2018 IEEE 59th Annual Symposium on Foundations of Computer Science

Near-Optimal Approximate Decremental All Pairs Shortest Paths

Shiri Chechik
Tel Aviv University, Israel.
shiri.chechik@gmail.com

***Abstract*—In this paper we consider the *decremental* approximate all-pairs shortest paths (APSP) problem, where given a graph G the goal is to maintain approximate shortest paths between all pairs of nodes in G under a sequence of online adversarial edge deletions.**

We present a decremental APSP algorithm for undirected weighted graphs with $(2+\epsilon)k-1$ stretch, $O(mn^{1/k+o(1)}\log(nW))$ total update time and $O(\log\log(nW))$ query time for a fixed constant ϵ, where W is the maximum edge weight (assuming the minimum edge weight is 1) and k is any integer parameter. This is an exponential improvement both in the stretch and in the query time over previous works.

***Keywords*-dynamic algorithms; shortest paths; emulator;**

I. Introduction

Dynamic algorithms are data structures that are designed to handle an online sequence of update operations while maintaining some key functionality on the graph. A notable and very well studied example is dynamic shortest paths. An update usually involves adding or removing a node or an edge from the graph. The algorithm is said to be *decremental* if it handles only deletions, *incremental* if it handles only insertions and *fully dynamic* if it handles both deletions and insertions.

In this paper we consider the problem of decremental (approximate) all pairs shortest paths (APSP) in weighted undirected graphs. Our algorithm supports the following two operations: 1. Delete(e) – delete the given edge e from the graph. 2. Distance(u,v) – return an estimate on the distance between u and v in the current graph G.

We say that the algorithm has *stretch* k if the returned estimated distance is always at least the real distance and at most k times the real distance. More formally, let $\widehat{\mathbf{dist}}(u,v)$ denote the estimated distance returned by the algorithm on a query (u,v). An algorithm has stretch k, if at any given time for any query (u,v) we have $\mathbf{dist}(u,v) \leq \widehat{\mathbf{dist}}(u,v) \leq k \times \mathbf{dist}(u,v)$, where $\mathbf{dist}(u,v)$ is the actual distance between u and v in the current version of the graph (the graph after all deletions occurred so far). A key concern in designing dynamic shortest paths algorithms is to minimize the update time, namely, the time it takes the algorithm to adapt to an update. Another important goal is to minimize the query time, that is, the time it takes the algorithm to answer a query. Typically, the query time is expected to be small (polylog or even better constant) while minimizing the update time. Finally, since we deal with approximate shortest paths algorithms, we would like to minimize the stretch as much as possible. In the decremental setting, we usually measure the aggregate sum of the update times over all deletions, which is referred to as the *total update time*.

Related work: The first non-trivial bound maintaining decremental single source shortest paths (SSSP) is due to Even and Shiloach [14]. More precisely, they presented a decremental SSSP for undirected, unweighted graphs with $O(mn)$ total update time[1]. A similar scheme was independently found by Dinitz [13]. Henzinger and King [15] later observed that the algorithm of Even and Shiloach can be easily adapted to directed graphs. King [22] later generalized this algorithm to directed graphs with small edge weights. Additional optimizations for the memory usage and conditional lower bounds can be found in [22], [23], [24], [21].

In the *randomized* approximate decremental undirected SSSP there was a lot of progress in recent years [10], [19], [17] culminating in the breakthrough result of Henzinger, Krinninger and Nanongkai [17] who presented a $(1+\epsilon)$ approximate decremental SSSP algorithm with near linear total update time of $O(m^{1+o(1)}\log(nW))$, where W is the maximum edge weight (assuming the minimum edge weight is 1). The first deterministic $(1+\epsilon)$ SSSP algorithm that goes beyond the $O(mn)$ total update time was presented in [9] with $\tilde{O}(n^2)$ total update time [2]. This result was later improved for sparse graph [9] and was also generalized to weighted graphs [8]. For further recent progress on the approximate directed SSSP see [18], [20].

We now turn our attention to the dynamic APSP problem, which has been extensively studied in the last three decades. There has been a large number of papers considering the dynamic APSP problem, of which we describe just the most up to date. For the fully dynamic case, there was a long chain of papers culminating in the breakthrough result of Demetrescu and Italiano [12], who devised a fully dynamic exact APSP for directed general graphs with non negative edge weights, with amortized cost of $\tilde{O}(n^2)$. This was later extended by Thorup [27] to handle negative edge weights.

[1] As usual, n (respectively, m) is the number of nodes (resp., edges) in the graph.

[2] Here and throughout, the $\tilde{O}(\cdot)$ notation hides both logarithmic factors in n and small polynomials in $1/\epsilon$ factors.

2575-8454/18/$31.00 ©2018 IEEE
DOI 10.1109/FOCS.2018.00025

Thorup [28] obtained later a fully dynamic exact APSP with worst case update time of $\tilde{O}(n^{2.75})$. Bernstein [6] presented fully dynamic APSP with $2+\epsilon$ stretch, $O(\log\log\log n)$ query time, and $\hat{O}(m)$ amortized update time, where $\hat{O}(f(n)) = f(n)n^{O(1/\sqrt{\log n})}$. Baswana, Khurana, and Sarkar [4] presented a fully dynamic APSP algorithm for undirected unweighted graphs such that for an integer parameter k, the construction of [4] has stretch $4k$, $\hat{O}(n^{1+1/k})$ amortized update cost, and $O(\log\log\log n)$ query time. Sankowski [26] used matrix multiplication in unweighted graphs and achieved subquadratic worst-case update time of $O(n^{1.932})$ at the cost of a superlinear query time of $O(n^{1.288})$.

Much of the work on dynamic APSP considers partially dynamic algorithms, namely, incremental or decremental solutions. Roditty and Zwick [25] presented very efficient randomized dynamic $(1+\epsilon)$ APSP for the only incremental and for the only decremental cases for undirected unweighted graphs with total update time of $\tilde{O}(mn)$, $O(1)$ query time and $\tilde{O}(n^2)$ space. The result of Roditty and Zwick [25] was later derandomized by Henzinger *et al.* at the cost of a slightly larger query time of $O(\log\log n)$. Moreover, Bernstein [7] generalized this result for directed weighted graphs, at the cost of an additional $\log(nW)$ factor for the total update time. Roditty and Zwick [25] also presented a decremental APSP algorithm with $\tilde{O}(mn)$ total update time, $(2k-1)$ stretch and $O(k)$ query time with smaller space of $O(m+n^{1+1/k})$. Bernstein and Roditty [11] later presented a decremental APSP for unweighted undirected graphs with $(2k-1+\epsilon)$ stretch, $O(k)$ query time and total update time of $\tilde{O}(n^{2+1/k+O(1)/\sqrt{\log n}})$. Note that all the above mentioned results do not break the $O(mn)$ total update time barrier for sparse graphs even for a very large stretch. Attempts were also made to break this $O(n^2)$ barrier. There are two previous results that managed to get decremental APSP algorithm beyond the $O(n^2)$ barrier, however these results have a huge stretch especially as we approach near linear total update time. Abraham, Chechik, and Talwar [1] presented a decremental all-pairs shortest paths for undirected unweighted graphs with stretch $2^{O(\rho k)}$ in total update time $\tilde{O}(mn^{1/k})$ and with $O(k\rho)$ time per query, where $\rho = (1+\lceil\frac{\log n^{1-1/k}}{\log(m/n^{1-1/k})}\rceil)$. The near linear decremental SSSP result of Henzinger, Krinninger and Nanongkai [17] allowed them also to obtain a decremental all-pairs shortest paths (APSP) for undirected weighted graphs with $(2+\epsilon)^k-1$ stretch, $O(k^k)$ query time and $O(m^{1+1/k+o(1)}\log^2(nW))$ total update time.

In the static regime of computing approximate APSP efficiently, Thorup and Zwick [29] in a seminal paper presented a distance oracle with $O(k)$ query time to, $2k-1$ stretch, $O(kn^{1+1/k})$ expected space, and $O(kmn^{1/k})$ construction time. Baswana and Kavitha [3] addressed the question of improving the construction time for dense graphs. Their construction time is $O(n^2\log n)$ and with query time of $O(k)$ for $k>2$ and of $\Theta(\log n)$ for $k=2$. Baswana *et al.* [2] presented a distance oracle with subquadratic construction time when $m=o(n^2)$ at the cost of introducing additional additive stretch. Wulff-Nilsen [30] further improved the construction time for dense graphs. He presented a distance oracle with $2k-1$ stretch, $O(kn^{1+1/k})$ size, $O(k)$ query time and $O(\sqrt{k}m+kn^{1+c/\sqrt{k}})$ construction time for some absolute constant c.

Our result: We present a decremental APSP algorithm for undirected weighted graphs with $(2+\epsilon)k-1$ stretch, $O(mn^{1/k+o(1)}\log(nW))$ total update time and $O(\log\log(nW))$ query time, for any integer parameter k. Our algorithm is randomized and assumes an oblivious adversary, that is, the future sequence of edge deletions do not depend on the algorithms past queries. The assumption of an oblivious adversary is very common when dealing with randomized dynamic algorithms and in particular previous approximate dynamic APSP algorithms [1], [17] were also randomized and assumed an oblivious adversary. Our result is an exponential improvement both in the stretch and in the query time over previous work [1], [17]. Moreover, our total update time almost match the best known bounds in the static case (up to poly logarithmic factors and dependence on ϵ), e.g. the bounds by Thorup and Zwick distance oracle [29], at the cost of an additional $1+\epsilon$ factor in the stretch. Note that though some improvements were made on the construction time of Thorup and Zwick distance oracle [29], these improvements are for some range of graph density and k. In particular, for sparse graph the $O(kmn^{1/k})$ construction time of Thorup and Zwick distance oracle [29] is still the state of the art.

In previous dynamic approximate shortest paths algorithms there were essentially two approaches used. The first was used by Roditty and Zwick [24] and was based on dynamically maintaining the Thorup-Zwick distance oracle. In the Thorup-Zwick distance oracle a collection of trees is maintained, where every node belongs only to a small number of trees. One of the technical challenges in maintaining the Thorup-Zwick distance oracle in the dynamic setting is that during the course of the algorithm, a node may join many trees. Recall that in the Even-Shiloach (**ES**) algorithm we pay the degree $\deg(v)$ of the node v each time its distance from the root increases. We therefore pay for each node the degree times the maximum distance d. Roditty and Zwick's algorithm [24] is based on the following two claims. First, if a node belongs to a tree T but it joins the tree at distance i from the root and leaves at distance j from the root, the total update time spent on v between the time it joins the tree and the time it leaves the tree is $O(\deg(v)(j-i))$ (rather than $O(\deg(v)\cdot d)$). Second, for each specific distance i such that $1\leq i\leq d$, the number of trees T such that v is part of and is at distance i

from the root of T is bounded by $O(n^{1/k})$. Combining these two claims, Roditty and Zwick [24] obtained their decremental approximate APSP with $(2k-1)$ stretch and $O(m \cdot n^{1/k} \cdot d)$ total update time for distances up to d.

The second approach [1], [17] used a different clustering approach than the Thorup-Zwick distance oracle that bounds better the total number of trees a node may belong to during the entire course of the algorithm. Specifically, [1], [17] show a construction that maintains a collection of trees such that every node v is part of at most $\tilde{O}(n^{1/k})$ distinct trees during the entire course of the algorithm. For each such tree T, [1], [17] invoke Monotone Even-Shiloach algorithm on an emulator, which requires $O(|E(T)|^{1+o(1)})$ total update time. Where an emulator is a graph (not necessarily a subgraph) that approximately preserves distances of the original graph G. The emulator of [1], [17] has the key property that every two vertices have a path P in the emulator of length "close" to the length of their shortest in G and in addition P consists of a "small" number of edges. This gave a total update time of $O(mn^{1+1/k+o(1)})$. However, in order to guarantee this nice property that every node during the entire course of the algorithm is contained in a small number of trees, previous constructions [1], [17] had to pay a huge exponential in k stretch, this large stretch seems to be essential in previous constructions in order to allow small overlap between the trees.

In this paper, we essentially combine these two approaches by introducing a new clustering approach. Instead of using the Thorup-Zwick distance oracle and trying to maintain it dynamically, we rather suggest a new clustering approach that loosely speaking starts by handling dense areas and slowly moves to sparser and sparser areas while ignoring dense areas that were covered before. This approach uses hierarchical truncated trees. The idea of truncated trees appeared in somewhat different context [5] by Baswana *et al.* where the goal was to handle a single failure in the context of distance oracles. We extend on this idea and show how to maintain such efficient clustering in the dynamic setting. This approach is non trivial to maintain in the dynamic setting while keeping both the stretch and running time low as dense areas might become sparse at some point. We show that this clustering approach obtains similar bounds as in the Thorup and Zwick distance oracle but with the additional crucial property that in our construction every node belongs to at most $O(n^{1/k})$ trees during the entire course of the algorithm. This crucial property allows us to maintain monotone Even-Shiloach on an emulator in each such tree.

The use of monotone Even-Shiloach algorithm on an emulator rather than simply invoking Even-Shiloach algorithm on the graph greatly complicates the analysis. We present a new construction of a dynamic emulator that keeps the hop-set of the graph low and maintains some other key properties. This construction is somewhat similar to the one given by Henzinger, Krinninger and Nanongkai [17], where they present a $(1+\epsilon)$ decremental single source shortest paths (SSSP) in near linear total update time of $O(m^{1+o(1)} \log(nW))$. We note that it is likely that we can combine our new clustering approach with Henzinger *et al.* emulator [17]. However, we find the emulator presented in this paper slightly more direct and appropriate for the dynamic setting. Moreover, our dynamic emulator is much sparser and contains only $O(n^{1+o(1)})$ edges whereas the emulator of [17] contained $O(m^{1+o(1)})$ edges. We believe that this emulator with the improved size could be of independent interest. Roughly speaking, the emulator of Henzinger *et al.* starts by using the static Thorup-Zwick [29] distance oracle and maintains some properties of it dynamically. This complicated the construction and analysis of the emulator, as it uses some heavy structure of clusters and bunches. We rather present a more direct approach for maintaining an emulator by presenting a different static emulator and dynamically maintaining this emulator. Note that there are essentially two clustering used in this paper and in [17], one for maintaining the emulator itself and the second is for maintaining shortest paths trees on the emulator.

Our emulator will also give similar bounds for the Decremental SSSP of $O(m^{1+o(1)} \log(nW))$ total update time.

II. Preliminaries

We let $G = (V, E)$ to always refer to the *current* version of the graph (the graph after all deletions occurred so far). Let $\textbf{dist}(u, v, H)$ for nodes u and v and a graph H be the distance between u and v in H. When $H = G$ we simply write $\textbf{dist}(u, v)$, that is, $\textbf{dist}(u, v) = \textbf{dist}(u, v, G)$. Let $\textbf{dist}(u, A)$ for a node u and a subset A of the nodes be the minimal distance $\textbf{dist}(u, v)$ for $v \in A$. For a node v and distance d, let $B(v, d)$ be all nodes at distance at most d from v, namely, $B(v, d) = \{u \in V \mid \textbf{dist}(u, v) \leq d\}$, and, let $B^-(v, d)$ be all nodes at distance strictly less than d from v, namely, $B^-(v, d) = \{u \in V \mid \textbf{dist}(u, v) < d\}$. For a graph H, let $V(H)$ be the nodes in H and let $E(H)$ be the edges of H. For a weighted graph H and an edge $(u, v) \in E(H)$ let $\omega(u, v, H)$ be the weight of the edge (u, v) in H.

We next state the classic decremental SSSP of Even and Shiloach [14] that was later generalized by King [22] to directed weighted graphs.

Lemma 2.1: [14], [22] Let $G = (V, E)$ be a dynamic weighted graph and s be a fixed source such that all weights in G are multiples of β for some parameter β. The Even-Shiloach tree $\textbf{ES}(G, s, d)$ decrementally maintains single source shortest paths tree from s up to distance d in total update time of $O(m \cdot d/\beta)$.

Remark 2.2: The result of the ES-tree is usually stated without the β term. That is, maintaining SSSP for graph with integer weights takes $O(m \cdot d)$ total update time. However, if all edge weights are multiples of some term β, one can simply reduce to the previous case by diving all edge weights by β (and returning the distance returned by the ES-tree multiply by β in the query phase). This observation is due to Bernstein [6].

The classic **ES** algorithm can only handle edge deletions. Or to be more precise, it can handle both edge deletions and edge insertions under the assumption that distances *never* decrease. Henzinger *et al.* [16] developed a modification of this algorithm which can handle occasional distance decreases, which they called the Monotone Even-Shiloach algorithm, denoted **MES**(monotone Even-Shiloach). The basic idea is that **MES** simply ignores distance decreases. More precisely, the classic **ES** algorithm from a root w maintains for every vertex v a distance label $\hat{d}(w,v)$ with the guarantee that we always have $\hat{d}(w,v) = \mathbf{dist}(s,v)$. **MES**, on the other hand, will only guarantee that $\hat{d}(w,v) \geq \mathbf{dist}(s,v)$. It does so by running classical **ES**, with one modification: whenever classic **ES** adds an edge (u,v) to the shortest path tree, it sets $\hat{d}^{\text{NEW}}(w,v) = \min\left\{\hat{d}(w,u)+\omega(u,v), \hat{d}^{\text{OLD}}(w,v)\right\}$. **MES**, on the other hand, sets $\hat{d}^{\text{NEW}}(w,v) = \max\left\{\hat{d}(w,u)+\omega(u,v), \hat{d}^{\text{OLD}}(w,v)\right\}$. See the full version for further discussion on Even-Shiloach and monotone Even-Shiloach.

Definition 2.3: Let **MES**(G,s,d) refer to running monotone Even-Shiloach from a fixed source s up to distance d. In particular, whenever we have $\hat{d}(s,v) > d$, we remove v from the graph.

Note that **MES** does not guarantee any approximation ratio that would work for an arbitrary sequence of insertions and deletions. Every invocation of **MES** will require a separate approximation error analysis to show that for the particular graph and update sequence at hand, $\hat{d}(s,v)$ remains a good approximation to $\mathbf{dist}(s,v)$. However, the asymptotic bound of the running time of **MES** is similar to **ES**.

To ease presentation, we show algorithms with $(2 + O(\epsilon))k - 1$ stretch rather than $(2+\epsilon)k-1$ stretch, to get the desired $(2+\epsilon)k-1$, one can simply invoke the algorithm with $\epsilon' = \epsilon/c$ for large enough constant c. This will only increase the running time bound by a constant factor. In addition, to simplify the presentation we present our result for the unweighted case and with slightly larger query time. In the full version we explain the modifications needed to handle weighted graphs and how to improve the query time.

The rest of the paper is organized as follows. In Section III we present a simpler version of our algorithm that uses exact **ES** algorithm with larger total update time of $\tilde{O}(mn^{1+1/k})$. In Section IV we show how the use of an emulator and **MES** with the techniques from Section III can reduce the total update time to $O(mn^{1/k+o(1)})$. In the full version we give a detailed explanation of our emulator. For simplicity, our data structure returns distances rather than paths. Our data structure can also be tweaked to return paths (in additional time proportional to the number of edges on the path). Some of the proofs are deferred to the full version.

III. Warmup: APSP Algorithm with exact Even-Shiloach and High Level Idea of our Clustering Approach

We present a construction that for a given distance d runs in total update time of $\tilde{O}(mn^{1/k} \cdot d)$ and for every query (s,t) returns a distance $\widehat{\mathbf{dist}}(s,t)$ such that $\mathbf{dist}(s,t) < \widehat{\mathbf{dist}}(s,t)$ and in addition if $\mathbf{dist}(s,t) < d$ then $\widehat{\mathbf{dist}}(s,t) \leq ((2+\epsilon)k-1)d$. To get the decremental $((2+\epsilon)k-1)$-APSP we simply invoke this construction on every $d = (1+\epsilon)^i$ for $1 \leq i \leq \log n$ and in the query phase the algorithm invokes the query algorithm for all distances $(1+\epsilon)^i$ for $1 \leq i \leq \log n$ and returns the minimum distance found. This incurs an additional $\log n$ factor to the total update time and $\log n$ factor to the query time (that can be reduced to $\log\log n$ factor using binary search).

High Level Intuition of Our Clustering Approach: The main novelty of our clustering approach is that it can be easily adapted to the dynamic setting while maintaining the crucial property that each node is added to a small number of clusters during the entire course of the algorithm. Our clustering approach works for a given distance d and loosely speaking works as follows.

The algorithm samples subset of the nodes A_i uniformly at random such that each subset A_i contains in expectation $\tilde{O}(n^{(k-i)/k})$ nodes for $0 \leq i \leq k-1$, by sampling every node independently at random with $\tilde{O}(1/n^{i/k})$ probability. For each node $w \in A_i$, the algorithm computes and maintains the cluster $C(w)$ loosely defined as follows. A node $v \in C(w)$ if the following two conditions hold. First, the distance from w to v is at most $O(k \cdot d)$ (the constant in the $O(k \cdot d)$ term will be fixed later on). Second, there is no node $w' \in A_j$ for $j > i$ such that $\mathbf{dist}(w',v) \leq \mathbf{dist}(w,v) + \epsilon(j-i)d$. Let $p_i(v)$ be the closest node to v in A_i. (For comparison, in the Thorup-Zwick distance oracle[29] the cluster $C_{TZ}(w)$ for $w \in A_i \setminus A_{i+1}$ is defined as $C_{TZ}(w) \leftarrow \{v \in V \mid \mathbf{dist}(v,w) < \mathbf{dist}(v,p_{i+1}(v))\}$).

We will later see that returning $\mathbf{dist}(p_i(s),t)+\mathbf{dist}(p_i(s),s)$ or $\mathbf{dist}(p_i(t),s)+\mathbf{dist}(p_i(t),t)$ for the minimal index i such $t \in C(p_i(s))$ or $s \in C(p_i(t))$ gives the desired stretch.

We also need to show that we can maintain these clusters efficiently. In the static case, a similar argument to the one

used in the Thorup-Zwick distance oracle[29], can show that w.h.p. $B^-(v, \textbf{dist}(v, p_{i+1}(v)))$ contains at most $n^{(i+1)/k}$ nodes. To see this, consider the nodes in increasing order of their distance from v. As every node in A_{i+1} is sampled with probability $\tilde{O}(1/n^{(i+1)/k})$ then by Chernoff bound w.h.p. A_{i+1} contains a node in the first $n^{(i+1)/k}$ nodes of that order. This implies that $B^-(v, \textbf{dist}(v, p_{i+1}(v)))$ contains at most $n^{(i+1)/k}$ nodes w.h.p. Note that this claim is true w.h.p. in any given time of algorithm. It follows that the number of nodes $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$ is $\tilde{O}(n^{1/k})$ in expectation. In the static version, it is therefore sufficient to simply add v to the cluster of all nodes in $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$. In the dynamic regime, things are more complicated as it might be that after some time the distance $\textbf{dist}(v, A_{i+1})$ increases even just by 1 and the new set of nodes $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$ is completely different than the previous set. Hence, adding v to all these clusters might cause v to join to too many clusters during the entire course of the algorithm. To overcome this, we are a bit more careful with the clusters we add v to. Instead of adding v to all clusters of $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$, we only add v to the clusters of nodes $w \in A_i$ that are much closer to w than to $p_{i+1}(v)$ by at least ϵd. It follows that the first time v will be added to the clusters $w \in A_i$ such that w is not in $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$ is when $\textbf{dist}(v, p_{i+1}(v))$ increases by at least ϵd. Since our clusters are trimmed by distance $O(kd)$, this can happen at most $O(k/\epsilon)$ times. Therefore, each time $\textbf{dist}(v, p_{i+1}(v))$ increases by at least ϵd, the entire ball $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$ may change completely but we still have the property that the new set $B^-(v, \textbf{dist}(v, p_{i+1}(v))) \cap A_i$ contains $\tilde{O}(n^{1/k})$ nodes. In other words, our analysis is roughly as follows. Let $d^1_{i+1}(v) = \textbf{dist}(v, p_{i+1}(v))$ be the initial distance between v and $p_{i+1}(v)$. Note that, initially we have $|B^-(v, d^1_{i+1}(v)) \cap A_i| \leq \tilde{O}(n^{1/k})$ and that as long as $\textbf{dist}(v, p_{i+1}(v)) \leq d^1_{i+1}(v) + \epsilon d$, v may join the clusters of only nodes in $B^-(v, d^1_{i+1}(v)) \cap A_i$. Once $\textbf{dist}(v, p_{i+1}(v)) > d^1_{i+1}(v) + \epsilon d$, we define $d^2_{i+1}(v) = \textbf{dist}(v, p_{i+1}(v))$ (note that $d^2_{i+1}(v) > d^1_{i+1}(v) + \epsilon d$). Note also that by the same reasoning as before we now have $|B^-(v, d^2_{i+1}(v)) \cap A_i| \leq \tilde{O}(n^{1/k})$ and that as long as $\textbf{dist}(v, p_{i+1}(v)) \leq d^2_{i+1}(v) + \epsilon d$ v may only join clusters of nodes in $B^-(v, d^2_{i+1}(v)) \cap A_i$ and so on. As the tree is trimmed at distance $O(kd)$, this process can continue $O(k/\epsilon)$ times and each time we show that v may join the clusters of at most $\tilde{O}(n^{1/k})$ nodes.

The construction for distance d: The algorithm computes a collection $A_0, A_1, ..., A_{k-1}$ of subsets of the nodes as follows. Each A_i is obtained by independently sampling every node in V with probability $\min\{1, c \log n / n^{i/k}\}$, where c is a large enough constant. We have $\mathrm{E}[|A_i|] = O(\log n \cdot n^{(k-i)/k})$ for every $1 \leq i \leq k-1$. Note also that $A_0 = V$.

The algorithm maintains for every node $w \in A_i$, a subset $C(w)$ of the nodes, hereafter referred to as the cluster of w (where some nodes may be added to or removed from this subset during the course of the algorithm). In addition, it maintains an **ES** tree $T(w)$ for every node $w \in A_i$ on $C(w)$. Let $\hat{d}(w, v)$ be the distance between w and v in the **ES** tree $T(w)$. We will later see that $\hat{d}(w, v) = \textbf{dist}(w, v)$.

The cluster of a node $w \in A_i$ is defined as follows. Every node $v \in V$ belongs to $C(w)$ if the following two conditions hold. First, there is no node $w' \in A_j$ for $j > i$ such that $\textbf{dist}(w', v) \leq \textbf{dist}(w, v) + \epsilon(j - i)d$. Second, $\textbf{dist}(w, v) \leq (1+\epsilon)(i+1)d$. Note that w may belong to more than one A_i. In this case we define $C(w)$ according to the smallest index i such that $w \in A_i$. For every node v and index i the algorithm also stores and maintains $p_i(v)$, the closest node in A_i

The way to efficiently construct and maintain the trees is a bit technical, we therefore due to space limitations is deferred it to the full version (together with most of the proofs). The important take-away is that we can construct and maintain the trees $T(w)$ such that $V(T(w)) = C(w)$ in the desired total update time.

The query algorithm: Given are two nodes s and t. Find the minimal index i such that either $s \in T(p_i(t))$ or $t \in T(p_i(s))$. If no such index is found return ∞. Otherwise, if $s \in T(p_i(t))$ then return $\hat{d}(p_i(t), s) + \hat{d}(p_i(t), t)$ else return $\hat{d}(p_i(s), t) + \hat{d}(p_i(s), s)$.

Analysis:

Lemma 3.1: Consider $w \in A_i$. If a node v belongs to $C(w)$ then every node z on the shortest path $P(v, w)$ from v to w is also in $C(w)$.

The next auxiliary claim is a simple corollary of the definition of clusters.

Claim 3.2: Consider a node v. Let $\textbf{dist}(v, A_{i+1}) = d'$. For every $w' \in A_i$ such that $\textbf{dist}(v, w') \geq d' - \epsilon d$ we have $v \notin C(w')$.

The next lemma bounds the number of trees $T(w)$ to which a node v ever belonged to.

Lemma 3.3: Consider a node v. During the entire running time of the algorithm, the total number of distinct clusters v ever joined to is $O(\log n \cdot k^2 \cdot n^{1/k}/\epsilon)$ in expectation.

The next lemma bounds the total running time of the algorithm.

Lemma 3.4: The expected total update time of the algorithm is $\tilde{O}(mn^{1/k}d)$.

The next lemma bounds the stretch.

Lemma 3.5: Consider two nodes s and t. If $\textbf{dist}(s, t) \leq d$ then the algorithm returns at most $(2(1+\epsilon)k - 1)d$.

Proof: Consider two nodes s and t whose distance is at most d.

To prove the claim, notice that we only need to prove that there is an index i such that either 1. $t \in T(p_i(s))$ and $\mathbf{dist}(s, p_i(s)) \leq (1+\epsilon)i \cdot d$ or 2. $s \in T(p_i(t))$ and $\mathbf{dist}(t, p_i(t)) \leq (1+\epsilon)i \cdot d$. To see this, recall that the depth of all trees is at most $(1+\epsilon)kd$ and that $i \leq k-1$. Consider the maximal index i such that either $\mathbf{dist}(s, p_i(s)) \leq (1+\epsilon)i \cdot d$ or $\mathbf{dist}(t, p_i(t)) \leq (1+\epsilon)i \cdot d$. (note that there is such an index as $\mathbf{dist}(s, p_0(s)) = \mathbf{dist}(s, s) = 0$ as all nodes belong to A_0). Assume w.l.o.g. that for this index i we have $\mathbf{dist}(s, p_i(s)) \leq (1+\epsilon)i \cdot d$. Note that by triangle inequality, $\mathbf{dist}(t, p_i(s)) \leq (1+\epsilon)i \cdot d + d$. Assume, towards a contradiction, that $t \notin T(p_i(s))$. As $V(T(p_i(s))) = C(p_i(s))$ we also have $t \notin C(p_i(s))$. By definition of $C(p_i(s))$ we have that there exists a node $w' \in A_j$ for some $j > i$ such that $t \in T(w')$ and $\mathbf{dist}(t, w') \leq (1+\epsilon)i \cdot d + d + \epsilon(j-i)d \leq (1+\epsilon) \cdot j \cdot d$. Since $\mathbf{dist}(t, p_j(t)) \leq \mathbf{dist}(t, w')$ we get a contradiction to the maximality of i. Hence, $t \in T(p_i(s))$. ∎

IV. Improving the running time using an Emulator

In this section we show how to reduce the total update time to $O(mn^{1/k+o(1)})$. The main difference with the previous algorithm is that here we do not use exact **ES** but rather invoke an **MES** on our Emulator. The emulator approximates the distances in the graph G. In addition, for every shortest path in G, the emulator has an alternative shortest path that closely approximates the distances in G and has a small number of edges. As already shown in previous papers (see e.g. [7]) maintaining decrementally shortest path up to a bounded hop-length can be done efficiently, where a hop-length of a path is the number of edges on it.

A. Main properties required from the Emulator

We next summarize the main properties we need from this emulator.

We need first to set some parameters. We set $t = \sqrt{\log n}$ and $\epsilon_2 = \epsilon/(ck \log n)$ for large enough constant c. Our emulator uses subsets of the nodes $Z_0, .., Z_{t-1}$, obtained as follows. For every i such that $0 \leq i \leq t-1$, obtain a set Z_i by sampling every node with probability $\min\{1, c \log n / n^{i/t}\}$ for some sufficiently large constant c. To slightly simplify the analysis we also add to Z_i all sets Z_j for $j > i$. Note that $\mathrm{E}[|Z_i|] = O(\log n \cdot n^{(t-i)/t})$ and that $Z_0 = V$.

The following definition summarizes the properties of the emulator and it uses the sets $Z_0, .., Z_{t-1}$. Note that the sets $Z_0, .., Z_{t-1}$ are not the same as sets $A_0, ..., A_{k-1}$ from previous section and is obtained by using a different sample realizations (in particular we might have $t \neq k$).

Definition 4.1 ($(\tilde{d}, \epsilon_1, \epsilon_2)$-emulator): We say that a dynamic set of edges E^* is a $(\tilde{d}, \epsilon_1, \epsilon_2)$-emulator if the following conditions hold.
(1) For every i such that $0 \leq i \leq t-1$, there is a subset T_i of the nodes Z_i with the following properties. For every node $w \in T_i$, E^* has edges from w to all nodes v such that $\mathbf{dist}(w, v) \leq \lceil \tilde{d} \cdot (6/\epsilon_2)^{i+1}/2 \rceil$ (and perhaps to some other nodes as well). If a node $w \in Z_i \setminus T_i$ then there is a node $w' \in T_j$ for some $j > i$ such that the distance from w to w' is at most $3\tilde{d}(6/\epsilon_2)^j$. In this case we say that w' *covers* w.
(2) The weight of all edges satisfy the following:
(2.1) The weight of all edges in E^* are of at least the distance in G.
(2.2) The weight of all edges in E^* is with at most $1+\epsilon_1$ stretch or at most $8\tilde{d}$. That is, for every edge $(w, v) \in E^*$ we have that either $\omega(w, v, E^*) \leq (1+\epsilon_1)\mathbf{dist}(w, v)$ or $\omega(w, v, E^*) \leq 8\tilde{d}$.
(2.3) The weight of all edges are multiples of $\epsilon_2 \tilde{d}$.

Property (1) is the property that is in charge on having long enough shortcuts, where a shortcut is a direct edge in the emulator of weight that is close to the distance between its endpoints. We will later want to consider some shortest path P and show that we can partition the path P into intervals where each interval (perhaps but the last one) is of length at least $\tilde{d}$ and each such interval has an alternative path in the emulator that consists of a constant number of hops. Consider a node $w \in Z_i$ on P, by Property (1) either w itself had shortcuts in E^* to all nodes at distance $\lceil \tilde{d} \cdot (6/\epsilon_2)^{i+1}/2 \rceil$ from it and in particular to the node at distance $\lceil \tilde{d} \cdot (6/\epsilon_2)^{i+1}/2 \rceil$ from it on the path P. Otherwise, there is a node $w' \in T_j$ for some $j > i$ such that the distance from w to w' is quite small (compare to the length of the maximal shortcuts of w'), that is, at most $3\tilde{d}(6/\epsilon_2)^j$ and w' has shortcuts to all nodes at distance at most $\lceil \tilde{d} \cdot (6/\epsilon_2)^{i+1}/2 \rceil$ from it. We can therefore pick a node z at distance $O(\tilde{d} \cdot (6/\epsilon_2)^{i+1})$ from w on P and show that w has a two hops path to z (through w'). The length of the 2-hop path is close to the shortest path as the distance from w to w' is quite small compare to the distance from w to z. Using this reasoning we can show that we can partition the path P into intervals with the properties mentioned above.

Property (2.1) verifies that the emulator never shrinks distances. Property (2.2) verifies that the distances in the emulator are close to the shortest paths (notice that using the emulator we lose in the approximation ratio both because the edges in the emulator do not represent shortest paths but rather good approximation to the shortest paths and second because in order to get bounded hop paths we need to deviate from the shortest path to close by nodes (e.g., the node w' in the explanation above)). Property (2.3) is used for the efficiently of invoking **MES** on the emulator - we will later invoke the emulator up to distance $\tilde{d} \cdot n^{o(1)}$. If all distances are

multiples of $\epsilon_2\tilde{d}$ we can divide all weights by $\epsilon_2\tilde{d}$ and have the depth of the tree be $n^{o(1)}$ which is much more efficient than maintaining the tree up to depth $\tilde{d} \cdot n^{o(1)}$.

1) Some Intuition about the construction of the emulator.: The construction algorithm gets a dynamic graph G', a target distance d and a parameter ϵ. Roughly speaking, the goal of the emulator is to produce a dynamic graph G_{out} that handles distances $d_{out} = d \cdot n^{o(1)}$ (for large enough $n^{o(1)}$ - we will have the term $n^{o(1)}$ to be roughly $O(c/\epsilon)^{\sqrt{\log n}}$ for some constant c). As we will later see, the minimal edge weight in G_{out} is d and all edge weights are multiples of ϵd. This will ensures that invoking **MES** in the graph G_{out} up to distance $d \cdot n^{o(1)}$ takes $O(|E(G_{out})|n^{o(1)})$ time.

Loosely speaking, we want that every node to either have in the emulator shortcuts to all nodes at distance d from it or to have a 2-hop path to every node in a far enough distance. Consider a node v. If the number of nodes at distance at most d from v is small enough then the algorithm can simply add shortcuts (i.e. edges) between v and all nodes at most d from it. Otherwise, the algorithm can group all nodes in $B(v, d)$ together and look at all the nodes at distance d/ϵ from v. If the algorithm adds shortcuts from v to all nodes at distance at most d/ϵ from v then all nodes in $B(v, d)$ will be satisfied, in the sense that they have 2-hop $(1 + \epsilon)$-approximate shortest paths to all nodes at distance roughly d/ϵ from them. Since now more nodes will be satisfied by adding these shortcuts, the algorithm can allow adding more shortcuts. If the ball $B(v, d/\epsilon)$ is still too dense we continue looking on the ball $B(v, d/\epsilon^2)$ and so on. Each time we look at larger and larger balls but we can add more and more shortcuts (as more and more nodes will be satisfied by these shortcuts). This process continues until there is a sparse enough ball $B(v, d/\epsilon^i)$ or the ball makes all other nodes satisfy. By allowing the number of shortcuts to be roughly $n^{1/\sqrt{\log n}}$ larger than the number of nodes that are satisfied by these shortcuts (that is, looking for the minimal index i such that $|B(v, d/\epsilon^{i+1})| \leq |B(v, d/\epsilon^i)|n^{1/\sqrt{\log n}}$), in each such iteration either the algorithm finds a sparse enough ball or the number of nodes in the ball increases by a $n^{1/\sqrt{\log n}}$ factor. This guarantees that the maximal distance considered until finding a sparse enough ball is at most $d/(\epsilon^{\sqrt{n}})$.

This approach works well in the static case. However, in the dynamic case fixing the center of the ball, i.e., the node v can be problematic as the adversary can disconnects v from the graph (by deleting all of its incident edges) making all these shortcuts to be useless.

To overcome this, we use randomization. The random sets $Z_0, ..., Z_t$ are supposed to take care of the different density of the nodes in the following sense. The algorithm maintains **MES** trees from subsets of the nodes (referred to as i-active nodes) in $Z_0, ..., Z_{t-1}$, such that the depth of the trees of nodes in Z_{t-1} is the largest and the depth gets smaller and smaller for trees of nodes $w \in Z_i$ as i decreases. The depth of trees $w \in Z_i$ is $d(c/\epsilon)^{i+1}$. The algorithm adds to the emulator shortcuts from w to all nodes in its tree. We say that a node v is covered by w if the distance from v to w is at most $c' \cdot d(c/\epsilon)^i$ for some constant c'. Note that in this case v has a 2-hop paths (through w) from it to far enough nodes (at distance $d(c/\epsilon)^{i+1}$) and in addition these 2-hop paths to far enough nodes are close to the original distances as the distance from v to w is at most $c' \cdot d(c/\epsilon)^i$ which is much smaller than $d(c/\epsilon)^{i+1}$. A node $w \in A_i$ is i-active if either $i = k - 1$ or there is no node $w' \in Z_j$ for $j > i$ that covers w.

Consider a vertex v and let i be the maximal index such that the ball around v of radius $d(6/\epsilon)^i$ (taking $d(6/\epsilon)^i$ rather than $d(1/\epsilon)^i$ is done for technical reasons) contains more than $n^{i/t}$ nodes. The set Z_i is supposed to take care of v in the following sense. W.h.p., Z_i contains a node w in the ball $B(v, d(1/\epsilon)^i, G')$. The algorithm maintains **MES** from w up to distance $d(1/\epsilon)^{i+1}$ and therefore v is covered. To see that v does not belong to too many trees, note that for any $j \geq i$, the only nodes $w \in Z_j$ that v may belong to their trees are nodes at distance $d(1/\epsilon)^{j+1}$ from it, but by the maximality of i there are less than $n^{(j+1)/t}$ such nodes. Moreover, as each node in added to Z_j independently with probability $\tilde{O}(1/n^{j/t})$, then in expectation there shouldn't be more than $\tilde{O}(n^{1/t})$ nodes in $B(v, d(1/\epsilon)^{j+1}, G') \cap Z_j$. Moreover, (by setting the parameters right) all nodes in $B(v, d(1/\epsilon)^i, G')$ are also covered by w and therefore will not be j-active for any $j < i$. For $j < i$, the only nodes $w \in Z_j$ that v may belong to their trees are nodes at distance $d(1/\epsilon)^{j+1} \leq d(1/\epsilon)^i$ from it, but as mentioned above all these nodes are already covered and therefore will not be j-active. The nice thing in this approach is that it works not only in the static case but also in the dynamic regime. The index i described above may change, more precisely it may decrease as balls around v may become sparser, but at any given time w.h.p v belongs only to few trees of nodes $w \in Z_j$ for $j \geq i$ and to no tree of a node $w \in Z_j$ for $j < i$.

However, in the dynamic regime, this approach by itself is not enough. We want to take care of any distance and in order to do it we need to invoke the **MES** for a large distance, which will be too slow (even just for maintaining one **MES**). To overcome this, we maintain a hierarchy of dynamic graphs $Em(E', d_i, \epsilon_2)$ for exponentially growing distances d_i. Each emulator $Em(E', d_i, \epsilon_2)$ is based on the previous emulator $Em(E', d_{i-1}, \epsilon_2)$ making larger and larger shortcuts. This makes the construction and analysis quite technical.

B. The Construction using the Emulator

Similarly to Section III, we focus on a construction that for a given distance d runs in total update time of $\tilde{O}(mn^{1/k+o(1)})$

and for every query (s,t) returns a distance $\widehat{\mathbf{dist}}(s,t)$ such that $\mathbf{dist}(s,t) < \widehat{\mathbf{dist}}(s,t)$ and in addition if $\mathbf{dist}(s,t) < d$ then $\widehat{\mathbf{dist}}(s,t) \leq ((2+\epsilon)k-1)d$. To get the decremental $((2+\epsilon)k-1)$-APSP we again invoke this construction for every distance $d = (1+\epsilon)^i$ for $1 \leq i \leq \log n$ and in the query algorithm returns the minimum distance found by one of these data structures.

We set some parameters $d_i = (6/\epsilon_2)^{i \cdot t}$. In the full version we show how to compute dynamic graphs G_i for $1 \leq i \leq \log n$ such that each dynamic graph G_i is a $(d_i, (1+\epsilon_2)^i, \epsilon_2)$-emulator.

The construction for distance d: The construction is very similar to the construction presented in the previous section. The sets $A_0, ..., A_{k-1}$ are obtained exactly as in the previous section and the pivots $p_i(v)$ for $0 \leq i \leq k-1$ are defined the same.

Let μ be the maximal index such that $d_{\mu+1} < d$. d_μ will be the target distance on which we use the emulator. Note that on one hand, we want the target distance of the emulator to be as close as possible to d so we have long enough shortcuts in the emulator and on the other hand we have to have some gap between the target distance of the emulator and d as the additive stretch in the emulator depends on the target distance.

We set $\ell_i = d_\mu(6/\epsilon_2)^{i+1}$ and $\beta_2 = 8\ell_{t-2} + 8d_\mu$. We thus have

$$\begin{aligned} \beta_2 &= 8\ell_{t-2} + 8d_\mu < 9\ell_{t-2} = 9d_\mu(6/\epsilon_2)^{t-1} \\ &< d_{\mu+1} \cdot \epsilon_2 < \epsilon d_{\mu+1}/(k \cdot \log n) \leq \epsilon d/(k \cdot \log n). \end{aligned}$$

Construct the $(d_\mu, (1+\epsilon_2)^\mu, \epsilon_2)$-emulator G_μ. The algorithm maintains **MES** trees $T(w)$ in the graph G_μ for a subset of the nodes $w \in A_i$. Some nodes may be added or removed (perhaps multiple times) to the tree during the algorithm. Let $\hat{d}(w,v)$ be the distance between w and v in the monotone tree $T(w)$ (or ∞ in case $v \notin T(w)$).

Since we are using an emulator where both insertions and deletions are allowed, we have to be a bit more careful in order to maintain monotonicity. Specifically, the algorithm stores $\hat{d}_{last}(w,v) = \hat{d}(w,v)$ - the last distance in the **MES** of $T(w)$ assigned to v. If v later joins again $T(w)$, the distance assigned to it will be at least the previous distance $\hat{d}_{last}(w,v)$ (if v was never in $T(w)$ then $\hat{d}_{last}(w,v)$ is 0).

We say that v is w-close if there is an edge (z',v) such that $z' \in T(w)$ and there is no index $j > i$ such that $\hat{d}(p_j(v),v) \leq \hat{d}(w,z') + \omega(z',v) + \epsilon(j-i)d$ and in addition $\hat{d}(w,z') + \omega(z',v) \leq (1+2\epsilon)(i+1) \cdot d$.

The trees of $w \in A_{k-1}$ initially contain all nodes at distance at most $(1+2\epsilon)k \cdot d$ from w and are maintained by maintaining **MES** tree from w up to distance $(1+2\epsilon)k \cdot d$ in the graph G_μ. Assume all trees on level i or higher were constructed and consider $w \in A_i$. The tree of w is constructed as follows. Initially, $w \in T(w)$. The algorithm maintains in a heap H all nodes that already have a neighbor in the constructed tree $T(w)$ (and were not previously removed from the heap). Similarly to Dijkstra's algorithm the algorithm picks the node v with the smallest length of a path that is obtained by adding a single edge (z,v) to the constructed tree $T(w)$.

The algorithm checks if v is w-close (as z is the node z' with minimal $\hat{d}(w,z') + \omega(z',v)$ and $z' \in T(w)$, to check if v is w-close, the algorithm simply examines the distance $\hat{d}(w,z') + \omega(z',v)$ against all distances $\hat{d}(p_j(v),v))$. If v is w-close then the algorithm connects v to $T(w)$ by adding the edge (v,z).

In addition, the algorithm also maintains the following heaps (these heaps will be used for the efficiency of the update algorithm). For every j such that $0 \leq j < k-1$, and every vertex v, a heap $H_j(v)$ containing the following pairs: $(w,(z,v))$ for every edge (z,v) such that $w \in A_j$, $z \in T(w)$ and $v \notin T(w)$. The value of $(w,(z,v))$ in the heap is $\hat{d}(w,z) + \omega(z,v)$.

In addition, the algorithm maintains for every vertex v, an index $0 \leq j < k-1$ and vertex $w \in A_j$ such that $v \in T(w)$, a heap $H_j(v,w)$ containing the following. All edges of the form (u,v) such that $u \in T(w)$ with value $\hat{d}(w,u) + \omega(u,v)$. (this heap will be used to find a replacement edge for v in case the edge to its parent is deleted or in case the label of its parent increases).

This concludes the construction.

The query algorithm: Given are two nodes s and t. Find the minimal index i such that either $s \in T(p_i(t))$ or $t \in T(p_i(s))$. If $s \in T(p_i(t))$ then return $\hat{d}(p_i(t),s) + \hat{d}(p_i(t),t)$ otherwise return $\hat{d}(p_i(s),t) + \hat{d}(p_i(s),s)$. This concludes the query algorithm.

We next describe the update algorithm. We note that we need to slightly modify the update algorithm presented in the previous section. In the previous section all distances are shortest path distances in G, here this is not the case as we consider distances in the emulator and we do not necessarily have monotonicity in the distances (that is, some distances in the emulator can shrink over time).

The update algorithm - Delete an edge (x,y): The goal of the update algorithm is to make sure that by the end of the update algorithm there are no nodes w and v such that v is w-close and $v \notin T(w)$. Doing so efficiently makes the update algorithm to be technical.

We next describe the different steps in the update algorithm.

Remove the edge (x,y) from all relevant heaps. That is, delete the edge (x,y) from all heaps $H_i(y)$ and $H_i(x)$ it belongs to for some $0 \leq i < k-1$, namely, delete all pairs of

the form $(w,(x,y))$ from all $H_i(y)$ and all pairs of the form $(w,(y,x))$ from all $H_i(x)$ for $0 \leq i < k-1$. In addition, delete (x,y) from all heaps $H_j(y,w)$ and $H_j(x,w)$ for some $0 \leq j < k-1$ and $w \in A_j$, it belongs to.

In a couple of places in our update algorithm, we add and remove nodes from some tree $T(w)$. This requires updating the relevant heaps. Instead of repeating the same operations multiple times, we define here once how to update the heaps when a node joins or leaves a tree $T(w)$.

The following Procedure **Update-Heaps-Add**(w,v) is called when a node v is added to $T(w)$. Let $w \in A_i$ for some $0 \leq i < k-1$. Remove all keys $(w,(*,v))$ from the heap $H_i(v)$. For every neighbor u of v such that $u \notin T(w)$, update the key $(w,(v,u))$ in $H_i(u)$ with the value $\hat{d}(w,v)+\omega(v,u)$. If $u \in T(w)$, then add the key (v,u) to the heap $H_i(v,w)$ with value $\hat{d}(w,v)+\omega(v,u)$ and the key (u,v) to the heap $H_i(u,w)$ with value $\hat{d}(w,u)+\omega(v,u)$.

The following Procedure **Update-Heaps-Remove**(w,v) is called when a node v is removed from $T(w)$. Let i be the index such that $w \in A_i$. Iterate over all neighbors u of v such that $u \in T(w)$ and add $(w,(u,v))$ to $H_i(v)$ with value $\hat{d}(w,u)+\omega(u,v)$. Moreover, remove all instances $(w,(v,z))$ from $H_i(z)$ for some $z \notin T(w)$. For every neighbor u of v such that $u \in T(w)$ remove the edge (v,u) from $H_i(u,w)$.

The algorithm updates the trees $T(w)$ for $w \in A_i$ from $i = k-1$ to 0. For $i = k-1$, the algorithm simply deletes the edge from all **MES** trees $T(w)$ (which contain both x and y) for $w \in A_{k-1}$ by invoking the delete operation of **MES** (while maintaining the property that only nodes at distance at most $(1+2\epsilon)k \cdot d$ from w stay in $T(w)$).

Assume the algorithm updated all trees $T(w)$ for $w \in A_j$ for every $j > i$ and consider A_i. Note that for every $w \in A_i$ some nodes may need to be added to $T(w)$ (as their distance to their pivots on previous levels increased) or removed (as their distance to w increased). The algorithm fixes the trees of A_i as follows.

It maintains two heaps H^1 and H^2, both are initially set to be empty. Let $\hat{V}_i$ be the set of vertices v whose $\tilde{d}_i(v)$ increased as a result of the current deletion (recall that $\tilde{d}_i(v) = \min\{\hat{d}(v,p_j(v)) - \epsilon(j-i)d \mid j > i\}$).

Here we maintain two heaps rather than one, because it is more convenient to treat differently nodes whose $\tilde{d}_i(v)$ increased and nodes whose distance to w in $T(w)$ increased. This is done to avoid a situation in which we add a node v from $\hat{V}_i$ to a tree $T(w)$ but we add it to a subtree of the removed edge and only later discover that actually its new distance in $T(w)$ is larger than $\tilde{d}_i(v)$ and the node was not supposed to be added so we just wasted time on adding it to $T(w)$.

For every $v \in \hat{V}_i$ add to the heap H^1 the minimum of the heap $H_i(v)$ together with its value.

The heap H^2 contains initially the following. Initially for every $w \in A_i$ such that the deleted edge is in $T(w)$, it contains (w,v) for the node v that lost its edge to its parent in $T(w)$ with value $\hat{d}(w,v)$.

As long as H^1 or H^2 are not empty the algorithm extracts the minimum of the minimums of the two heaps. We treat the two cases differently.

If the minimum comes from the heap H^1: Let $(w,(u,v))$ be the minimum of H^1 with value $d_{min}(v)$. If v is w-close, that is if $d_{min}(v) < \tilde{d}_i(v)$ do the following: 1. connect v to $T(w)$ by adding the edge (u,v). 2. Set $\hat{d}(w,v) = \max\{\hat{d}_{last}(w,v), \hat{d}(w,u) + \omega(u,v)\}$. 3. Invoke Procedure **Update-Heaps-Add**(w,v). 4. Add the next minimum of $H_i(v)$ to H^1.

If the minimum is from H^2 do the following. Let (w,v) be the minimum in H^2 with value $d_{min}(v)$. Try to connect v to w with a single edge such that the total weight is $d_{min}(v)$. This is done by looking at the minimum (q,v) from $H_j(v,w)$ and checking if the value of this minimum is at most $d_{min}(v)$. If successful then do the following: 1. Connect v to $T(w)$ by adding the edge (q,v). 2. If $\hat{d}(w,v) < d_{min}(v)$ (this means that this is not the first time (w,v) is extracted from H^2 during the current update and thus its label is increased) then invoke Procedure **Update-Heaps-Add**(w,v).

Otherwise (if v cannot be connected to $T(w)$ with distance $d_{min}(v)$) do the following: 1. if $d_{min}(v) + \epsilon_2 d_\mu < \tilde{d}_i(v)$ then update the value of v in H^2 to be $d_{min}(v) + \epsilon_2 d_\mu$. Else ($d_{min}(v) + \epsilon_2 d_\mu \geq \tilde{d}_i(v)$) remove v from the heap. 2. Update $\hat{d}_{last}(w,v) = d_{min}(v) + \epsilon_2 d_\mu$. 3. Invoke Procedure **Update-Heaps-Remove**(w,v).

This concludes the update algorithm.

Note that if v was added to H^2 because the label of its parent increases or because the edge to its parent is deleted, and v is reconnected to the tree $T(w)$ without increasing its label then the algorithm does not invoke Procedures **Update-Heaps-Add** and **Update-Heaps-Remove**, that is, the time spent on v is $O(\log n)$ (to extract the minimum from H^2). This is similar to the **ES** algorithm where we pay $O(\deg(v))$ only if the distance to v from the root increases. Note also that when a distance in G_μ increases, it must increase by at least $\epsilon_2 d_\mu$, as all weights in G_μ are multiples of $\epsilon_2 d_\mu$.

Similarly to the previous section, here we also have the property that there is no node v that is w-close and $v \notin T(w)$.

Claim 4.2: After the update algorithm, there is no node v that is w-close and $v \notin T(w)$ for some $w \in A_i$ and $0 \leq i < k-1$.

We next bound the stretch of the algorithm. We rely on the following Lemma that is proved in the full version.

Lemma 4.3: Maintaining **MES** on a $(\tilde{d}, \epsilon_1, \epsilon_2)$-emulator E^* from s has the following properties. For every $w \in Z_i$: $\hat{d}(s, w, E^*) \leq (1+\epsilon_1)(1+6\epsilon_2)\mathbf{dist}(s, w)+\tilde{\beta}_2-8\tilde{\ell}_{i+1}$. Where $\tilde{\ell}_i = \tilde{d}(6/\epsilon_2)^{i+1}$ and $\tilde{\beta}_2 = 8\tilde{\ell}_{t-2} + \tilde{d}$, and $\hat{d}(s, w, E^*)$ is the distance in the **MES** tree from s invoked on the graph E^*.

It is important to note that we use Lemma 4.3 only in the base case, that is, to maintain the trees of A_{k-1}. To maintain the trees of A_i for $i < k-1$ we use a modification that allows us to maintain approximate trees in which nodes might be added and removed over the course of the algorithm.

In our case our emulator is $(d_\mu, (1 + \epsilon_2)^\mu, \epsilon_2)$-emulator. We have $(1 + \epsilon_2)^\mu(1 + \epsilon_2) < 1 + \epsilon/(10 \cdot k)$. For every $w \in Z_i$, let $\hat{\delta}(s, w) = (1 + \epsilon/(10k))\mathbf{dist}(s, w) + \beta_2 - 8\ell_{i-1}$ or $\hat{\delta}(s, w) = 0$ in case $w = s$. (recall that $\ell_i = d_\mu(6/\epsilon_2)^{i+1}$ and $\beta_2 = 8\ell_{t-2} + d_\mu$.)

The next lemma shows that every node v "close" enough to w for some $w \in A_i$ is either in $T(w)$ or has a node $z \in A_j$ for $j > i$ such that $v \in T(z)$ and the distance $\hat{d}(z, v)$ is "small" enough. The proof of this lemma is quite technical and relies heavily on the properties of the emulator.

Lemma 4.4: Consider nodes $v \in V$ and $w \in A_i$ for some $0 \leq i < k-1$ such that $\hat{\delta}(w, v) < (1 + 2\epsilon)(i + 1)d$ then 1. either $v \in T(w)$ and $\hat{d}(w, v) \leq \hat{\delta}(w, v)$ or
2. $v \notin T(w)$ and there exists a node $z \in A_j$ for $j > i$ such that $v \in T(z)$, $\hat{d}(v, z) \leq \hat{\delta}(w, v) + \epsilon(j - i)d$.

In addition, for every v and $w \in A_i$ for some $0 \leq i \leq k-1$ we have, $\hat{d}_{last}(w, v) \leq \hat{\delta}(w, v)$.

Proof:

First, we note that the claim initially (before any update) holds. The proof is similar to the analysis in the previous section as **MES** and **ES** operate the same before any update. We therefore can show that initially we have $\hat{d}(w, v) = \mathbf{dist}(w, v)$ for $v \in T(w)$ and the analysis is similar to the previous section.

Assume the claim holds up until the last update and consider a deletion of an edge e. The proof is by induction on i from $k-1$ to 0. For $i = k-1$, the claim follows by Lemma 4.3.

Assume correctness for $j > i$ and consider $w \in A_i$. We use again induction to prove correctness for every $v \in V$. The induction is on $\hat{\delta}(w, v)$.

For $\hat{\delta}(w, v) = 0$, that is $v = w$ the claim is clear . Assume correctness for every v' such that $\hat{\delta}(w, v') < \hat{\delta}(w, v)$ and consider v such that $\hat{\delta}(w, v) < (1 + 2\epsilon)(i + 1)d$. Let i' be the index such that $v \in Z_{i'}$. Let z' be the node that covers v. We consider two cases: (a) $v \neq z'$ and (b) $v = z'$.

Assume the first case (a), that is, $v \neq z'$. Note that, $z' \in Z_{j'}$ for some $j' > i'$. Note also that, $\mathbf{dist}(v, z') \leq \omega(v, z', G_\mu) \leq 3d_\mu(6/\epsilon_2))^{j'} = 3\ell_{j'-1}$. We get,

$$
\begin{aligned}
& \hat{\delta}(w, z') = \\
& (1 + \epsilon/(10k))\mathbf{dist}(w, z') + \beta_2 - 8\ell_{j'-1} \\
\leq\ & (1 + \epsilon/(10k))(\mathbf{dist}(w, v) + \mathbf{dist}(v, z')) + \beta_2 - 8\ell_{j'-1} \\
\leq\ & (1 + \epsilon/(10k))(\mathbf{dist}(w, v) + 3\ell_{j'-1}) + \beta_2 - 8\ell_{j'-1} \\
<\ & (1 + \epsilon/(10k))\mathbf{dist}(w, v) + 6\ell_{j'-1} + \beta_2 - 8\ell_{j'-1} \\
\leq\ & (1 + \epsilon/(10k))\mathbf{dist}(w, v) + \beta_2 - 2\ell_{j'-1} \\
\leq\ & (1 + \epsilon/(10k))\mathbf{dist}(w, v) + \beta_2 - 8\ell_{i'-1} \\
=\ & \hat{\delta}(w, v),
\end{aligned}
$$

where the forth inequality follows for every $(1 + \epsilon/(10k)) < 2$ and the last inequality follows for every small enough ϵ_2 ($\epsilon_2 < 4$).

We get that $\hat{\delta}(w, z') < \hat{\delta}(w, v)$, we can therefore use induction hypothesis on z'. By induction hypothesis on z', we have that either (1) $z' \in T(w)$ and $\hat{d}(w, z') \leq \hat{\delta}(w, z')$ or (2) $z' \notin T(w)$ and there exists a node $z \in A_j$ for $j > i$ such that $z' \in T(z)$, $\hat{d}(z, z') \leq \hat{\delta}(w, z') + \epsilon(j - i)d$.

Assume first that (1) happens, that is, $z' \in T(w)$ and $\hat{d}(w, z') \leq \hat{\delta}(u, z')$. By straight forward calculation we can show that $\hat{d}(w, z') + \omega(z', v) \leq \hat{\delta}(w, z') + \omega(z', v) \leq \hat{\delta}(w, v)$.

In addition, before iteration i of the update algorithm, we have by induction hypothesis $\hat{d}_{last}(w, v) \leq \hat{\delta}(w, v)$. We need to show that if the algorithm updates $\hat{d}_{last}(w, v)$, then we still have $\hat{d}_{last}(w, v) \leq \hat{\delta}(w, v)$.

By construction, the algorithm increases $\hat{d}_{last}(w, v)$ only if the following two conditions hold: 1. $\hat{d}_{last}(w, v) < \tilde{d}_i(v)$. 2. There is no node u that is already in $T(w)$ at the time $\hat{d}_{last}(w, v)$ is increased such that $\hat{d}(w, u) + \omega(u, v) \leq \hat{d}_{last}(w, v)$.

Both conditions are straight forward by construction. Note that if the algorithm increases $\hat{d}_{last}(w, v)$ to $\hat{\delta}(w, v)$ then at that time z' was already connected to $T(w)$. To see this, recall that we take the minimum over both heaps H^1 and H^2 and as $\hat{d}(w, z') < \hat{\delta}(w, v)$, the value $\hat{d}(w, z')$ should have been considered before the value $\hat{\delta}(w, v)$. Hence, by the second condition the algorithm does not increase $\hat{d}_{last}(w, v)$ to be more than $\hat{\delta}(w, v)$.

If (2) happens, that is, $z' \notin T(w)$ and there exists a node $z \in A_j$ for $j > i$ such that $z' \in T(z)$, $\hat{d}(z, z') \leq \hat{\delta}(w, z') + \epsilon(j-i)d$, then we have the following by again straight forward

calculation.

$$\begin{aligned}
& \hat{d}(z,z') + \omega(z',v) \leq \hat{\delta}(w,z') + \epsilon(j-i)d + \omega(z',v) \\
\leq\ & (1+\epsilon/(10k))\mathbf{dist}(w,z') + \beta_2 - 8\ell_{j'-1} + \epsilon(j-i)d + \\
& \omega(z',v) \\
\leq\ & (1+\epsilon/(10k))(\mathbf{dist}(w,v) + \mathbf{dist}(v,z')) + \beta_2 - 8\ell_{j'-1} + \\
& \epsilon(j-i)d + \omega(z',v) \\
\leq\ & (1+\epsilon/(10k))(\mathbf{dist}(w,v) + 3\ell_{j'-1}) + \beta_2 - 8\ell_{j'-1} + \\
& \epsilon(j-i)d + 3\ell_{j'-1} \\
<\ & (1+\epsilon/(10k))\mathbf{dist}(w,v) + 4\ell_{j'-1} + \beta_2 - 8\ell_{j'-1} + \\
& \epsilon(j-i)d + 3\ell_{j'-1} \\
=\ & (1+\epsilon/(10k))\mathbf{dist}(w,v) + \beta_2 - \ell_{j'-1} + \epsilon(j-i)d \\
<\ & (1+\epsilon/(10k))\mathbf{dist}(w,v) + \beta_2 - 8\ell_{i'-1} + \epsilon(j-i)d \\
=\ & \hat{\delta}(w,v) + \epsilon(j-i)d
\end{aligned}$$

Hence, by construction either $v \in T(z)$ and the claim follows or there exists $y \in A_r$ for $r > j$ such that $v \in T(y)$ and $\hat{d}(y,v) \leq \hat{\delta}(w,v) + \epsilon(r-i)d$ and again the claim follows. Notice that in any case we have $\tilde{d}_i(v) \leq \hat{\delta}(w,v)$. Hence, the algorithm will not increase $\hat{d}_{last}(w,v)$ above $\hat{\delta}(w,v)$ and in addition, if $v \in T(w)$ then $\hat{d}(w,v) \leq \hat{\delta}(w,v)$.

Finally, we are left with case (b), that is, where $v = z'$, that is v covers v.

Recall that G_μ has edges from v to all nodes at distance at most $\lceil d_\mu(6/\epsilon_2)^{i'+1}/2 \rceil$ from it.

Let $P(v,w)$ be the shortest path from v to w in G. If v and w are connected in G_μ by a single edge, then the claim follows by construction.

Otherwise, let q be the node at distance $\lceil d_\mu(6/\epsilon_2)^{i'+1}/2 \rceil$ from v in $P(v,w)$.

By induction hypothesis we can show the claim is true for q. Using similar arguments as above we can also show that the claim is true for v. ∎

The next lemma bounds the stretch.

Lemma 4.5: Consider nodes s and t. If $\mathbf{dist}(s,t) \leq d$ then the distance returned by the query algorithm is at most $(2(1+2\epsilon)k-1)d$.

Proof: Consider two nodes s and t whose distance is at most d. To prove the claim, notice that we only need to prove that there is an index i such that either 1. $s \in T(p_i(t))$ and $\hat{\delta}(t,p_i(t)) \leq i \cdot d + 2i\epsilon d$ or 2. $t \in T(p_i(s))$ and $\hat{\delta}(s,p_i(s)) \leq i \cdot d + 2i\epsilon d$. To see this, recall that the depth of all trees is at most $(1+2\epsilon)kd$ and that $i \leq k-1$.

Consider the maximal index i such that either $\hat{\delta}(s,p_i(s)) \leq i \cdot d + 2i\epsilon d$ or $\hat{\delta}(t,p_i(t)) \leq i \cdot d + 2i\epsilon d$. (note that there is such an index as $\hat{\delta}(s,p_0(s)) = \hat{\delta}(s,s) = 0$ as all nodes belong to A_0). Assume w.l.o.g. that for this index i, we have $\hat{\delta}(s,p_i(s)) \leq i \cdot d + 2i\epsilon d$ (rather than $\hat{\delta}(t,p_i(t)) \leq i \cdot d + 2i\epsilon d$).

We claim that $t \in T(p_i(s))$, which implies the lemma. Note that

$$\begin{aligned}
& \hat{\delta}(t,p_i(s)) = (1+\epsilon/(10k))\mathbf{dist}(t,p_i(s)) + \beta_2 - 8\tilde{\ell}_{i+1} \\
\leq\ & (1+\epsilon/(10k))(\mathbf{dist}(s,p_i(s)) + \mathbf{dist}(s,t)) + \beta_2 - 8\tilde{\ell}_{i+1} \\
\leq\ & \hat{\delta}(s,p_i(s)) + (1+\epsilon/(10k))d \\
\leq\ & i \cdot d + 2i\epsilon d + (1+\epsilon/(10k))d \\
\leq\ & (1+2\epsilon)(i+1) \cdot d.
\end{aligned}$$

By Lemma 4.4 we have either (1) $t \in T(p_i(s))$ or (2) $t \notin T(p_i(s))$ and there exists a node $z \in A_j$ for $j > i$ such that $t \in T(z)$, $\hat{d}(t,z) \leq \hat{\delta}(t,p_i(s)) + \epsilon(j-i)d$. However, we claim that (2) contradicts the maximality of i. To see this, note that if $t \in T(z)$ we have $\hat{d}(t,z) \leq \hat{\delta}(t,p_i(s)) + \epsilon(j-i)d \leq i \cdot d + 2i\epsilon d + (1+\epsilon/(10k))d + \epsilon(j-i)d = (i+1) \cdot d + j\epsilon d + i\epsilon d + \epsilon/(10k)d$.

Note that, $\hat{d}(t,z) \geq \mathbf{dist}(t,z)$. Hence, we get $\hat{\delta}(t,p_j(t)) \leq (1+\epsilon/(10k))((i+1) \cdot d + j\epsilon d + i\epsilon d + \epsilon/(10k)d) + \beta_2 \leq j \cdot d + 2j\epsilon d$, with contradiction again to the maximality of i. Hence, $t \in T(p_i(s))$. ∎

The next auxiliary claim will help us bound the number of trees each node belongs to.

Claim 4.6: Consider a node $v \in V$. Let $\mathbf{dist}(v,A_{i+1}) = d'$. For every $w' \in A_i$ such that $\mathbf{dist}(v,w') \geq d' - \epsilon d/2$ we have $v \notin T(w')$.

Lemma 4.7: Consider a node $v \in V$. During the entire running time of the algorithm, the total number of distinct trees that v ever joined to is $O(k^2 \cdot \log n \cdot n^{1/k}/\epsilon)$ in expectation.

The next lemma bounds the total running time of the algorithm.

Lemma 4.8: The expected total running time of the algorithm is $\tilde{O}(mn^{1+1/k+o(1)})$.

REFERENCES

[1] Ittai Abraham, Shiri Chechik, and Kunal Talwar. Fully dynamic all-pairs shortest paths: Breaking the $o(n)$ barrier. In *International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 1–16, 2014.

[2] Surender Baswana, Akshay Gaur, Sandeep Sen, and Jayant Upadhyay. Distance oracles for unweighted graphs: Breaking the quadratic barrier with constant additive error. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008*, pages 609–621, 2008.

[3] Surender Baswana and Telikepalli Kavitha. Faster algorithms for approximate distance oracles and all-pairs small stretch paths. In *Proceedings of the 47th Annual Symposium on Foundations of Computer Science*, FOCS, pages 591–602, 2006.

[4] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Transactions on Algorithms*, 8(4):35, 2012.

[5] Surender Baswana, Utkarsh Lath, and Anuradha S. Mehta. Single source distance oracle for planar digraphs avoiding a failed node or link. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 223–232, 2012.

[6] Aaron Bernstein. Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, pages 693–702, 2009.

[7] Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 725–734, 2013.

[8] Aaron Bernstein. Deterministic Partially Dynamic Single Source Shortest Paths in Weighted Graphs. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80, pages 44:1–44:14, 2017.

[9] Aaron Bernstein and Shiri Chechik. Deterministic decremental single source shortest paths: Beyond the o(mn) bound. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 389–397. ACM, 2016.

[10] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1355–1365, 2011.

[11] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1355–1365, 2011.

[12] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

[13] Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In *Theoretical Computer Science, Essays in Memory of Shimon Even*, pages 218–240, 2006.

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

[15] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS, pages 664–, 1995.

[16] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. In *Proceedings of the 54th Annual Symposium on Foundations of Computer Science*, FOCS, pages 538–547, 2013.

[17] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, FOCS, pages 146–155, 2014.

[18] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Sublinear-time decremental algorithms for single-source reachability and shortest paths on directed graphs. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 674–683, 2014.

[19] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A subquadratic-time algorithm for decremental single-source shortest paths. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 1053–1072, 2014.

[20] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In *Proceedings of the 42nd International Colloquium, ICALP*, pages 725–736, 2015.

[21] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 21–30, 2015.

[22] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS, pages 81–91, 1999.

[23] Valerie King and Mikkel Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *COCOON*, pages 268–277, 2001.

[24] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

[25] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. *SIAM J. Comput.*, 41(3):670–683, 2012.

[26] Piotr Sankowski. Subquadratic algorithm for dynamic shortest distances. In *International Computing and Combinatorics Conference (COCOON)*, pages 461–470, 2005.

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

[28] Mikkel Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 112–119, 2005.

[29] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[30] Christian Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 202–208, 2012.