(mn^{1/2} \log n/\epsilon)$ over all k edge deletions in $\mathcal{G}$.*

Proof. We use the following data structures: For every node $x \in D$ we maintain an Even-Shiloach tree (see Section 2.2) rooted at x of depth $2/\epsilon + 1$. Furthermore, for every node x, we maintain the degree of x in $\mathcal{G}$. We also use an adequate graph representation that allows us to store and change the edges of $\mathcal{H}$ and their weights in constant time per update.

We now explain how to process the i-the edge deletion of, say, the edge (u,v). First of all we decrease 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 spend time $\sqrt{n}$ for inserting all edges of u (or v) in G_i into H_i. As this happens at most once for every node in the decremental graph $\mathcal{G}$, we spend total time $O(n^{3/2})$ for inserting edges.

After this procedure, for every node $x \in D$, we do the following: First of all, we report the deletion of (u,v) to the Even-Shiloach tree rooted at x. Every node y has a level in this Even-Shiloach tree. If the level of y increases to ∞, then $d_{G_i}(x,y) > 2/\epsilon + 1$ and therefore we remove the edge (x,y) from $\mathcal{H}$. If the level of y increases, but does not reach ∞, then $d_{G_i}(x,y) \leq 2/\epsilon + 1$

and we update the weight of the edge (x, y) in $\mathcal{H}$. The constant time needed for deleting (x, y) from H_i or increasing the weight of (x, y) in H_i can be charged to the level increase in the Even-Shiloach tree and therefore does not have to be considered separately in the running time analysis. Maintaining the Even-Shiloach tree takes total time $O(m/\epsilon)$ for each dominator (see Section 2.2). Since there are $O(n^{1/2} \log n)$ dominators, the total time for maintaining all Even-Shiloach trees is $O(mn^{1/2} \log n/\epsilon)$ in total.

Thus, the total update time for maintaining the emulator is $O(n\sqrt{n} + mn^{1/2} \log n/\epsilon)$, which is $O(mn^{1/2} \log n/\epsilon)$. □

Lemma 3.9 (Sum of degrees)**.** *For the dynamic graph $\mathcal{H}$ described above we have $\sum_{x \in V} \deg_{\mathcal{H}}(x) = O(n^{3/2} \log n)$.*

Proof. Let $I(\mathcal{H})$ denote the set of edges that are inserted into the emulator $\mathcal{H}$, i.e.,

$$I(\mathcal{H}) = \{(u, v) \mid (u, v) \in E(H_{i+1}) \text{ and } (u, v) \notin E(H_i) \text{ for some } 0 \leq i < k\}$$

and, for every node x let $I(x)$ denote the set of edges that are inserted into $\mathcal{H}$ and are adjacent to x, i.e.,

$$I(x) = \{(x, y) \mid (x, y) \in I(\mathcal{H})\}.$$

Clearly, $I(\mathcal{H}) = \bigcup_{x \in V} I(x)$ and $\sum_{x \in V} |I(x)| \leq 2|I(\mathcal{H})|$.

We first show that $\sum_{x \in V} \deg_{\mathcal{H}}(x) \leq 2|E(H_0)| + 2|I(\mathcal{H})|$. For every node x and every $i \leq k$ the edges adjacent to x in H_i were either already contained in H_0 or have been inserted into $\mathcal{H}$. Therefore we have $\deg_{H_i}(x) \leq \deg_{H_0}(x) + |I(x)|$. Since $\deg_{\mathcal{H}}(x)$ is the maximum of all $\deg_{H_i}(x)$, we get $\deg_{\mathcal{H}}(x) \leq \deg_{H_0}(x) + |I(x)|$. In total, it follows that

$$\sum_{x \in V} \deg_{\mathcal{H}}(x) \leq \sum_{x \in V} \deg_{H_0}(x) + \sum_{x \in V} |I(x)| \leq 2|E(H_0)| + 2|I(\mathcal{H})|.$$

Recall that by Observation 3.5 we have $|E(H_0)| = O(n^{3/2} \log n)$ and by Observation 3.6 we have $|I(\mathcal{H})| = O(n^{3/2})$. Therefore we get

$$\sum_{x \in V} \deg_{\mathcal{H}}(x) \leq 2|E(H_0)| + 2|I(\mathcal{H})| = O(n^{3/2} \log n) + O(n^{3/2}) = O(n^{3/2} \log n).$$ □

Lemma 3.10 (Number of changes)**.** *The number of changes in the dynamic graph $\mathcal{H}$ described above is $\phi(\mathcal{H}) = O(n^{3/2} \log n/\epsilon)$.*

Proof. The following kinds of changes appear in $\mathcal{H}$: edge insertions, edge deletions, and edge weight increases. By Observation 3.6 the total number of edge insertions is $O(n^{3/2})$ and by Observation 3.5 the initial number of edges is $|E(H_0)| = O(n^{3/2} \log n)$. Every deleted edge is either an edge contained in H_0 or an edge that has been inserted. Therefore the total number of edge deletions is $O(n^{3/2} + n^{3/2} \log n) = O(n^{3/2} \log n)$.

Finally, we bound the number of edge weight increases. The only weighted edges are those adjacent to dominators. The maximum weight of these edges is $2/\epsilon + 1$ and the minimum weight is 1. Therefore the weight of such an edge can increase at most $2/\epsilon + 1$ times. As there are $O(n^{1/2} \log n)$ dominators, each having $O(n)$ weighted edges, the total number of edge weight increases is $O(n^{3/2} \log n/\epsilon)$.

By summing up the three quantities for edge insertions, edge deletions, and edge weight increases we can bound the total number of changes as follows:

$$\phi(\mathcal{H}) = O(n^{3/2} + n^{3/2} \log n + n^{3/2} \log n/\epsilon) = O(n^{3/2} \log n/\epsilon).$$ □

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 main result of this section is Lemma 3.11 below. The idea is to maintain an extension of the ES-tree (as described in Section 2.2) on a locally persevering emulator $\mathcal{H}$ of a graph $\mathcal{G}$. We call our modification of the ES-tree *monotone ES-tree* by the "monotone" way in which the algorithm handles edge insertions. Because the emulator is locally persevering, the monotone ES-tree provides an approximation of the distances in $\mathcal{G}$.

Lemma 3.11 (Monotone ES-tree)**.** *Let $\mathcal{H} = (H_i, w_i)_{0 \le i \le k}$ be a (α, β, τ)-locally persevering emulator of a dynamic graph $\mathcal{G} = (G_i)_{0 \le i \le k}$, for some parameters α, β, and τ. Let r be an arbitrary node (called* root*) and let R^d be an integer, the* distance range parameter*. There is a data structure, called* monotone Even-Shiloach tree (monotone ES-tree)*, for maintaining $(\alpha+\beta/\tau, \beta)$-approximate single-source shortest paths to r under k deletions that provides the following operations:*

- Delete(u, v)*: Delete edge (u, v) from the current graph G_i*
- Distance(x)*: This procedure returns a distance estimate $\delta(x, r)$ of the distance between x and the root r in the current graph G_i such that $\delta(x, r) \ge d_{G_i}(x, r)$. If $d_{G_i}(x, r) \le R^d$, then also $\delta(x, r) \le \alpha d_{G_i}(x, r) + \beta$. If $d_{G_i}(x, r) > R^d$, then the procedure either returns $\delta(x, r) = \infty$ or it returns $\delta(x, r) \le \alpha d_{G_i}(x, r) + \beta$.*

The total update time for k deletions is $O(\phi(\mathcal{H}) + \sum_{x \in V} \deg_{\mathcal{H}}(x) R^d \log n)$ plus the time needed to maintain $\mathcal{H}$.

Before we start explaining the algorithm we clarify a crucial detail about the order of update operations 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 changes 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.[13] We will pretend that these changes have a specific order. In particular, we will present the update operations on (H_i, w_i) to our monotone ES-tree algorithm in the following order: *First*, we process the edge insertions, one after the other. Afterwards, we process the edge deletions and edge weight increases (also one after the other). This order is crucial for the correctness of our algorithm.

Implementation. 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. 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 dynamic graph $\mathcal{G}$.

The overall algorithm now is as follows. We maintain an ES-tree rooted at r up to depth $(\alpha+\beta/\tau)R^{\mathrm{d}} + \beta$ on the graph $\mathcal{H}$ (see Algorithm 1). This ES-tree itself cannot deal with insertions. We now give a separate procedure for processing the insertion of an edge (u, v) (see Algorithm 2). The current level $\ell(u)$ of u is compared to the level $\ell'(u) = \ell(v) + w(u, v)$ that could be achieved

[13] 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.

by connecting u to v in the tree using the new edge (u, v). If $\ell'(u) < \ell(u)$, then the algorithm makes v the parent of u in the tree, otherwise the tree is not changed. In either case the level of u is not changed. In the case $\ell'(u) < \ell(u)$ the algorithm also updates the value of v in the heap $N(u)$ of u. Edge deletions and edge weight increases are handled just like in the usual ES-tree (see Algorithm 1).

Algorithm 2 Monotone ES-tree

```
1: procedure INSERT(u, v)
2:     if ℓ(v) + w(u, v) < ℓ(u) then
3:         Make v the parent of u in the tree
4:         Update value of v in heap N(u) of u to ℓ(v) + w(u, v)
5: procedure DELETE(u, v)
6:     INCREASE(u, v, ∞)
7: procedure INCREASE(u, v, w'(u, v))
8:     Remove tree edge (u, v)
9:     Update value of v in heap N(u) of u
10:    Put u into heap H with value ℓ(u)
11:    RECONNECT( )
12: procedure RECONNECT( )
13:    Exactly the same as in Algorithm 1 except that the maximum finite level is (α+β/τ)R^d+β
```

Analysis. We first argue about the correctness of the monotone ES-tree and afterwards argue about its running time. In the following we let $\ell_i(u)$ be the level of u in the monotone ES-tree after it has processed the i-th edge deletion in $\mathcal{G}$ (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 changes caused by the i-th deletion in $\mathcal{G}$. We say that a node u is *stretched* if $\ell_i(u) > \ell_i(v) + w_i(u, v)$ for some edge $(u, v) \in E(H_i)$. Note that for a node u that is not stretched we have $\ell_i(u) \leq \ell_i(v) + w_i(u, v)$ for every edge $(u, v) \in E(H_i)$.

Our analysis uses four simple observations about the algorithm.

Observation 3.12. *The following holds for the monotone ES-tree:*

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

We now want to 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 x is a $(\alpha + \beta/\tau, \beta)$-approximation of the distance of x to the root. Remember that processing an edge deletion in $\mathcal{G}$ might mean processing a series of changes in $\mathcal{H}$. We will first show that the approximation guarantee holds for every node that is *stretched* after the algorithm has processed the i-th deletion. Afterwards we will show that it holds for *every* node.

Lemma 3.13. *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. 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 our algorithm sees the changes 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) adjacent to x such that x is stretched since this insertion (see Observation 3.12). Let $\ell'(x)$ denote the level of x after the insertion of (x, y) has been processed. By Observation 3.12, nodes do not change their level as long as they are stretched and therefore $\ell_i(x) = \ell'(x)$. Either the insertion of (x, y) was caused by the deletion of the i-th edge or it was caused 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. All insertions caused the i-th deletion are processed in a row and *before* any other changes to the emulator are processed. Since edge insertions do not change the level of any node and the level of the stretched node x is not changed we have $\ell'(x) = \ell_{i-1}(x)$. In both cases we have $\ell'(x) = \ell_{i-1}(x)$ and therefore, by the assumptions of the lemma,

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

Furthermore, $d_{G_{i-1}}(x, r) \leq d_{G_i}(x, r)$ because in the decremental setting distances never decrease. Therefore we get $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x, r) + \beta$ for every stretched node x, as desired. □

In order to prove the approximation guarantee also for non-stretched nodes we have to exploit the properties of the (α, β, τ)-locally persevering emulator $\mathcal{H}$. We first show a useful property of persevering paths: If two nodes are connected by a persevering path in (H_i, w_i), then their levels differ by at most the length of this path.

Lemma 3.14. *For every path P from a node x to a node y that is persevering up to time i we have $\ell_i(x) \leq \ell_i(y) + w_i(P)$.*

Proof. We show by induction on i for every edge (u, v) on P we have $\ell_i(u) \leq \ell_i(v) + w_i(u, v)$. The lemma then clearly follows by repeated applications of this inequality.

By the definition of persevering paths, every edge (u, v) is contained in (H_i, w_i). Consider first the case that u with level $\ell_i(u)$ is not stretched. Then, by the definition of being stretched, we have $\ell_i(u) \leq \ell_i(v) + w_i(u, v)$. Now consider the case that u is stretched. Let $\ell'(u)$ and $\ell'(v)$ denote the levels of u and v directly before u changes from not being stretched to being stretched (this has to happen because right after the initialization no node is stretched). Let $w'(u, v)$ denote the weight of the edge (u, v) directly before the change. Since u is non-stretched before the change, we have $\ell'(u) \leq \ell'(v) + w'(u, v)$ By Observation 3.12, u does not change its level as long as it is stretched. Therefore we have $\ell_i(u) = \ell'(u)$. Since levels never decrease in our algorithm we have $\ell_i(v) \geq \ell'(v)$. By the definition of persevering paths we have $w_i(u, v) \geq w'(u, v)$. We therefore get

$$\ell_i(u) = \ell'(u) \leq \ell'(v) + w'(u, v) \leq \ell_i(v) + w_i(u, v)$$ □

Using the property above, we would ideally like to do the following: We would like to split a shortest path from x to the root r into subpaths of length $\leq \tau$ and replace each subpath by a persevering path such that the length of each subpath and the persevering path by which it is replaced are approximately the same. Repeated applications of the inequality of Lemma 3.14 would then allow us to bound the level of x. However, this approach alone does not work because the definition of a locally persevering emulator does not always guarantee the existence of a persevering path. Instead of a persevering path, the locally persevering emulator might also provide us with a shortest path of G_i that is contained in the current emulator H_i. In principle this is a nice property

because a shortest path is even better than an approximate shortest path. But the problem now is that nodes on this path could be stretched and only for non-stretched nodes the difference in levels of two nodes can be bounded by the weight of the edge between them. We can resolve this issue by induction, which allows us to use the contained path only partially.

Lemma 3.15 (Correctness)**.** *For every node x and every $i \leq k$, $d_{G_i}(x,r) \leq \ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x,r) + \beta$.*

Proof. We start with a proof of the first inequality $d_{G_i}(x,r) \leq \ell_i(x)$. Consider the (weighted) path P from x to the root r in the monotone ES-tree. 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 3.12. By repeated applications of this inequality for every edge on P we get $\ell_i(x) \geq w_i(P) + \ell_i(r) = w_i(P)$ (since the level of the root r is always 0). Since P is a path in H_i we have $w_i(P) \geq d_{G_i}(x,r)$ by the definition of a locally persevering emulator.

We now prove the second inequality $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x,r) + \beta$. Let α, β, and τ be the parameters of the locally persevering emulator $\mathcal{H}$. Consider a shortest path $P = \langle x_l, x_{l-1}, \ldots, x_0 \rangle$ from $x = x_l$ to $r = x_0$ in G_i. We prove the following claim by induction on j: $\ell(x_j) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x_j,r) + \beta$. Clearly, this claim implies the inequality we have to show. The claim is clearly true if x_j is the root node itself and by Lemma 3.13 it is also true if x_j is stretched. From now on we assume that $x_j \neq r$ and that x_j is not stretched.

Consider first the case that $d_{G_i}(x,r) < \tau$. If there is a shortest path P from x to r in G_i that is also contained in (H_i, w_i), we do the following. Consider the first edge (x,y) on P. Note that $d_{G_i}(y,r) < d_{G_i}(x,r)$. Therefore we may apply the induction hypothesis and get $\ell_i(y) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(y,r) + \beta$. Since the edge (x,y) is contained in (H_i, w_i) with weight 1 and since x is not stretched we have $\ell_i(x) \leq \ell_i(y) + 1 = \ell_i(y) + d_{G_i}(x,y)$. By combining both inequalities we get

$$\begin{aligned}
\ell_i(x) &\leq (\alpha + \beta/\tau) \cdot d_{G_i}(y,r) + \beta + d_{G_i}(x,y) \\
&\leq (\alpha + \beta/\tau) \cdot (d_{G_i}(x,y) + d_{G_i}(y,r)) + \beta \\
&= (\alpha + \beta/\tau) \cdot (d_{G_i}(x,r)) + \beta
\end{aligned}$$

where the last equation follows from the fact that y lies on a shortest path from x to r.

If there is no shortest path from x to r in G_i that is also contained in (H_i, w_i), then, since $\mathcal{H}$ is an (α, β, τ)-locally persevering emulator, we know that there is a path P from x to r in (H_i, w_i) that is persevering up to time i such that $w_i(P) \leq \alpha d_{G_i}(x,r) + \beta$, which clearly also satisfies $w_i(P) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x,r) + \beta$. Since P is persevering up to time i we may apply Lemma 3.14 and get $\ell_i(x) \leq \ell_i(r) + w_i(P) = 0 + w_i(P)$. By combining both inequalities we get $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x,r) + \beta$ as desired.

Now consider the case that $d_{G_i}(x,r) \geq \tau$. If x is not connected to r in G_i there is nothing to prove. Otherwise, there exists a node z (on the shortest path from x to r in G_i) such that $d_{G_i}(x,z) = \tau$ and $d_{G_i}(x,r) = d_{G_i}(x,z) + d_{G_i}(z,r)$. If there is a shortest path P from x to z in G_i that is also contained in (H_i, w_i), we proceed similar as before. We consider the first edge (x,y) on P and apply the induction hypothesis on y to obtain $\ell_i(x) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(x,r) + \beta$. (Note that the node z is not relevant for this argument.)

If there is no shortest path from x to z in G_i that is also contained in (H_i, w_i), then, since $\mathcal{H}$ is an (α, β, τ)-locally persevering emulator, we know that there is a path P from x to z in (H_i, w_i) that is persevering up to time i such that $w_i(P) \leq \alpha d_{G_i}(x,z) + \beta$. Since P is persevering up to time i we may apply Lemma 3.14 and get $\ell_i(x) \leq \ell_i(z) + w_i(P)$. Since $d_{G_i}(z,r) < d_{G_i}(x,r)$, we may apply the induction hypothesis on z and get that $\ell_i(z) \leq (\alpha + \beta/\tau) \cdot d_{G_i}(z,r) + \beta$. We now

combine these three inequalities and, together with $d_{G_i}(x,z) = \tau$, obtain the desired approximation as follows.

$$\begin{aligned}
\ell_i(x) \leq \ell_i(z) + w_i(P) &\leq (\alpha + \beta/\tau) \cdot d_{G_i}(z,r) + \beta + \alpha d_{G_i}(x,z) + \beta \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(z,r) + \beta + \alpha d_{G_i}(x,z) + \beta \cdot 1 \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(z,r) + \beta + \alpha d_{G_i}(x,z) + \beta \cdot d_{G_i}(x,z)/\tau \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(z,r) + \beta + (\alpha + \beta/\tau) \cdot d_{G_i}(x,z) \\
&= (\alpha + \beta/\tau) \cdot (d_{G_i}(x,z) + d_{G_i}(z,r)) \\
&= (\alpha + \beta/\tau) \cdot d_{G_i}(x,r)\,.
\end{aligned}$$

The last equation follows from the definition of z. □

Finally, we provide the running time analysis. Here we have to consider the following technical detail. It is unavoidable that the total update time of a dynamic algorithm that allows edge weight increases depends on the number of edge weight increases (see [11] for a discussion of this issue). Now remember that the monotone ES-tree algorithm runs on the emulator $\mathcal{H}$ and every edge deletion in $\mathcal{G}$ could result in *several* edge weight increases in the emulator $\mathcal{H}$. This is the reason why the term $\phi(\mathcal{H})$, which counts the number of weight changes in the dynamic emulator $\mathcal{H}$, has to appear in the running time.

Lemma 3.16 (Running time)**.** *The monotone ES-tree has a total update time of* $O(\phi(\mathcal{H}) + \sum_{x \in V} \deg_{\mathcal{H}}(x) R^d \log n)$ *in addition to the time needed for maintaining the* $(1+\epsilon, 2, 2/\epsilon)$*-locally persevering emulator* $\mathcal{H}$*.*

Proof. Remember that the monotone ES-tree algorithm runs on the emulator $\mathcal{H}$. An edge deletion in $\mathcal{G}$ could result in several changes in the emulator $\mathcal{H}$. All of these changes have to be processed by the algorithm with constant time per change plus the time needed for running the procedure Reconnect. Therefore the total update time is equal to the number of changes $\phi(\mathcal{H})$ in $\mathcal{H}$ plus the cumulated time for running the reconnection procedure.

We now bound the running time of the procedure Reconnect. Here, the well-known level-increase argument works. Consider a node x that is processed in the while loop of the reconnection-procedure after some operation in the emulator. Now the following holds: If the level of a node increases, the algorithm has to spend time $O(\deg_{\mathcal{H}}(x) \log(x))$ where $\deg_{\mathcal{H}}(x)$ is the maximum degree of x in $\mathcal{H}$ over time and bounds the degree of x in the current emulator. If the level of x does not increase, the algorithm has to spend time $O(1)$.

We now argue that in the second case we do not have to count the running time separately. Consider the situation when the level of a node y increases and its child x in the tree 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 $O(1)$ to the time $O(\deg_{\mathcal{H}}(x) \log n)$ that we already charge to y. The only other situation for a node being processed without increasing its level is when x is processed directly after the edge weight increase (or deletion) of an edge (x,y). In that case we charge the running time of $O(1)$ to the weight increase (or delete) operation. By the term $O(\phi(\mathcal{H}))$ we already charge constant time to each such operation anyway.

Since the monotone ES-tree is only maintained up to depth R^{d}, at most R^{d} level increases are possible for every node. Therefore the total time needed for every node x in the procedure Reconnect is $R^{\mathrm{d}} \deg_{\mathcal{H}}(x) \log n$. Thus, the total update time of the monotone ES-tree is $O(\phi(\mathcal{H}) + \sum_{x \in V} \deg_{\mathcal{H}}(x) R^{\mathrm{d}} \log n)$. □

3.3 Putting Everything Together: $\tilde{O}(n^{5/2})$-total-time Algorithm for $(1+\epsilon, 2)$- and $(2, 0)$-approximate APSP

In the following we show how the monotone ES-tree of Lemma 3.11 together with the locally persevering emulator of Lemma 3.4 can be used to obtain $(1+\epsilon, 2)$- and $(2+\epsilon, 0)$-approximate decremental APSP algorithms with $\tilde{O}(n^{5/2}/\epsilon^2)$ total update time.

We will proceed as follows: we first show, as a simple consequence of the previous two parts of this sections, how to maintain a $(1+\epsilon, 2)$-approximate monotone ES-tree (Corollary 3.17). This monotone ES-tree provides us with $(1+\epsilon, 2)$-approximate *single-source* shortest paths. The main work now is to extend this to an algorithm for $(1+\epsilon, 2)$-approximate *all-pairs* shortest paths. We define an *approximate* version of the center cover data structure (Definition 3.19) and show how such a data structure can be turned into an approximate decremental APSP algorithm (Lemma 3.20). Afterwards we show that a $(1+\epsilon, 2)$-approximate center cover data structures exists (Lemma 3.21) by using Corollary 3.17. We can use this data structure to obtain a $(1+\epsilon, 2)$-approximate APSP algorithm (due to Lemma 3.20). The $(2+\epsilon, 0)$-approximate algorithm then follows as a corollary from the $(1+\epsilon, 2)$-approximate algorithm. Our arguments mainly follow Roditty and Zwick [33] with necessary adaptions for *approximate* instead of exact center cover data structures. The most interesting technical details are the following:

- 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.
- The $(1+\epsilon, 2)$-approximate center cover data structure does *not* directly imply a $(1+\epsilon, 2)$-approximate decremental APSP algorithm, but gives a slightly worse approximation. This only causes a problem for relatively small distances. Therefore we can resolve this issue by maintaining a monotone ES-tree of small depth for every node.
- The $(2+\epsilon, 0)$-approximation follows from the $(1+\epsilon, 2)$-approximation by the following simple observation: we can easily find out whether the distance of two nodes is 1 by checking whether there is an edge between the two nodes in the current graph.

First, we show a simple consequence of the main results of the previous two parts in this section (Lemma 3.4 and Lemma 3.11). The monotone ES-tree gives a $(1+\epsilon, 2)$-approximation of the true distance if we run it on the emulator we described in Section 3.1. This is a key tool for the following results.

Corollary 3.17 ($(1+\epsilon, 2)$-approximate monotone ES-tree)**.** *Given a dynamic graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$, an integer R^d, and a distinguished node r, we can maintain a $(1+\epsilon, 2)$-approximate monotone ES-tree rooted at r up to depth R^d under k edge deletions. After each deletion i, the data structure can provide an estimate $\delta(x, r)$ of the distance $d_{G_i}(x, r)$ in constant time such that $d_{G_i}(x, r) \leq \delta(x, r)$. Furthermore, if $d_{G_i}(x, r) \leq R^d$, then $\delta(x, r) \leq (1+\epsilon) d_{G_i}(x, r) + 2$ with high probability. The total update time of this data structure is $O(n^{3/2} \log n/\epsilon + n^{3/2} R^d \log^2 n + mn^{1/2} \log n/\epsilon)$.*

The term $mn^{1/2} \log n/\epsilon$ in the running time stated above comes from the time needed for maintaining an emulator. If we want to maintain several $(1+\epsilon, 2)$-approximate monotone ES-trees with different root nodes, then the total update time for maintaining this emulator only has to be counted once. We will consider this fact when we apply this result later on.

Proof of Corollary 3.17. Let $\mathcal{H}$ denote the $(1+\epsilon, 2, 2/\epsilon)$-locally persevering emulator of Lemma 3.4. The total update time for maintaining $\mathcal{H}$ is $O(mn^{1/2}\log n/\epsilon)$. We maintain a monotone ES-tree rooted at r with range parameter R^{d} on $\mathcal{H}$, as given by Lemma 3.11. Let $\delta(x,r)$ denote the level of the node x in the monotone ES-tree of r after deletion i. By Lemma 3.11 we have $d_{G_i}(x,c) \leq \delta(x,c)$, and furthermore, if $d_{G_i}(x,c) \leq R^{\mathrm{d}}$, then

$$\delta_{G_i}(x,c) \leq (1+\epsilon+2/(2/\epsilon))d_{G_i}(x,c)+2 \leq (1+2\epsilon)d_{G_i}(x,c)+2$$

which can be reduced to $\delta(x,c) \leq (1+\epsilon)d_{G_i}(x,c)+2$ as desired by letting the algorithm run with $\epsilon' = \epsilon/2$. By Lemma 3.4, the sum of the maximum degrees of nodes in the emulator is $\sum_{x\in V}\deg_{\mathcal{H}}(x) = O(n^{3/2}\log n)$ and the total number of changes in $\mathcal{H}$ is $\phi(\mathcal{H}) = O(n^{3/2}\log n/\epsilon)$. Therefore, by Lemma 3.11, the total update time of the monotone ES-tree (without the time for maintaining the emulator $\mathcal{H}$) is

$$O\left(\phi(\mathcal{H}) + \sum_{x\in V}\deg_{\mathcal{H}}(x)R^{\mathrm{d}}\log n\right) = O(n^{3/2}\log n/\epsilon + n^{3/2}R^{\mathrm{d}}\log^2 n)\,.$$

Together with the time needed for maintaining $\mathcal{H}$, this gives a total update time of

$$O(n^{3/2}\log n/\epsilon + n^{3/2}R^{\mathrm{d}}\log^2 n + mn^{1/2}\log n/\epsilon)\,. \qquad \square$$

We now define an approximate version of the center cover data structure and show how such a data structure can be turned into an approximate decremental APSP algorithm. We 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.

Definition 3.18 (Approximate center cover)**.** *Let U be a set of nodes of 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 $c \in U$ 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 $c \in U$.*

Definition 3.19. *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 a set of* centers *$C = \{1,2,\ldots,l\}$ and a set of nodes $U = \{c^1, c^2, \ldots, c^l\}$ such that, for every $0 \leq i \leq k$, U is an (α,β)-approximate center cover of G_i with parameter R^c. For every center $j \in C$ we call c^j the* location *of center j and for every node x we say that x is (α,β)-covered by j if x is (α,β)-covered by c^j. The data structure provides the following update operations:*

- Initialize(G_0, R^c, R^d)*: Initialize the data structure*
- Delete(u,v)*: Delete the edge (u,v) from the current graph*

The data structure provides the following query operations, applied on the current graph G_i after the i-th edge deletion:

- Distance(j,x)*: This procedure returns a distance estimate $\delta(c^j,x)$ of the distance between the current location c^j of center j and the node x such that $\delta(c^j,x) \leq \alpha d_{G_i}(c^j,x)+\beta$, provided that $d_{G_i}(c^j,x) \leq R^d$. If $d_{G_i}(c^j,x) > R^d$, then this procedure either returns $\delta(c^j,x) = \infty$ or it returns $\delta(c^j,x) \leq \alpha d_{G_i}(c^j,x)+\beta$.*

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

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 algorithm with slightly worse approximation guarantee. The proof of this result follows [33].

Lemma 3.20 (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 $T(n, m, R^c, R^d)$. Then, for every $\epsilon \leq 1$, there is an $(\alpha + 2\alpha^2\epsilon, 2\beta + 2\alpha\beta)$-approximate decremental APSP algorithm with $O(\log \log n)$ query time and a total update time of $\sum_p T(n, m, R^c_p, R^d_p)$ where $R^c_p = \epsilon 2^p$ and $R^d_p = \alpha\epsilon 2^p + \beta + 2^{p+1}$ (for $0 \leq p \leq \log n$).*

The query time can be reduced to $O(1)$ if there is an (α', β')-approximate decremental APSP algorithm for some constants α' and β' with constant query time (at the cost of additional update time for this algorithm).

Proof. The algorithm considers $\log n$ layers where the p-th layer is responsible for the distance range from 2^p to 2^{p+1}. For each layer the algorithm maintains a center cover data structure using the parameters $R^{\mathrm{c}}_p = \epsilon 2^p$ and $R^{\mathrm{d}}_p = 2^{p+1} + \alpha\epsilon 2^p + \beta$. The distance estimate of the center cover data structure for two nodes x and y is denoted by $\delta(x, y)$. Let i be the index of the last deletion.

For every layer p, we can compute a distance estimate $\delta'_p(x, y)$ for two 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., $d_{G_i}(x, c^j) \leq \alpha R^{\mathrm{c}}_p + \beta$. If x is not (α, β)-covered by any center we set $\delta'_p(x, y) = \infty$. Otherwise we query the center cover data structure to get estimates $\delta_p(c^j, x)$ and $\delta_p(c^j, y)$ of the distances from c^j to x and y, respectively. (Remember that these distance estimates might be ∞.) We now set $\delta'_p(x, y) = \delta_p(c^j, x) + \delta_p(c^j, y)$. Note that, given p, we can compute $\delta'_p(x, y)$ in constant time. The query procedure will rely on three properties of the distance estimate $\delta'_p(x, y)$.

1. The distance estimate never under-estimates the real distance, i.e., $\delta'_p(x, y) \geq d_{G_i}(x, y)$.
2. If $d_{G_i}(x, y) \geq 2^p$ and $\delta'_p(x, y) \neq \infty$, then $\delta'_p(x, y) \leq (\alpha + 2\alpha^2) d_{G_i}(x, y) + 2\beta + 2\alpha\beta$.
3. If x is in a connected component of size at least R^{c}_p and $d_{G_i}(x, y) \leq 2^{p+1}$, then $\delta'_p(x, y) \neq \infty$.

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

$$d_{G_i}(x, y) \leq d_{G_i}(c^j, x) + d_{G_i}(c^j, y) \leq \delta_p(c^j, x) + \delta_p(c^j, y) = \delta'_p(x, y)\,.$$

which means that $\delta'_p(x, y)$ never under-estimates the real distance. For the second property we remark that if $\delta'_p(x, y) \neq \infty$, it must be the case that we have found a center j with location c^j that (α, β)-covers x. Therefore $d_{G_i}(x, c^j) \leq \alpha R^{\mathrm{c}} + \beta$. Furthermore it must be the case that $\delta_p(x, c^j) \neq \infty$ and $\delta_p(c^j, y) \neq \infty$ and therefore $\delta_p(x, c^j) \leq \alpha d_{G_i}(x, c^j) + \beta$ and $\delta_p(c^j, y) \leq \alpha d_{G_i}(c^j, y) + \beta$. Now

simply consider the following chain of inequalities:

$$\begin{aligned}\delta'_p(x,y) = \delta_p(x,c^j) + \delta_p(c^j,y) &\leq \alpha(d_{G_i}(c^j,x) + d_{G_i}(c^j,y)) + 2\beta \\ &\leq \alpha(d_{G_i}(c^j,x) + d_{G_i}(c^j,x) + d_{G_i}(x,y)) + 2\beta \\ &= \alpha(2d_{G_i}(c^j,x) + d_{G_i}(x,y)) + 2\beta \\ &\leq \alpha(2\alpha\epsilon 2^p + 2\beta + d_{G_i}(x,y)) + 2\beta \\ &\leq \alpha(2\alpha\epsilon d_{G_i}(x,y) + 2\beta + d_{G_i}(x,y)) + 2\beta \\ &= (\alpha + 2\alpha^2\epsilon)d_{G_i}(x,y) + 2\beta + 2\alpha\beta\end{aligned}$$

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

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

as well as

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

Therefore we get $\delta_p(c^j,x) \leq \alpha d_{G_i}(c^j,x)+\beta < \infty$ as well as $\delta_p(c^j,y) < \infty$ which implies $\delta'_p(x,y) \neq \infty$ as desired.

A query time of $O(\log n)$ is immediate as we can simply return the minimum of all distance estimates $\delta'_p(x,y)$. 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 $\delta'_p(x,y) \neq \infty$. 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 algorithm with query time $O(1)$ for some constants α' and β', We first compute the distance estimate $\delta''(x,y)$ of the second algorithm for which we know that $d_{G_i}(x,y) \in [\delta''(x,y)/\alpha' - \beta', \delta''(x,y)]$. Now there is only a constant number of indices p such that $\{2^p,\ldots,2^{p+1}\} \cap [\delta''(x,y)/\alpha' - \beta', \delta''(x,y)] \neq \emptyset$. For every such index we compute $\delta'_p(x,y)$ and return the minimum distance estimate obtained by this process. □

Our next goal is to obtain a useful approximate center cover data structure. We show the existence of $(1+\epsilon,2)$-approximate center cover data structures with favorable running time. Here we use the $(1+\epsilon,2)$-approximate monotone ES-tree of Corollary 3.17 together with a well-known idea of finding a center cover by random sampling of nodes. Again, the proof is rather straightforward.

Lemma 3.21 (Existence of $(1+\epsilon,2)$-approximate center cover data structure)**.** *For all parameters R^c and R^d such that $R^c \leq R^d$, there is a $(1+\epsilon,2)$-approximate center cover data structure that maintains a $(1+\epsilon,2)$-approximate center cover with high probability. It has constant query time and a total update time of*

$$O(n^{5/2}\log^2 n/(\epsilon R^c) + n^{5/2}\log^3 n R^d/R^c + mn^{1/2}\log n/\epsilon)\,.$$

The term $mn^{1/2}\log n/\epsilon$ in the running time stated above comes from the time needed for maintaining an emulator. If we want to maintain more than one $(1+\epsilon,2)$-approximate center cover data structure, then the total update time for maintaining the emulator only has to be counted once. We will consider this fact when we apply this result later on.

Proof of Lemma 3.21. It is well-known (see for example [37] and [33]) that, by random sampling, we can obtain a set $U = \{c^1,c^2,\ldots,c^l\}$ of size $O(n\log n/R^{\mathrm{c}})$ that is a center cover of G_i for every

$i \leq k$. 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 $i \leq k$. 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.

For every center j, we maintain the $(1+\epsilon, 2)$-approximate monotone ES-tree of Corollary 3.17 rooted at c^j up to depth R^{d}. As there are $O(n \log n / R^{\mathrm{c}})$ centers, the total update time for maintaining all the monotone ES-trees is

$$O((n^{3/2} \log n/\epsilon + n^{3/2} \log^2 n R^{\mathrm{d}}) n \log n / R^{\mathrm{c}} + mn^{1/2} \log n/\epsilon)$$

which is equal to

$$O(n^{5/2} \log^2 n/(\epsilon R^{\mathrm{c}}) + n^{5/2} \log^3 n R^{\mathrm{d}}/R^{\mathrm{c}} + mn^{1/2} \log n/\epsilon)\,.$$

Note that the second term in the running time of Corollary 3.17 does *not* have to be multiplied by the number of centers because it comes from maintaining an emulator that all monotone ES-trees can share. Furthermore, for every node x we maintain a list of centers that cover x, i.e., a list containing every $j \in C$ such that the level of x in the monotone ES-tree of c^j is at most $(1+\epsilon)R^{\mathrm{c}} + 2$. To maintain this list we just have to observe the level increases in each monotone ES-tree. If for some center j and some node x the level of x in the monotone ES-tree rooted at c^j exceeds the threshold $(1+\epsilon)R^{\mathrm{c}} + 2$ we remove j from the list of x. Therefore we can charge the running time for maintaining this list to the level-increases in the monotone ES-tree, which means that the running time for maintaining the list is already included in the total update time stated above.

We now show how to answer queries of a center cover data structure, as specified in Definition 3.19, in constant time. Let i be the index of the last deletion. For every node x and every center j, let $\delta(x, c^j)$ denote the estimate of the distance of x to the location of center j as returned by the monotone ES-tree rooted at c^j. Given a center j and a node x we answer a query for the distance of x to c^j by returning $\delta(c^j, y)$ which gives a $(1+\epsilon, 2)$-approximation of the true distance by Corollary 3.17. Given a node x, we answer a query for finding a nearby center by returning any center j in the list of centers of x. If this list is empty we return $\perp$. Note that for every center j in the list we know that $d_{G_i}(x, c^j) \leq (1+\epsilon)R^{\mathrm{c}} + 2$ as required because $d_{G_i}(x, c^j) \leq \delta(x, c^j)$ by Corollary 3.17. If x is in a connected component of size at most R^{c} we can ensure that we really find a center j in that list because, by our random choice of centers, we have $d_{G_i}(x, c^j) \leq R^{\mathrm{c}}$ for some center j with high probability in that case. If $d_{G_i}(x, c^j) \leq R^{\mathrm{c}}$, then also $d_{G_i}(x, c^j) \leq R^{\mathrm{d}}$ and by Corollary 3.17 we get $\delta(x, c^j) \leq (1+\epsilon) d_{G_i}(x, c^j) + 2 \leq (1+\epsilon)R^{\mathrm{c}} + 2$. Therefore the list of centers of x contains j. □

Finally we show how to obtain the $(1+\epsilon, 2)$-approximate decremental APSP algorithm. We combine the $(1+\epsilon, 2)$-approximate center cover data structure of Lemma 3.21 with the approximate APSP algorithm of Lemma 3.20 that uses this data structure. However, the approximation guarantee we get from this algorithm is not as good as desired because the additive errors accumulate. Therefore we need a special treatment of relatively short distances. It is a standard technique to maintain small-depth shortest paths trees to handle these cases. We can use our monotone ES-tree for this purpose.

Theorem 3.22. *Given $\epsilon \leq 1$ and a decremental graph $\mathcal{G} = (G_i)_{0 \leq i \leq k}$, there is a $(1+\epsilon, 2)$-approximate decremental APSP algorithm with constant query time and a total update time of $O(n^{5/2} \log^2 n/\epsilon^2 + n^{5/2} \log^4 n/\epsilon)$. It is correct with high probability.*

Proof. By Lemma 3.21 there is a $(1+\epsilon, 2)$-approximate center cover data structure that has constant query time and a total update time of

$$O(n^{5/2}\log^2 n/(\epsilon R^{\mathrm{c}}) + n^{5/2}\log^3 n R^{\mathrm{d}}/R^{\mathrm{c}} + mn^{1/2}\log n/\epsilon)\,.$$

By applying Lemma 3.20, this data structure gives us an an approximate decremental APSP algorithm. The parameters used by the algorithm are $R^{\mathrm{c}}_p = \epsilon 2^p$ and $R^{\mathrm{d}}_p = (1+\epsilon)\epsilon 2^p + 2 + 2^{p+1} \leq 2^{p+3}$ (for $0 \leq p \leq \log n$) and therefore its total running time is[14]

$$O\left(\sum_{p=0}^{\log n}\left(n^{5/2}\log^2 n/(\epsilon R^{\mathrm{c}}_p) + n^{5/2}\log^3 n R^{\mathrm{d}}_p/R^{\mathrm{c}}_p\right) + mn^{1/2}\log n/\epsilon\right)\,.$$

Now consider the following arithmetic simplifications:

$$\begin{aligned}
O\left(\sum_{p=0}^{\log n} n^{5/2}\log^2 n/(\epsilon R^{\mathrm{c}}_p)\right) &= O\left(\sum_{p=0}^{\log n} n^{5/2}\log^2 n/(\epsilon^2 2^p)\right)\\
&= O\left(\frac{n^{5/2}\log^2 n}{\epsilon^2}\sum_{p=0}^{\log n}\frac{1}{2^p}\right) = O\left(n^{5/2}\log^2 n/\epsilon^2\right)\\
O\left(\sum_{p=0}^{\log n} n^{5/2}\log^3 n R^{\mathrm{d}}_p/R^{\mathrm{c}}_p\right) &= O\left(\sum_{p=0}^{\log n} n^{5/2}\log^3 n 2^{p+3}/(\epsilon 2^p)\right)\\
&= O\left(\sum_{p=0}^{\log n} 8n^{5/2}\log^3 n/\epsilon\right) = O\left(n^{5/2}\log^4 n/\epsilon\right)
\end{aligned}$$

Therefore the total update time of the algorithm provided by Lemma 3.20 can also be expressed as

$$O\left(n^{5/2}\log^2 n/\epsilon^2 + n^{5/2}\log^4 n/\epsilon + mn^{1/2}\log n/\epsilon\right)\,. \tag{1}$$

The query time of the algorithm provided by Lemma 3.20 can be reduced to $O(1)$. The reason is that Bernstein and Roditty [12] provide, for example, a decremental $(5+\epsilon', 0)$-approximate APSP algorithm for some constant ϵ'. The total update time of the Bernstein-Roditty algorithm is

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

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

Let i be the index of the last edge deletion. The approximation guarantee of the algorithm provided by Lemma 3.20 with $\alpha = 1+\epsilon$ and $\beta = 2$ is

$$d_{G_i}(x,y) \leq \delta(x,y) \leq (1+\epsilon+2(1+\epsilon)^2\epsilon)\cdot d_{G_i}(x,y) + 4 + 4(1+\epsilon)\,.$$

Using $\epsilon \leq 1$ we get

$$\delta(x,y) \leq (1+9\epsilon)d_{G_i}(x,y) + 12\,.$$

[14]The term $mn^{1/2}\log n$ shows up in the running time only once because it is the time for maintaining an emulator that can be shared by all $\log n$ layers.

By running the algorithm with $\epsilon' = \epsilon/18$ we get

$$\delta(x,y) \leq (1+\epsilon/2)d_{G_i}(x,y) + 12\,.$$

It is left to show how to get the approximation down to $(1+\epsilon, 2)$. Here we need a special treatment of relatively small distances.

Note that if $d_{G_i}(x,y) \geq 24/\epsilon$ we get

$$\delta(x,y) \leq (1+\epsilon/2)d_{G_i}(x,y) + 12 \leq (1+\epsilon/2)d_{G_i}(x,y) + \epsilon d_{G_i}(x,y)/2 = (1+\epsilon)d_{G_i}(x,y)\,.$$

We now show how to deal with pairs of nodes whose distance is relatively small. In addition to the algorithm above, we maintain the $(1+\epsilon, 2)$-approximate monotone ES-tree of Corollary 3.17 with parameter $R^{\mathrm{d}} = 24/\epsilon$ rooted at every node. The total update time for these additional monotone ES-trees is[15]

$$O((n^{3/2}\log n/\epsilon + n^{3/2}R^{\mathrm{d}}\log^2 n)n + mn^{1/2}\log n/\epsilon)$$

which is equal to

$$O(n^{5/2}\log n/\epsilon + n^{5/2}\log^2 n/\epsilon + mn^{1/2}\log n/\epsilon)\,. \tag{3}$$

For every pair of nodes x and y, let $\delta'(x,y)$ denote the distance estimate obtained by querying the distance to y in the monotone ES-tree rooted at x of depth $R^{\mathrm{d}} = 24/\epsilon$ (this query takes constant time). If $d_{G_i}(x,y) \leq 24/\epsilon$, then by Lemma 3.11 we can be sure that y is contained in the monotone ES-tree root at x and we get

$$d_{G_i}(x,y) \leq \delta'(x,y) \leq (1+\epsilon)d_{G_i}(x,y) + 2\,.$$

Thus, the minimum between $\delta(x,y)$ and $\delta'(x,y)$, the distance estimates of the APSP algorithm of Lemma 3.20 and the additional ES-trees, respectively, provides a $(1+\epsilon, 2)$-approximation of the distance $d_{G_i}(x,y)$.

The query time of the overall algorithm is obviously constant because we just have to return the minimum of two data structures with constant query time. Now for calculating the overall running time, observe that $m \leq n^2$ and thus $O(mn^{1/2}\log n/\epsilon) = O(n^{5/2}\log n/\epsilon)$. The dominating terms of (1), (2), and (3) are $n^{5/2}\log^2 n/\epsilon^2$ and $n^{5/2}\log^4 n/\epsilon$ and therefore the total update time is

$$O\left(n^{5/2}\log^2 n/\epsilon^2 + n^{5/2}\log^4 n/\epsilon\right)\,. \qquad \square$$

The $(2+\epsilon, 0)$-approximate decremental APSP algorithm 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. *For every $\epsilon \leq 1$ there is a $(2+\epsilon, 0)$-approximate decremental APSP algorithm with constant query time and a total update time of $O(n^{5/2}\log^2 n/\epsilon^2 + n^{5/2}\log^4 n/\epsilon)$. It is correct with high probability.*

Proof. By running the algorithm of Theorem 3.22 we can, after each edge deletion i, for all nodes x and y, query for a distance estimate $\delta(x,y)$ in constant time that satisfies:

$$d_{G_i}(x,y) \leq \delta(x,y) \leq (1+\epsilon)d_{G_i}(x,y) + 2$$

[15] The term $mn^{1/2}\log n$ shows up in the running time only once because it is the time for maintaining an emulator that can be shared by all monotone ES-trees.

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

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

If $d_{G_i}(x,y) < 2$, then we actually have $d_{G_i}(x,y) \leq 1$ 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,

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

we get $d_{G_i}(x,y) \leq \delta'(x,y) \leq (2+\epsilon)d_{G_i}(x,y)$. Clearly, this algorithm can answer queries in constant time by returning the distance estimate $\delta'(x,y)$ and has the same total update time as the $(1+\epsilon,2)$-approximate algorithm, namely $O(n^{5/2}\log^2 n/\epsilon^2 + n^{5/2}\log^4 n/\epsilon)$. □

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

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

Theorem 4.1 (Main result of Section 4: Deterministic $O((mn\log n)/\epsilon)$ total update time)**.** *Given a decremental graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$ and $0 < \epsilon \leq 1$, there is a deterministic decremental algorithm for maintaining $(1+\epsilon)$-approximate shortest paths with a total update time of $O((mn\log n)/\epsilon)$ and a query time of $O(\log\log n)$.*

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 2.11. Recall that R^{c} and R^{d} are the *coverage range* and *distance maintenance range* parameters where we want (a) every node (in a connected component of size at least R^{c}) to be within a distance of R^{c} from some center, and we want (b) to maintain a distance from each center to every node within a distance of R^{d}.

Roditty and Zwick [33], following an argument of Ullman and Yannakakis [37], observed that making each vertex a center independently at random with probability $(c\ln n)/R^{\mathrm{c}}$, where c is a constant and $1 \leq R^{\mathrm{c}} \leq n$, gives a set C of centers such that with probability at least $1 - n^{-(c-1)}$ 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 $O((cn\ln n)/R^{\mathrm{c}})$. The randomized decremental APSP algorithm of [33] simply chooses a large enough value of c so that with high probability C fulfills the center cover property with parameter R^{c} not only in the initial graph but continues to fulfill them in all the $O(n^2)$ graphs generated during $O(n^2)$ edge deletions. This is only possible because it is assumed that the "adversary" that generates the deletions is oblivious, i.e., he 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 $O(nm\log n)$. Once we have such a data structure, we can use the multilayer approach of Roditty and Zwick, as discussed in Section 2.3, to obtain an algorithm for maintaining decremental approximate shortest paths by using the parameters $R^{\mathrm{c}}_p = \epsilon 2^p$ and $R^{\mathrm{d}}_p = 2^{p+2}$ in every layer p.

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 describe 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 Even-Shiloach tree is actually a concept rather than a new implementation: we implement it in a straightforward by building a new Even Shiloach tree every time we have to move it. However, analyzing the total update time needs new insights and a careful charging argument. To separate the analysis of the moving Even-Shiloach tree from the charging argument, we describe the data structure in two pieces:

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

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

Combining these two pieces gives the center cover data structure.

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 ($\mathsf{n}_{\mathsf{open}}$) and the *moving distance* ($\mathsf{d}_{\mathsf{move}}$). 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 dynamic graph $\mathcal{G} = (G_i)_{0 \le i \le k}$ maintains a set of* centers $C = \{1, 2, \ldots, l\}$ *and a set of nodes $U = \{c^1, c^2, \ldots, c^l\}$. For every center $j \in C$ we call c^j the* location *of center j. The data structure provides the following update operations.*

- DELETE(u, v)*: Delete edge (u, v) from the graph*
- 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^j to node x. It is required that there exists a path in the current graph from c^j to x.*

The data structure provides the following query operations, applied on the current graph G_i after the i-th edge deletion:

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

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

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

In the rest of Section 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-the 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 ($\mathsf{d}_{\mathsf{move}}$))**.** *The* total moving distance*, denoted by* $\mathsf{d}_{\mathsf{move}}$*, is defined as* $\mathsf{d}_{\mathsf{move}} = \sum_{0 \leq i < k} \sum_{j \in C_i} d_{G_i}(c_i^j, c_{i+1}^j)$.

The main result of this section is that we can maintain a moving centers data structure in $O(m(\mathsf{n}_{\mathsf{open}} R^{\mathrm{d}} + \mathsf{d}_{\mathsf{move}}))$ time, as in Proposition 4.4 below. The data structure is actually very simple: we maintain an Even-Shiloach tree of depth at most R^{d} at every node for which we open a center and for every node to which we move a center. Note that our algorithm treats an Even-Shiloach tree at each such node as a new tree, regardless of whether we open a center or move a center there. While the algorithm can naively treat each Even-Shiloach tree as a new one, the analysis cannot: if we do so, we will get a total update time of $O((\mathsf{n}_{\mathsf{open}} + \mathsf{n}_{\mathsf{move}}) m R^{\mathrm{d}})$, where $\mathsf{n}_{\mathsf{move}}$ is the total number of move-operations (since maintaining each Even-Shiloach tree takes $O(mR^{\mathrm{d}})$ total update time). Instead, we bound the cost incurred by the move-operation based on how far a center is moved, i.e., the moving distance ($\mathsf{d}_{\mathsf{move}}$). This argument allows us to replace the unfavorable term $\mathsf{n}_{\mathsf{move}} m R^{\mathrm{d}}$ by $\mathsf{d}_{\mathsf{move}} m$. The deterministic center cover data structure of Section 4.2 will generate a sequence of center open and move requests so that $\mathsf{d}_{\mathsf{move}} = O(n)$. For simplifying the analysis, we state the following result under a technical assumption, that will always be fulfilled by the intended use of the moving centers data structure in Section 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* $O((\mathsf{n}_{\mathsf{open}} R^d + \mathsf{d}_{\mathsf{move}}) m)$*, where* $\mathsf{n}_{\mathsf{open}}$ *is the number of open operations and* $\mathsf{d}_{\mathsf{move}}$ *is the total moving distance. 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 is currently a center; and (2) for every node a doubly-linked *center list* of centers by which it is covered. Recall that a node is covered by a center iff the node is contained in the ES-tree of depth R^{d} 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

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

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 depth 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 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 $O(\mathsf{n}_{\mathsf{open}}m+\mathsf{n}_{\mathsf{move}}m)$ for all initializations where $\mathsf{n}_{\mathsf{move}}$ 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 $O(\mathsf{n}_{\mathsf{open}}m+\mathsf{n}_{\mathsf{move}}m)$ will be absorbed by $O((\mathsf{n}_{\mathsf{open}}R^{\mathrm{d}}+\mathsf{d}_{\mathsf{move}})m)$, 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 decreases, i.e., $C_i \subseteq C_k$ for every $0 \leq i \leq k$. We will show that $\sum_{0<i\leq k}\sum_{j\in C_i} T(i,j) = O((\mathsf{n}_{\mathsf{open}}R^{\mathrm{d}} + \mathsf{d}_{\mathsf{move}})m)$.

Consider the $i+1$-th edge deletion and let $j \in C_{i+1}$ be a center. By Corollary 2.8, the total time for processing this deletion in the ES-tree of center j is

$$T(i+1,j) = \sum_{x\in V} \deg(x) \cdot \left(\min\left(d_{G_{i+1}}(x,c^j_{i+1}), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x,c^j_{i+1}), R^{\mathrm{d}}\right)\right) \tag{4}$$

If j has already been opened before the i-th edge deletion, then, by the triangle inequality, we get

$$d_{G_i}(x,c^j_i) \leq d_{G_i}(x,c^j_{i+1}) + d_{G_i}(c^j_i,c^j_{i+1})$$

which is equivalent to

$$d_{G_i}(x,c^j_{i+1}) \geq d_{G_i}(x,c^j_i) - d_{G_i}(c^j_i,c^j_{i+1})\,.$$

It follows that

$$\min\left(d_{G_i}(x,c^j_{i+1}), R^{\mathrm{d}}\right) \geq \min\left(d_{G_i}(x,c^j_i) - d_{G_i}(c^j_{i+1},c^j_i), R^{\mathrm{d}}\right) \geq \min\left(d_{G_i}(x,c^j_i), R^{\mathrm{d}}\right) - d_{G_i}(c^j_i,c^j_{i+1})\,.$$

Therefore we get

$$\begin{aligned}
T(i+1,j) &\leq \sum_{x\in V} \deg(x) \cdot \left(\min\left(d_{G_{i+1}}(x,c^j_{i+1}), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x,c^j_i), R^{\mathrm{d}}\right) + d_{G_i}(c^j_i,c^j_{i+1})\right)\\
&= \sum_{x\in V} \deg(x) \cdot \left(\min\left(d_{G_{i+1}}(x,c^j_{i+1}), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x,c^j_i), R^{\mathrm{d}}\right)\right) + \sum_{x\in V} \deg(x) \cdot d_{G_i}(c^j_i,c^j_{i+1})\\
&\leq \sum_{x\in V} \deg(x) \cdot \left(\min\left(d_{G_{i+1}}(x,c^j_{i+1}), R^{\mathrm{d}}\right) - \min\left(d_{G_i}(x,c^j_i), R^{\mathrm{d}}\right)\right) + 2m \cdot d_{G_i}(c^j_i,c^j_{i+1})\,.
\end{aligned}$$

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

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

Consider now a center j and the o_j-th edge deletion. By (4) we can bound the running time $T(o_j, j)$ as follows:

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

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

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

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

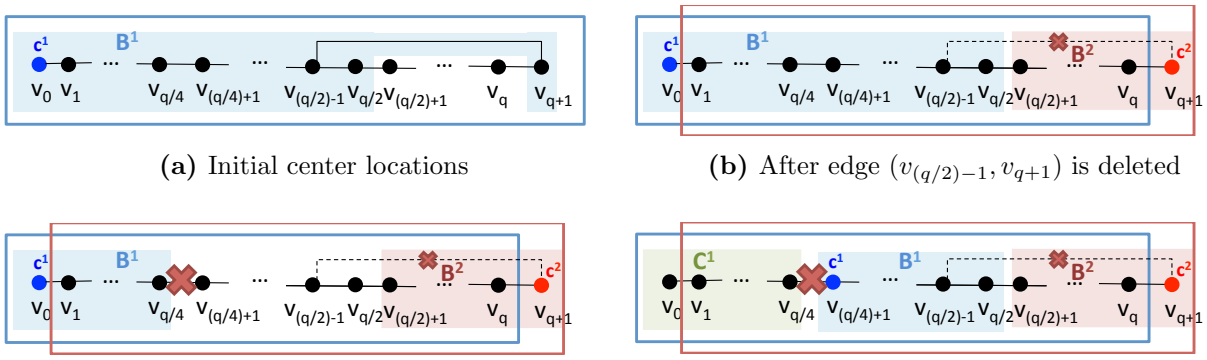
$$\begin{aligned}
\sum_{0<i\leq k} \sum_{j\in C_i} T(i,j) &= \sum_{j\in C_k} \sum_{o_j\leq i\leq k} T(i,j) \\
&\leq \sum_{j\in C_k} mR^{\mathrm{d}} + \sum_{j\in C_k} \sum_{o_j\leq i<k} 2m \cdot d_{G_i}(c_i^j, c_{i+1}^j) \\
&= \sum_{j\in C_k} mR^{\mathrm{d}} + 2m \cdot \sum_{0\leq i<k} \sum_{j\in C_i}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

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

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 2.10. 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 a distance of 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)**.** *Given a decremental graph $\mathcal{G} = (G_i)_{0\leq i\leq k}$ and parameters R^c and R^d such that $R^c \leq R^d$, there is a center cover data structure with a total* deterministic *update time of $O(mnR^d/R^c)$ and constant query time.*



(a) Initial center locations

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

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

(d) After moving c^1

Figure 1: Example of our algorithm for maintaining the center cover data structure using the moving centers data structure, as in Proposition 4.5. We use $q = R^c$ 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 $v_{(q/4)+1}$.

4.2.1 High-Level Ideas

Our algorithm will internally use the moving centers data structure (MovingCenter) from Section 4.1. 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 1 for an example; note that $q = R^c$): For each center j, it maintains two sets B^j and C^j, where B^j is always defined to be the set of nodes whose distance to center j is at most $R^c - |C^j|$. Initially, the algorithm sets $C^j = \emptyset$ and chooses a set of centers such that all sets B^j are disjoint (see Figure 1a). The sets C^j will never decrease during the algorithm. After an edge deletion, there might be a node in a large connected component that is not covered by any center anymore (e.g., v_{q+1} in Figure 1b). In this case, the algorithm simply opens a new center at that node. However, before doing so it has to check whether $|B^j \cup C^j| < R^c/2$ for some center j. (For example, after edge $(v_{q/4}, v_{(q/4)+1})$ is deleted as in Figure 1c, $|B^1 \cup C^1| = (q/4 + 1) < q/2 = R^c/2$.) 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 1d, 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 details. Recall from Proposition 4.4 that opening and maintaining a center together cost $O(mR^c)$ time in total and a move-operation incurs a total time of $O(m)$ per one unit moving distance. So, to get the desired $O(mn \log n)$ 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

$$\mathsf{n}_{\mathsf{open}} = O(n/R^c) \quad \text{and} \quad \mathsf{d}_{\mathsf{move}} = O(n).$$

To guarantee that we open $\mathsf{n}_{\mathsf{open}} = O(n/R^{\mathrm{c}})$ centers, we imagine that each node holds a coin in 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 in the beginning. Instead, our algorithm will maintain two sets for each center j: the set B^j of *borrowed* nodes from which center j has borrowed coins but might have to return the coins back, and the set C^j of *collected* nodes from which center j has collected coins that it will never return. After all edge deletions, j will hold the coins of all nodes in $B^j \cup C^j$. Our algorithm will maintain $B^j \cup C^j$ with two properties:

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

These two properties easily imply that every center will get at least $R^{\mathrm{c}}/2$ coins in the end – center j simply collects coins from those in $B^j \cup C^j$; consequently, they guarantee that $\mathsf{n}_{\mathsf{open}} = O(n/R^{\mathrm{c}})$ 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 is *moved through* u. To guarantee that the total moving distance is $O(n)$, we need one more property:

3. (Confinement) If the location of center j is moved *only through nodes in* C^j, and it can be moved through each node at most once.

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 $\mathsf{d}_{\mathsf{move}} = O(n)$ as desired.

It is left to check that the algorithm we have sketched earlier satisfies all three properties above. The largeness property can be guaranteed using the fact that after every edge deletion, the algorithm will move every center j such that $|B^j \cup C^j| < R^{\mathrm{c}}/2$ to a new node; the only non-obvious thing 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 1a is a subset of $B^1 \cup C^1$ in Figure 1d.)

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 in 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 detail). To show the initial-disjointness property (Property (P1)), we use the fact that j is of distance at least R^{c} from other centers when j is opened which implies that $B^j \cap B^{j'} = \emptyset$. Additionally, we will prove that $C^{j'}$ contains only nodes in connected components of size less than R^{c} whereas the 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 j to C^j and moves j to a node outside of this connected component. For example, in Figure 1d 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. Besides, 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 invoke the same queries (i.e., MovingCenter.FindCenter and MovingCenter.Distance) from the moving centers data structure (see Procedures CenterCover.FindCenter and CenterCover.Distance in Algorithm 3).

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 is described in Procedure CenterCover.GreedyOpen in Algorithm 3. 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.FindCover(x) returns $\perp$ and the size of the connected component containing x is at least R^{c} (we refer to Remark 4.22 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. 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 14 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. (It can be easily shown 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 16). 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. Afterwards, we report the deletion of the edge (u, v) to the moving centers data structure so that it updates the distances between centers and nodes to the new distances in G_{i+1}. Finally, we execute the CenterCover.GreedyOpen procedure to make sure that every node remains covered: if there is a node x that is not covered, we open a center at x. This completes the deletion operation.

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 2.10.

procedure CenterCover.GreedyOpen( )
  // Greedily open new centers at nodes that are not covered
  Let G_i denote the current graph
  for every 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 Remark 4.22 for the implementation detail.*
    if (CenterCover.FindCenter(x) $= \perp$) **and** ($|\operatorname{Comp}_{G_i}(x)| \ge R^{\mathrm{c}}$) **then**
      $j \leftarrow$ MovingCenter.Open(x)
      // Tell the moving centers data structure to open a new center at x. Let j be the index of this center.
      Set $C^j \leftarrow \emptyset$, $r^j \leftarrow R^{\mathrm{c}}/2$, and $c^j \leftarrow x$.

procedure CenterCover.Initialize(G_0, R^{c}, R^{d})
  // Parameters: The initial graph G_0 of the dynamic graph $\mathcal{G}$ and integers R^{c} and R^{d}.
  // Initialize the moving centers data structure MovingCenter *(see Definition 4.2)*
  MovingCenter.Initialize(G_0, R^{c}, R^{d})
  CenterCover.GreedyOpen( )

procedure CenterCover.FindCenter(v)
  // Parameter: Node v.
  return MovingCenter.FindCenter(j, v)

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

procedure CenterCover.Delete(u, v)
  // Parameter: $(i+1)$-th deleted edge (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 such that $|\operatorname{Comp}_{G_i}(c^j)| \ge r^j$ and $|\operatorname{Comp}_{G_{i+1}}(c^j)| < r^j$.
    // Find a center j for which the connected component containing it becomes smaller than r^j. See Remark 4.23 for how to find such j. (Actually, there will be at most one such j (see Lemma 4.15).)
  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. See Remark 4.21 for how to check the if statement.
    if u and c^j are *not* connected in G_{i+1} **then** $y \leftarrow u$; **else** $y \leftarrow v$.
    Set $C^j \leftarrow C^j \cup \operatorname{Comp}_{G_{i+1}}(c^j)$, $r^j \leftarrow r^j - |\operatorname{Comp}_{G_{i+1}}(c^j)|$, and $c^j \leftarrow y$.
    MovingCenter.Move(j, y)
  MovingCenter.Delete(u, v)
    // Report edge deletion to moving centers data structure MovingCenter *(Definition 4.2).*
  CenterCover.GreedyOpen( )

4.2.3 Analysis

The correctness of Algorithm 3 is immediate. As the procedure GreedyOpen is called after every edge deletion, no node in a connected component of size at least R^c will ever be uncovered. 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 to bound the number $\mathsf{n}_{\mathsf{open}}$ of open-operations performed by the algorithm and the total moving distance $\mathsf{d}_{\mathsf{move}}$. 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 $\mathsf{n}_{\mathsf{open}} = O(n/R^c)$ open-operations and that the total moving distance is $\mathsf{d}_{\mathsf{move}} = O(n)$. 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. Afterwards we will show that all other operations of the algorithm can also be carried out within this total update time. To make our arguments precise we will use the following notation.

Definition 4.6. *Remember that $\mathcal{G} = (G_i)_{0 \leq i \leq k}$ is the 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. In the rest of this section we use the following notation:*

- *For all nodes x and y we denote by $d_i(x,y) = d_{G_i}(x,y)$ the distance between x and y in the graph G_i.*
- *For every node x, we denote by $\mathrm{Comp}_i(x) = \mathrm{Comp}_{G_i}(x)$ the nodes in the connected component of x in the graph G_i.*
- *For every center j, we denote by c_i^j, r_i^j, and C_i^j the values of c^j, r^j, B^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. Before we go into the details of the running time analysis we 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 $|\mathrm{Comp}_i(x)| < r$, then $\mathrm{Comp}_i(x) = B'$. Furthermore, $|\mathrm{Comp}_i(x)| < r$ if and only if $|B'| < r$.*

Proof. Clearly $B' \subseteq \mathrm{Comp}_i(x)$ and thus $|B'| \leq |\mathrm{Comp}_i(x)|$. Therefore, if $|\mathrm{Comp}_i(x)| < r$ also $|B'| < r$. Now assume that $|B'| < r$. We first show that $\mathrm{Comp}_i(x) \subseteq B'$. Let y be a node in $\mathrm{Comp}_i(x)$ and assume by contradiction that $d_i(x,y) > r$. Since $y \in \mathrm{Comp}_i(x)$, x and y are connected and therefore the shortest path from x to y has to go over some node z such that $d_i(x,z) = r$. The shortest path P from x to z contains $d_i(x,z) = r$ edges and $r+1$ nodes. For every node z' on P we have $d_i(x,z') \leq r$ and thus $P \subseteq B'$. Since $|P| = r+1$ we get $|B'| \geq r+1$ which contradicts our assumption. Therefore $d_i(x,y) \leq r$ which means that $\mathrm{Comp}_i(x) \subseteq B'$. Now $|\mathrm{Comp}_i(x)| \leq |B'| < r$ 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. Afterwards the algorithm only modifies r^j and C^j when a center is moved. Since r^j is increased by exactly the amount by which $|C^j|$ is decreased, the equation still holds. □

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 the decremental setting 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. No center will ever be moved back into such a connected component. Since $B_i^j \subseteq \operatorname{Comp}_i(c_i^j)$, we conclude that B_i^j and C_i^j are disjoint. □

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

Lemma 4.11 (Initial disjointness)**.** *When the algorithm opens a center j after the i-th edge deletion, then $B_i^j \cup C_i^j$ is disjoint from $B_i^{j'} \cup C_i^{j'}$ 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. Consider the $(i+1)$-th deletion of an edge (u, v). We only have to argue that the claim holds for centers that the algorithm has already opened before. 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 the decremental setting we also have $B_{i+1}^j \subseteq B_i^j$ and the claim follows.

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

The observation needed for proving the shrinking property is $B_{i+1}^j \cup A \subseteq B_i^j$. From this observation we get $B_{i+1}^j \cup C_{i+1}^j = B_{i+1}^j \cup C_i^j \cup A \subseteq B_i^j \cup C_i^j$ as desired. We first prove $A \subseteq B_i^j$ and then we prove $B_{i+1}^j \subseteq B_i^j$. Let B' be the set $B' = \{z \in V \mid d_{i+1}(x,z) \leq r_i^j\}$. Since $|A| < r_i^j$ we get $A \subseteq B'$ by Observation 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 P from x to v in G_i consisting of $d_i(x,v)$ many edges. Every edge on P except for the last one (which is (u,v)) is also contained in G_{i+1} and therefore all nodes on P except for v are contained in A. Therefore we get $|A| \geq |P \setminus \{v\}| = d_i(x,v)$. To prove that $z \in B_i^j$ we show that $d_i(x,z) \leq r_i^j$, which can be seen from the following chain of inequalities:

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

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 $B_{i+1}^j \cup C_{i+1}^j$ and $B_{i+1}^{j'} \cup C_{i+1}^{j'}$ are disjoint. This shows that, for *all* centers j and j' such that $j \neq j'$, $B_{i+1}^j \cup C_{i+1}^j$ and $B_{i+1}^{j'} \cup C_{i+1}^{j'}$ are disjoint. $\square$

Largeness property. We now want to prove the largeness property which states that for every center j the size of the set $B^j \cup C^j$ is always at least $R^c/2$. The largeness property will follow from the invariant $|B^j| \geq r^j$. Before we can prove this invariant we have to argue that our algorithm really moves every center j that fulfills the "moving condition" $|\mathrm{Comp}_i(c^j)| \geq r^j$ and $|\mathrm{Comp}_i(c^j)| < r^j$. 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. *If, after the $(i+1)$-th deletion of an edge (u,v), $|\mathrm{Comp}_i(c_i^j)| \geq r_i^j$ and $|\mathrm{Comp}_{i+1}(c_i^j)| < r_i^j$, then $u \in B_i^j$ and $v \in B_i^j$.*

Proof. Suppose that $d_i(u, c_i^j) > r_i^j$. Let P a shortest path from c_i^j to u in G_i consisting of $d_i(u, c_i^j) > r_i^j$ many edges. By our assumption this path contains at least $r_i^j + 1$ nodes. The edge (u,v) can only appear as the last edge on the shortest path P. Therefore, after deleting it, there are still r_i^j nodes connected to c_i^j which contradicts the assumption that $\mathrm{Comp}_{i+1}(c_i^j) < r_i^j$. Thus, $d_i(u, c_i^j) \leq r_i^j$ which means that $u \in B_i^j$. Since the edge (u,v) is undirected the same argument works for v. $\square$

Lemma 4.15 (Uniqueness of center to move)**.** *After the $(i+1)$-th deletion of an edge (u, v), there can be at most one center j such that $|\operatorname{Comp}_{i+1}(c_i^j)| \geq r_i^j$ and $|\operatorname{Comp}_i(c_i^j)| < r_i^j$ and either u is connected to c_i^j (and v is disconnected from c_i^j) or v is connected to c_i^j (and u is disconnected from c_i^j).*

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

Now suppose that there are two centers $j \neq j'$ such that $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$ and $|\operatorname{Comp}_i(c_i^j)| < r_i^j$, and $|\operatorname{Comp}_i(c_i^{j'})| \geq r_i^{j'}$ and $|\operatorname{Comp}_i(c_i^{j'})| < r_i^{j'}$. By Observation 4.14, we get $u \in B_i^j$ and $v \in B_i^{j'}$ 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 $|B_i^j| \geq r_i^j$.*

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

We now show that the invariant is maintained for all centers that have already been opened when 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 applying Observation 4.7.

Now consider the case that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$. Since the invariant holds for i we have $|B_i^j| \geq r_i^j$. By Observation 4.7 this implies that $|\operatorname{Comp}_i(x)| \geq r_i^j$. Due to Lemma 4.15 we can be sure that the algorithm will move center j from node x to node y (where either $y = u$ or $y = v$). Remember that we have $c_i^j = x$, $c_{i+1}^j = y$, and $r_{i+1}^j = r_i^j - |\operatorname{Comp}_{i+1}(x)|$ in that case. Since x and y were connected in G_i but are not connected anymore in G_{i+1} we get $\operatorname{Comp}_{i+1}(y) = \operatorname{Comp}_i(x) \setminus \operatorname{Comp}_{i+1}(x)$. Due to $\operatorname{Comp}_{i+1}(x) \subseteq \operatorname{Comp}_i(x)$ it follows that

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

By Observation 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 $r_i^j = R^{\mathrm{c}}/2 - |C_i^j|$. Therefore we get the desired bound as follows:

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

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 to $\mathsf{n}_{\mathsf{open}} = O(n/R^{\mathrm{c}})$. This will be useful for our goal of limiting the total update time of the moving centers data structure to $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$.

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

Proof. Let C denote the set of centers. Note that moving a center does not change the size of C. Therefore, the size of C after all deletions 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

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

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 in C^j, and it can be moved through each node at most once.

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 P_i^j be the set of nodes of a shortest path from c_i^j to c_{i+1}^j in G_i. Then the following two properties hold for every center j and every $i \leq k$:*

- *$P_i^j \setminus \{c_{i+1}^j\} \subseteq C_k^j$ where the C_k^j is the value of C^j after all edge deletions*
- *For every $i' \leq k$ such that $i' \neq i$, $P_i^j \setminus \{c_{i+1}^j\}$ and $P_{i'}^j \setminus \{c_{i'+1}^j\}$ are disjoint.*

Proof. We will first prove the following claim for every center j and every $i \leq k$: $P_i^j \subseteq C_{i+1}^j \setminus C_i^j$. Consider the situation that the algorithm moves a center j from c_i^j to c_{i+1}^j after the $(i+1)$-th deletion of an edge (u, v). By the rules of the algorithm for moving centers we either have $c_{i+1}^j = u$ or $c_{i+1}^j = v$. Due to Observation 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 P_i^j be a shortest path from c_i^j to c_{i+1}^j in G_i. All nodes in P_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 $P_i^j \setminus \{c_{i+1}^j\} \subseteq \text{Comp}_{i+1}(c_i^j)$. Since $C_{i+1}^j = C_i^j \cup \text{Comp}_{i+1}(c_i^j)$, we get $P_i^j \subseteq C_{i+1}^j$. It is also clear that $P_i^j \setminus \{c_{i+1}^j\} \subseteq B_i^j$. Since B_i^j and C_i^j are disjoint (Observation 4.10), also P_i^j and C_i^j are disjoint. It therefore follows that $P_i^j \subseteq C_{i+1}^j \setminus C_i^j$.

Using this claim, the two properties demanded in the lemma now easily follow. Since no node is ever removed from C^j we get $P_i^j \subseteq C_{i+1}^j \setminus C_i^j \subseteq C_{i+1}^j \subseteq C_k^j$. For the second property we need the following observation: for every $i \leq k$ and every $i' < i$, P_i^j and $P_{i'}^j \setminus \{c_{i'+1}^j\}$ are disjoint. This is true for the following reason. Since $P_i^j \subseteq C_{i+1}^j \setminus C_i^j$, we get that P_i^j and C_i^j are disjoint. Since $P_{i'}^j \setminus \{c_{i'+1}^j\} \subseteq C_i^j$, also P_i^j and $P_{i'}^j \setminus \{c_{i'+1}^j\}$ are disjoint. □

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

Proof. We let C 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 P_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 P_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 $d_i(c_i^j, c_{i+1}^j) = |P_i^j \setminus \{c_{i+1}^j\}|$.

By Lemma 4.19 the sets $P_i^j \setminus \{c_{i'+1}^j\}$ and $P_{i'}^j \setminus \{c_{i'+1}^j\}$ are disjoint for $i \neq i'$ and therefore $\sum_{i \leq k} |P_i^j \setminus \{c_{i+1}^j\}| = |\bigcup_{i \leq k} P_i^j \setminus \{c_{i+1}^j\}|$. The value of the set C^j after all edge deletions is given by C_k^j for every center j. By Lemma 4.19 we have $P_i^j \subseteq C_k^j$ for every center j and every $i \leq k$. This implies that $\bigcup_{i \leq k} P_i^j \subseteq C_k^j$ 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 $\mathsf{d}_{\mathsf{move}} \leq n$ as follows.

$$\begin{aligned}\mathsf{d}_{\mathsf{move}} = \sum_{j \in C} \sum_{o_j \leq i < k} d_i(c_i^j, c_{i+1}^j) &= \sum_{j \in C} \sum_{o_j \leq i < k} |P_i^j \setminus \{c_{i+1}^j\}| \\ &= \sum_{j \in C} \left|\bigcup_{i<k} P_i^j \setminus \{c_{i+1}^j\}\right| \leq \sum_{j \in C} |C_k^j| = \left|\bigcup_{j \in C} C_k^j\right| \leq n\,. \qquad \square\end{aligned}$$

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

Remark 4.21 (Detail of Line 16 in Algorithm 3: Checking if two nodes are in the same connected component). We use the dynamic connectivity data structure of [25] to efficiently check whether two nodes are connected. This deterministic connectivity data structure has constant query time and a total update time of $O(mn^{1/2})$. Note that we update the connected components data structure directly after every edge deletion whereas we update the moving centers data structure just before the greedy open procedure.

Remark 4.22 (Detail of Line 4 in Algorithm 3). Performing the check in the if-condition of Line 4 takes time $O(nm)$ over all deletions. Given a node x, we first check whether x is covered by any center by querying the moving centers data structure (if x is covered the procedure returns a center covering x, otherwise it returns $\perp$.) This check takes constant time. If a node x is not covered we additionally have to check whether $|\operatorname{Comp}_i(x)| < R^{\mathrm{c}}$. Note that if $|\operatorname{Comp}_i(x)| < R^{\mathrm{c}}$ for some node x we do not have to consider this node anymore after future deletions because connected components never increase their size in the decremental setting. Therefore we may spend time $O(m)$ for every node x to perform a BFS tree computation for figuring out $|\operatorname{Comp}_i(x)|$. If $|\operatorname{Comp}_i(x)| < R^{\mathrm{c}}$, then we charge the work to the node and will never process the node again in the future and if $|\operatorname{Comp}_i(x)| \geq R^{\mathrm{c}}$, then we charge the work to the open-operation. Therefore the total running time over all deletions for performing this check in the if-condition is $O((n + \mathsf{n}_{\mathsf{open}})m)$ where $\mathsf{n}_{\mathsf{open}}$ is the total number of open-operations. Since $\mathsf{n}_{\mathsf{open}} = O(n)$ by Lemma 4.18 the running time is $O(nm)$.

Remark 4.23 (Detail of Line 14 of Algorithm 3). We now explain how to find a center j such that $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$ and $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$ after the $(i+1)$-th deletion of an edge (u, v). We

note that we could use the dynamic connectivity data structure of [27] to query the size of the connected component containing each c_i^j in $O(\log n)$ time. This would lead to a total update time of $O(mn \log n)$. There is also a simple way to improve the running time to $O(mn)$. For this we will need two observations.

First, we observe that it is sufficient to find j such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$ (without checking whether $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$). This is because, by Lemma 4.16 and Observation 4.7, it is anyway the case that $|\operatorname{Comp}_i(c_i^j)| \geq r_i^j$ for every center j. Second, we observe that, for every center j such that $|\operatorname{Comp}_{i+1}(c_i^j)| < r_i^j$, 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 connectivity data structure [25] discussed in Remark 4.21. 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 find out 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 iterate over every node and check whether it is connected to c_i^j in G_{i+1} in constant time using the dynamic connectivity data structure. By this process of counting the number of connected nodes we can determine the size of $\operatorname{Comp}_{i+1}(c_i^j)$ in time $O(n)$. The total running time for this algorithm is $O(n)$ per deletion and thus $O(mn)$ in total.

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. All other operations of the algorithm that are not part of the moving centers data structure can, as we have shown above, also be performed within this running time.

Theorem 4.24. *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 $O(\mathsf{n}_{\mathsf{open}} m R^{\mathrm{d}} + \mathsf{d}_{\mathsf{move}} m)$ where $\mathsf{n}_{\mathsf{open}}$ is the total number of open operations and $\mathsf{d}_{\mathsf{move}}$ 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 $O(n/R^{\mathrm{c}})$ and by Lemma 4.20 the total moving distance is $O(n)$. Therefore the total update time of the moving centers data structure is

$$O(\mathsf{n}_{\mathsf{open}} m R^{\mathrm{d}} + \mathsf{d}_{\mathsf{move}} m) = O(mnR^{\mathrm{d}}/R^{\mathrm{c}} + mn) = O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$$

because $R^{\mathrm{c}} \leq R^{\mathrm{d}}$. As argued in Remark 4.21, Remark 4.22, and Remark 4.23, all other operations of the algorithm can be implemented within a total update time of $O(mnR^{\mathrm{d}}/R^{\mathrm{c}})$. Therefore the claimed running time follows. □

5 Conclusion

We obtained two new algorithms for solving the decremental approximate APSP algorithm in unweighted undirected graphs. The first algorithm provides a $(1+\epsilon, 2)$-approximation and has a total update time of $\tilde{O}(n^{5/2}/\epsilon^2)$ and constant query time. The main idea behind this algorithm is to run an algorithm of Roditty and Zwick [33] on a sparse dynamic emulator. In particular, we

modify the central shortest paths tree data structure of Even and Shiloach [23, 28] to deal with edge insertions in a monotone manner. Our approach is conceptually different from the approach of Bernstein and Roditty [12] who also maintain an ES-tree in a sparse dynamic emulator. The second algorithm provides a $(1+\epsilon, 0)$-approximation and has a *deterministic* total update time of $\tilde{O}(mn \log n/\epsilon)$ and constant query time. We obtain it by derandomizing the algorithm of [33] using a new amortization argument based on the idea of relocating Even-Shiloach trees.

Our results directly motivate the following directions for further research. It would be interesting to extend our derandomization technique to other randomized algorithms. In particular, we ask whether it is possible to derandomize the *exact* decremental APSP algorithm of Baswana, Hariharan and Sen [7] (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, also 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?

The sparsification techniques used here and at other places only work for undirected graphs. Can we also get subcubic algorithms for *directed* graphs (without relying on these sparsification techniques)? In fact, not even decremental single-source reachability in directed graphs can currently be done faster than with a total time of $O(mn)$ for a sequence of $\Omega(m)$ deletions.

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 [17] 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

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

[2] G. Ausiello, P. G. Franciosa, and G. F. Italiano. Small Stretch Spanners on Dynamic Graphs. *Journal of Graph Algorithms and Applications*, 10(2):365–385, 2006. Announced at ESA, 2005.

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

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

[5] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM J. Comput.*, 28(1):263–277, 1998. Announced at FOCS, 1993.

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

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

[8] S. Baswana, S. Khurana, and S. Sarkar. Fully Dynamic Randomized Algorithms for Graph Spanners. *ACM Transactions on Algorithms*, 8(4):35:1–35:51, 2012. Announced at ESA, 2004, and SODA, 2008.

[9] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

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

[11] A. Bernstein. Maintaining Shortest Paths Under Deletions in Weighted Directed Graphs. In *STOC*, page to appear, 2013.

[12] A. Bernstein and L. Roditty. Improved Dynamic Algorithms for Maintaining Approximate Shortest Paths Under Deletions. In *SODA*, pages 1355–1365, 2011.

[13] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[14] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM J. Comput.*, 28(1):210–236, 1998. Announced at FOCS, 1993.

[15] E. Cohen and U. Zwick. All-pairs small-stretch paths. *J. Algorithms*, 38(2):335–353, 2001. Announced at SODA, 1997.

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

[17] C. Demetrescu and G. F. Italiano. A New Approach to Dynamic All Pairs Shortest Paths. *Journal of the ACM*, 51(6):968–992, 2004. Announced at STOC, 2003.

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

[19] D. Dor, S. Halperin, and U. Zwick. All-Pairs Almost Shortest Paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000. Announced at FOCS, 1996.

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

[21] M. Elkin. Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners. *ACM Transactions on Algorithms*, 7(2):20:1–20:17, 2011. Announced at ICALP, 2007.

[22] M. Elkin and D. Peleg. $(1 + \epsilon, \beta)$-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631, 2004. Announced at STOC, 2001.

[23] S. Even and Y. Shiloach. An On-Line Edge-Deletion Problem. *Journal of the ACM*, 28(1):1–4, 1981.

[24] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006. Announced at FOCS, 2001.

[25] M. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM Journal on Computing*, 31(2):364374, 2001. Announced at ICALP, 1997.

[26] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997. Announced at STOC, 1994.

[27] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. Announced at STOC, 1998.

[28] V. King. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *STOC*, pages 81–91, 1999.

[29] P. Loubal and B. A. T. S. Commission. *A Network Evaluation Procedure*. Bay Area Transportation Study Commission, 1967.

[30] J. D. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical Report LBS-TNT-26, London Business School, Transport Network Theory Unit, 1967.

[31] L. Roditty and R. Tov. Approximating the girth. *ACM Trans. Algorithms*, 9(2):15:1–15:13, Mar. 2013. Announced at SODA, 2011.

[32] L. Roditty and U. Zwick. On Dynamic Shortest Paths Problems. *Algorithmica*, 61(2):389–401, 2011. Announced at ESA, 2004.

[33] L. Roditty and U. Zwick. Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs. *SIAM Journal on Computing*, 41(3):670–683, 2012. Announced at FOCS, 2004.

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

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

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

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

[38] V. Vassilevska, R. Williams, and R. Yuster. All pairs bottleneck paths and max-min matrix products in truly subcubic time. *Theory of Computing*, 5(1):173–189, 2009. Announced at STOC, 2007.

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

[40] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.

Appendix

A Proof of Fact 1.1

Due to a reduction by Dor, Halperin, and Zwick [19], a combinatorial[18] algorithm for APSP, even a $(2-\epsilon, 0)$-approximation or $(1+\epsilon, 1)$-approximation one[19], 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 [39, Theorem 1.3], the $O(n^{3-\delta})$-time combinatorial algorithm will imply breakthrough results for a few other problems. Since dynamic algorithms are combinatorial in nature and can be used to solve static APSP, the same argument applies. In particular, two additive error of two in our $(1+\epsilon, 2)$-approximation algorithm is unavoidable if we wish to get a $O(n^{1-\delta})$ running time (a so-called *truly subcubic* time) and keep a small multiplicative error of $1+\epsilon$. For the same reason, a multiplicative error of two in our $(2+\epsilon, 0)$-approximation algorithm is also unavoidable. Similarly, the running time of our deterministic algorithm cannot be improved further unless we allow larger additive or multiplicative errors.

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

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