Deterministic Decremental Single Source Shortest Paths: Beyond the O(mn) Bound

Aaron Bernstein
Columbia University
New York, USA
bernstei@gmail.com

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

ABSTRACT

In this paper we consider the *decremental* single-source shortest paths (SSSP) problem, where given a graph G and a source node s the goal is to maintain shortest paths between s and all other nodes in G under a sequence of online adversarial edge deletions.

In their seminal work, Even and Shiloach [JACM 1981] presented an exact solution to the problem with only $O(mn)$ total update time over all edge deletions. Their classic algorithm was the best known result for the decremental SSSP problem for three decades, even when approximate shortest paths are allowed.

The first improvement over the Even-Shiloach algorithm was given by Bernstein and Roditty [SODA 2011], who for the case of an unweighted and undirected graph presented an *approximate* $(1+\epsilon)$ algorithm with constant query time and a total update time of $O(n^{2+O(1/\sqrt{\log n})})$. This work triggered a series of new results, culminating in a recent breakthrough of Henzinger, Krinninger and Nanongkai [FOCS 14], who presented a $(1+\epsilon)$-approximate algorithm whose total update time is near linear $O(m^{1+O(1/\sqrt{\log n})})$. In this paper they posed as a major open problem the question of derandomizing their result.

In fact, all known improvements over the Even-Shiloach algorithm are randomized. All these algorithms maintain some truncated shortest path trees from a small subset of nodes. While in the randomized setting it is possible to "hide" these nodes from the adversary, in the deterministic setting this is impossible: the adversary can delete all edges touching these nodes, thus forcing the algorithm to choose a new set of nodes and incur a new computation of shortest paths.

In this paper we present the first *deterministic* decremental SSSP algorithm that breaks the Even-Shiloach bound of $O(mn)$ total update time, for unweighted and undirected graphs. Our algorithm is $(1+\epsilon)$ approximate and achieves a total update time of $\tilde{O}(n^2)$. Our algorithm can also achieve the same bounds in the *incremental* setting. It is worth mentioning that for dense instances where $m = \Omega(n^{2-1/\sqrt{\log(n)}})$, our algorithm is also faster than all existing randomized algorithms.

*This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 1528/15).

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph Algorithms

Keywords

Shortest Paths, Dynamic Algorithms, Approximation Algorithms

General Terms

Algorithms, Theory

1. INTRODUCTION

The objective of dynamic graph algorithms is to handle an online sequence of update operations while maintaining a desirable functionality on the graph, e.g., the ability to answer shortest path queries. An update operation may involve a deletion or insertion of an edge or a node, or a change in an edge's weight. In case the algorithm can handle only deletions, it is called decremental, if it can handle only insertions it is called incremental, and if it can handle both it is called fully dynamic.

Computing shortest paths in a graph is one of the fundamental problems of graph algorithms, and has a wide variety of applications. Fully dynamic shortest paths has a very clear motivation, as many shortest path applications must deal with a graph that is changing over time. The incremental setting is somewhat more restricted, but is applicable to any setting in which the network is only expanding (e.g. in road networks some new roads may be constructed but it is very rare that roads are demolished; in social networks an edge may indicate that the two endpoint users know each other or communicated in the past and thus connections may only be added over time).

The decremental setting is often very important from a theoretical perspective, as decremental shortest paths (and decremental *single source* shortest paths especially) are used as a building block in a large variety of fully dynamic shortest paths algorithms; see e.g. [26, 9, 2, 3]. In addition, in many cases, a decremental shortest paths algorithm can be tweaked to give the same bounds in the incremental setting (this is also the case for our algorithm). Decremental shortest paths can also have applications to *static* graphs problems; see e.g. Madry's paper on efficiently computing multicommodity flows [32].

In this paper we consider the problem of (approximate) single source shortest paths (SSSP) in unweighted undirected

graphs, in the decremental setting. Specifically, given an unweighted undirected graph G and a source node s, our algorithm needs to preform the following two operations:

- Delete(e) – delete the edge e from the graph.
- Distance(v) – return the distance between s and v, i.e., $\mathbf{dist}(s, v)$, in the current graph G.

As we allow our algorithm to return approximate solutions, we say that an algorithm has an approximation guarantee of α (or equivalently stretch α) if its output on the query Distance(v) is never smaller than the actual shortest distance and is not more than α times the shortest distance. Formally, if we denote by $\widehat{\mathbf{dist}}(s, v)$ the output of the algorithm on the query Distance(v), we say that the algorithm has stretch α if after any adversarial sequence of deletions Delete(e_1), ..., Delete(e_ℓ), the following holds for any query Distance(v):

$$\mathbf{dist}(s, v) \leq \widehat{\mathbf{dist}}(s, v) \leq \alpha \cdot \mathbf{dist}(s, v) \ .$$

The goal in designing dynamic shortest path algorithms is twofold. The first is to minimize the time it takes the algorithm to adapt to an update, i.e., the Delete operation, and the second is to minimize the query time, i.e., the Distance operation. Typically one tries to keep the query time small (polylog or constant), while getting the update time as low as possible. In the decremental setting, which is the focus of this paper, one usually considers the aggregate sum of update times over the *entire* sequence of deletions, which is referred to as the *total update time*.

1.1 Related work

The most naive solution to dynamic SSSP is to simply invoke a static SSSP algorithm after every deletion. The classic Dijkstra's algorithm takes $\tilde{O}(m)$ [1] [2] time. There have been some improvements over the log factors (see e.g. [39, 40, 36]), but all algorithms for this problem require at minimum $O(m)$ time. Since there can be a total of m deletions, the total update time for the naive implementation is at least $O(m^2)$.

In 1981, Even and Shiloach [18] were the first to present a decremental SSSP algorithm with total update time better than the naive solution. More specifically, they presented an algorithm for undirected, unweighted graphs with constant query time and a total update time of $O(mn)$. A similar result was independently found by Dinitz [17]. This was later generalized to directed graphs by King [29]. The naive implementation of [29] requires in the worst case $O(n^3)$ memory. King and Thorup [30] later implemented a dynamic algorithm with only $O(n^{2.5})$ memory ($O(n^2\sqrt{nb})$ memory, where b is the maximal edge weight).

Dynamic shortest path algorithms and other related problems were extensively studied in the last three decades and many attempts were made to improve the classic Even and Shiloach [18] algorithm. However, no progress was made in the exact decremental SSSP problem.

Roditty and Zwick [37] presented an explanation for this lack of progress, by showing a reduction from boolean matrix multiplication to both the incremental and the decremental SSSP problem in unweighted undirected graphs. This implies that unless a major breakthrough in combinatorial boolean matrix multiplication is provided, no combinatorial algorithm for the problem can go beyond the $O(mn)$ total update time of Even and Shiloach (except perhaps by log factors). Very recently, Henzinger *et al* [25] proved the same result for *any* type of algorithm (not just a combinatorial one), assuming their online boolean matrix-vector multiplication conjecture.

These lower bounds motivated the study of the approximate version of this problem. There has been much progress in the case of *randomized* approximate solutions. Bernstein and Roditty [11] were the first to give an approximate decremental SSSP algorithm to go beyond the $O(mn)$ total update time of Even and Shiloach [18]: they presented a $(1+\epsilon)$ decremental SSSP algorithm for undirected unweighted graphs with constant query time and $O(n^{2+O(1/\sqrt{\log n})}) = O(n^{2+o(1)})$ total update time. Henzinger, Krinninger and Nanongkai [23] later improved the total update time for unweighted undirected graphs to $O(n^{1.8+o(1)} + m^{1+o(1)})$.

For *directed* weighted graphs, Henzinger, Krinninger and Nanongkai [22] presented a $(1+\epsilon)$ approximate decremental SSSP algorithm with total update time $O(mn^{0.984+o(1)} \log W)$, where W is the largest weight in the graph. Afterwards they improved the running time to $O(mn^{0.9+o(1)} \log W)$ [24].

Returning to undirected graphs, in a recent breakthrough Henzinger, Krinninger and Nanongkai [21] presented a $(1+\epsilon)$ decremental SSSP algorithm with near linear total update time: in particular, $O(m^{1+o(1)} \log W)$. This is close to optimal, as $O(m)$ is an obvious lower bound. They posed as a major open problem the derandomization of their result. In fact all the above improvements over the Even-Shiloach [18] algorithm are randomized. The challenge of derandomizing the above algorithms is discussed in Section 1.2.

There is also an extensive literature on the dynamic *all pairs* shortest paths problem, which covers various variants and aspects of this problem, including exact solutions [4, 29, 26, 15, 16, 41, 42, 6], approximate solutions [5, 38, 11, 10, 20, 1, 23, 21, 22, 24], fully dynamic algorithms [29, 38, 16, 38, 9, 8, 3], partially dynamic algorithms (i.e., only decremental or incremental) [6, 5, 38, 11, 23], and special families of graphs [31, 27, 19, 2]. It is worth noting that dynamic SSSP algorithms are a key ingredient in many of the all-pairs results mentioned above.

1.2 Randomization in Dynamic Algorithms

The large majority of the dynamic shortest path algorithms mentioned above are randomized, and for most dynamic shortest path problems there is a polynomial gap between the best randomized and deterministic algorithm. Bridging the this randomization gap is especially important in the dynamic setting because randomization almost always assumes a weaker adversary as well. In particular, all of the randomized algorithms above assume an *oblivious* adversary that does not have access to the random choices made by the users. Most of them also require the extra assumption of a *non-adaptive* adversary whose updates are fixed in advance and cannot be influenced by the queries returned by the user. These two extra assumptions (especially the latter) render randomized algorithms entirely unusable in certain settings.

Unfortunately, the very thing that makes randomized algorithms problematic in the dynamic setting is also what gives them their power. Consider, for illustration, a simple clustering problem where the goal is to maintain a set of $\backsim \sqrt{n}$ vertices called centers in an unweighted undirected graph, such that all other vertices are at distance at most $\sqrt{n}$ from one of the

[1] The $\tilde{O}$ notation suppresses polylogarithmic factors.

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

centers. (We assume for simplicity that the graph is always connected). This basic clustering tool is used in a huge number of approximate shortest path algorithms, both static and dynamic [33, 35, 34, 7, 43, 44, 12, 13, 14, 38, 11, 23, 21]. In the static setting, there is an obvious randomized algorithm: sample $O(\sqrt{n}\log(n))$ centers uniformly at random. A simple greedy deterministic algorithm also exists in the static setting, but in the dynamic setting, it is easy to see that an efficient deterministic algorithm cannot exist: whatever $\sqrt{n}$ centers we choose, the adversary can disconnect them while leaving the rest of graph intact, forcing us to restart from scratch. With randomization and an oblivious adversary, however, the problem becomes easy: once again we sample $O(\sqrt{n}\log(n))$ centers uniformly at random. The adversary then proceeds to change the graph, but since the updates are oblivious to our random choices we can argue that these centers are uniformly random in *all* versions of the graph, and so with high probability will form a valid clustering throughout the entire update sequence. The extra assumption of a non-adaptive adversary is often necessary to prevent the adversary from gaining information about our randomly chosen centers from our answers to whatever queries are supported by the algorithm.

Essentially every randomized algorithms for dynamic shortest paths (all pairs or single source) uses some generalization of the above clustering, and so is difficult to match with a deterministic algorithm. As far as we know the only exception is the decremental $(1+\epsilon)$-approximate all pairs shortest path algorithm with $O(mn)$ total update time of Henzinger *et al.* (for unweighted undirected graphs) [20], which is a derandomization of the clustering-based algorithm of Roditty and Zwick [38] that achieves the same bounds. For decremental SSSP in particular, all the approximation algorithms that break through the $O(mn)$ barrier rely on clustering using randomization, and all require an oblivious non-adaptive adversary. We develop a different approach that does not rely on any clustering scheme, and is the first to break through the $O(mn)$ barrier deterministically.

1.3 Our Results

Theorem 1.1 *Given an undirected unweighted graph G subject to a sequence of edge deletions, and a fixed source s, there exists a* deterministic *algorithm that maintains $(1+\epsilon)$ approximate distances from s to every vertex in total update time $O(m\log^3(n) + n^2\log(n)\epsilon^{-1})$. The query time is $O(1)$.*

We can easily extend our algorithm to work for graphs with small positive integer weights, at the cost of multiplying the update time $O(W)$. We can also extend our algorithm to work in the *incremental* setting, where we start with an empty graph, and the adversary inserts edges one at a time. In Section 6 we explain the modifications needed to make our algorithm work in the incremental case.

Finally, we can extend the algorithm to answer approximate shortest *path* queries. We leave this extensions for the full version of the paper.

Our algorithm does not match the randomized state of the art of $O(m^{1+o(1)})$, but it is optimal up to log factors for dense graphs, and is the first deterministic algorithm to go beyond the $O(mn)$ barrier. In addition to being deterministic, our algorithm has several secondary advantages. The first is that is much simpler than the three randomized algorithms for the problem [11, 23, 21]. The reason for this is that those algorithms all relied on variations of the same clustering technique of Thorup and Zwick [44] which is somewhat involved, especially in the dynamic setting. We develop an entirely different approach which is much simpler, and could potentially be of use in other deterministic algorithms for dynamic shortest paths. The simplicity also allows us to avoid the extra $m^{O(1/\sqrt{\log(n)})} = m^{o(1)}$ term incurred by all the randomized algorithms, which again arose from the Thorup and Zwick clustering. Thus, in addition to being deterministic, our algorithm is simpler and faster than the $O(n^{2+o(1)})$ of Bernstein and Roditty, and is in fact faster than *all* existing randomized algorithms for the problem in dense graphs where $m = \Omega(n^{2-1/\sqrt{\log(n)}})$.

2. PRELIMINARIES

In our model, edges are being deleted one by one from an unweighted undirected graph. Let $G = (V, E)$ always refer to the *current* version of the graph. Let m refer to the number of edges in the original graph, and n to the number of vertices. For any pair of vertices u, v, let $\pi(u, v)$ be the shortest $u - v$ path in G (ties can be broken arbitrarily), and let $\mathbf{dist}(u, v)$ be the length of $\pi(u, v)$. Let s be the fixed source from which our algorithm must maintain approximate distances, and let ϵ refer to our approximation parameter; when the adversary queries the distance to a vertex v, the algorithm must return an approximate distance $\widehat{\mathbf{dist}}(v)$ such that $\mathbf{dist}(s,t) \leq \widehat{\mathbf{dist}}(s,t) \leq (1+\epsilon)\mathbf{dist}(s,t)$.

Our algorithm will make use of several graphs that are different from G. Given any subset of vertices $V' \subseteq V$, we define the induced graph $G[V']$ to contain all the vertices V', and all edges $(u, v) \in E$ such that u and v are both in V'. Given any graph H, and any two vertices u, v in H, we define $\pi_H(s,t)$ to be the shortest $u-v$ path in H, and we define $\mathbf{dist}_H(s,t)$ to be the length of $\pi_H(s,t)$. Given any two sets S, T we define the set difference $S \setminus T$ to contain all elements s such that $s \in S$ but $s \notin T$.

We will measure the update time of the dynamic subroutines used by our algorithm in terms of their *total* update time over the entire sequence of edge changes. Note that although edges in the main graph G are only being deleted, there may be edge insertions into the auxiliary graphs used by the algorithm.

Definition 2.1 *Given a graph G subject to a sequence of edge deletions and insertions, define* MAX-EDGES(G) *to be the number of pairs (u, v) such that edge (u, v) is in the graph at* some *point during the update sequence. Note that if the update sequence contains only deletions, then* MAX-EDGES(G) *is simply the number of edges in the original graph.*

The basic building block of almost every decremental shortest path algorithms, including ours, is an algorithm of Even and Shiloach from 1981 that maintains a shortest path tree up to some depth d.

Definition 2.2 *Given any number d, the function* BOUND$_d(x)$ *is equal to x if $x \leq d$, and to ∞ otherwise.*

Definition 2.3 *Let G be a dynamic graph subject to a sequence of edge insertions and deletions, let s be a fixed source, and let d be some depth bound. Then, the algorithm $\mathbf{ES}(G, s, d)$ maintains the value* BOUND$_d(\mathbf{dist}(s, v))$ *for every vertex v over*

the entire sequence of changes to G. We refer to this as running an Even and Shiloach tree in G from source s up to depth d.

Lemma 2.4 *[18] Let $G = (V, E)$ be a dynamic graph with positive integer weights, let s be a fixed source, and say that for every vertex v we are guaranteed that the distance $\mathbf{dist}(s, v)$ never decreases due to an edge insertion. Then, the* total *update time of $\mathbf{ES}(G, s, d)$ over the entire sequence of edge updates is $O(m \cdot d + \Delta)$, where $m = \text{MAX-EDGES}(G)$, and Δ is the total number of edge changes.*

Remark 2.5 *The typical guarantee given for the ES-tree is that $\mathbf{ES}(G, s, d)$ has total update time $O(md)$ as long as the graph is subject to* only *deletions. But as pointed out in [11], the only-deletions assumption is only necessary to guarantee that distances do not decrease, so the same bound holds as long as we can separately guarantee this monotonicity for the insertions as well. The existence of insertions leads to the extra $O(\Delta)$ term because the algorithm needs to spend $O(1)$ time per edge change. (In the case of only deletions, we have $\Delta \leq m$, so we can ignore the Δ term).*

Corollary 2.6 *If a dynamic graph G and a source s satisfy all the assumptions of Lemma 2.4 except that weights in G are not necessarily integral, but all the weights are positive and integer multiples of some number x, then the* total *running time of $\mathbf{ES}(G, s, d)$ is $O(md/x + \Delta)$, where $m = \text{MAX-EDGES}(G)$. (We simply divide all weights by x and apply Lemma 2.4.)*

3. HIGH LEVEL OVERVIEW

To highlight the simplicity of our approach, and to make the technical details in the body of the paper easier to follow, we start with a high level description of how to maintain $(1 + \epsilon)$-approximate distances from a source s in total update time $\tilde{O}(n^{2.5})$, ignoring log factors that arise from some of the technical details. We assume for this section that ϵ is a fixed constant. Note that the Even and Shiloach tree of Lemma 2.4 already provides a method for maintaining *short* distances. In particular, running $ES(G, s, 5\sqrt{n}\epsilon^{-1})$ only requires $O(m\sqrt{n}) = O(n^{2.5})$ time and maintains all distances $\mathbf{dist}(s, v)$ for which $\mathbf{dist}(s, v) \leq 5\sqrt{n}\epsilon^{-1}$.

To maintain *long* distances, we will sparsify the graph G. Let us say that a vertex $v \in V$ is *heavy* if it has degree at least $\sqrt{n}$, and *light* otherwise. Our main observation is that any shortest path π in G can contain at most $3\sqrt{n}$ heavy vertices: intuitively, this is because no two heavy vertices on π can share a common neighbor because then there would be a very short path between them, which we could use to obtain a path shorter than the shortest path π. (The formal proof is only slightly complicated by the fact that two heavy vertices on the path *can* share a common neighbor if the two heavy vertices are already right next to each other on the shortest path π; however if they are at distance 3 away from each other in π, they cannot have a common neighbor.) Thus, since the neighborhood of a single heavy vertex contains $\sqrt{n}$ vertices, and there are only n vertices in total, there can only be $\sqrt{n}$ heavy vertices on the shortest path π.

Since we are only concerned with vertices v for which $\mathbf{dist}(s, v) >> \sqrt{n}$, we know that the shortest path from s to v will contain far more light vertices than heavy vertices. Thus, if we are only seeking an approximate distance, we can effectively ignore the heavy vertices and thus reduce the number of edges in the graph. More specifically, our sparsification works as follows. Let HEAVY be the set of heavy vertices in G, and let $G[\text{HEAVY}]$ be the subgraph of G induced by the heavy vertices. Consider the (still unweighted) graph G' which contains all the edges of G, as well as an edge between all pairs of vertices v, w such that v and w are both heavy and are in the same connected component in $G[\text{HEAVY}]$. (Of course these connected components will change as edges in G are deleted, but they are easy to maintain because dynamic connectivity is easy to do efficiently in undirected graphs.) As described, G' in fact contains more edges than G, but it is easy to exactly mimic G' with a sparse graph; include only the edges of G incident to at least one light vertex, and then for every component C in $G[\text{HEAVY}]$ create a new vertex c in G' and add an edge of weight $1/2$ from c to every vertex in the component C. The resulting graph has at most $O(n^{1.5})$ edges: $O(\sqrt{n})$ per light vertex, and a single extra edge of weight $1/2$ per heavy vertex.

Of course distances in G' differ from those in G. In fact they are *shorter* because G' allows us to skip over whole components of heavy vertices. But intuitively, as long as $\mathbf{dist}(s, v)$ is large, $\mathbf{dist}_{G'}(s, v)$ will not be too much smaller than $\mathbf{dist}(s, v)$, because $\mathbf{dist}(s, v)$ is dominated by light vertices anyway, so skipping over the heavy ones does not change much. In particular, we will prove in the paper that

$$\mathbf{dist}_{G'}(s, v) \leq \mathbf{dist}(s, v) < \mathbf{dist}_{G'}(s, v) + 5\sqrt{n} \ .$$

Thus, as long as $\mathbf{dist}(s, v) >> \sqrt{n}$, $\mathbf{dist}_{G'}(s, v) + 5\sqrt{n}$ will be a good approximation to $\mathbf{dist}(s, v)$.

All in all, our algorithm runs two Even and Shiloach trees; $ES(G, s, 5\sqrt{n}\epsilon^{-1})$ handles short distances, while $ES(G', s, n)$ handles long ones. The first part runs in time $O(m\sqrt{n}) = O(n^{2.5})$ because we have bounded depth, while the second runs in time $O(n^{2.5})$ because G' is sparse. To answer queries for a vertex v, the algorithm simply takes the minimum of the subroutine for short distances, and the subroutine for long ones. Using these ideas it is not hard to reduce the total update time to $\tilde{O}(n^2)$ by using $O(\log(n))$ different heaviness thresholds to handle $O(\log(n))$ ranges of $\mathbf{dist}(s, v)$, and taking the minimum of the $O(\log(n))$ subroutines.

4. THE THRESHOLD GRAPH

Definition 4.1 *Given a graph G and a positive integer threshold τ, we say that a vertex v in G is τ-heavy if v has degree at least τ, and we say that v is τ-light otherwise. Let $\text{HEAVY}(\tau)$ be the set of all τ-heavy vertices in G; note that when we say that a vertex v is τ-heavy or τ-light, this is* always *with respect to the main graph G, never with respect to any other graph the algorithm relies on.*

Definition 4.2 *Given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, and an integer $\tau \in [1, n]$, define the threshold graph $G_\tau = (V_\tau, E_\tau)$ as follows (note that although G is unweighted, the edges in G_τ have weights 1 and $1/2$):*

- *V_τ contains every vertex $v \in V$.*
- *V_τ also contains an additional vertex c for each connected component C in the induced subgraph $G[\text{HEAVY}(\tau)]$.*
- *E_τ contains all edges incident to τ-light vertices $v \in V$. All such edges are given weight 1.*

- *For every τ-heavy vertex $v \in V$, E_τ contains an edge from v to c of weight $1/2$, where c is the component vertex in $V_\tau \setminus V$ that corresponds to the component C in $G[\text{HEAVY}(\tau)]$ that contains v.*

For any pair of vertices $s, t \in V$ define $\pi_\tau(s,t)$ to be the shortest path from s to t in G_τ, and define $\mathbf{dist}_\tau(s,t)$ to be the weight of this path.

Lemma 4.3 *The number of edges in the threshold graph G_τ is always $O(n\tau)$*

PROOF. This follows directly from the definition of G_τ. E_τ contains the $O(n\tau)$ edges incident to τ-light vertices in V, as well as a single additional edge for every τ-heavy vertex in V. □

Lemma 4.4 *For any graph $G = (V, E)$, any positive integer threshold τ, and any pair of vertices $s, t \in V$:*

$$\mathbf{dist}_\tau(s,t) \leq \mathbf{dist}(s,t) < \mathbf{dist}_\tau(s,t) + \frac{5n}{\tau}$$

.

PROOF. We first show the simpler claim that $\mathbf{dist}_\tau(s,t) \leq \mathbf{dist}(s,t)$. Consider the shortest $s-t$ path $\pi(s,t) \in G$. We will exhibit a (not necessarily simple) path $P_\tau(s,t) \in G_\tau$ with weight exactly $\mathbf{dist}(s,t)$. $P_\tau(s,t)$ contains all the edges of $\pi(s,t)$ that are incident to some τ-light vertex: these edges have weight 1 in both G and G_τ. The only edges that remain are edges $(v,w) \in \pi(s,t)$ where v and w are both τ-heavy. Since v and w are neighbors in G, they are part of the same connected component C in $G[\text{HEAVY}(\tau)]$, and so G_τ contains a path of length 1 from v to w; namely, the path $(v,c) \circ (c,w)$. We thus replace every edge $(v,w) \in \pi(s,t)$ with a path of length 1 in G_τ. Hence exhibiting a path $P_\tau(s,t) \in G_\tau$ with weight exactly $\mathbf{dist}(s,t)$, as needed.

We now prove that $\mathbf{dist}(s,t) \leq \mathbf{dist}_\tau(s,t) + \frac{5n}{\tau}$. Let $\pi_\tau(s,t)$ be the shortest $s-t$ path in G_τ. Let L_{π_τ} be the set of τ-light vertices $v \in V \bigcap \pi_\tau(s,t)$. Now, let $V^* \subseteq V$ be the set of vertices containing

- All the vertices in L_{π_τ}
- All the τ-heavy vertices in G
- All the neighbors of τ-heavy vertices in G

Let G^* be the subgraph of G induced by V^*. We first show that there must exist an $s-t$ path in G^*. We construct this path by looking at $\pi_\tau(s,t)$. $\pi_\tau(s,t)$ contains edges incident to τ-light vertices in G^*, as well as subpaths of length 2 of the form $(v,c) \circ (c,w)$, where v and w are τ-heavy vertices that are in the same connected component C in $G[\text{HEAVY}(\tau)]$. The edges on $\pi_\tau(s,t)$ incident to light vertices exist in G^* as well, so we can follow those directly. For every subpath $(v,c) \circ (c,w)$, since v and w are both in the same connected component in $G[\text{HEAVY}(\tau)]$, there is a $v-w$ path in G using only heavy vertices, so that path is in G^* as well. We get that there exists an $s-t$ path in G^*.

Now, let $\pi^*(s,t)$ be the *shortest* $s-t$ path in G^*. Let $\mathbf{dist}^*(s,t)$ be the length of $\pi^*(s,t)$. Since G^* is a subgraph of G, we know that $\mathbf{dist}(s,t) \leq \mathbf{dist}^*(s,t)$. We now show that

$$\mathbf{dist}^*(s,t) < \mathbf{dist}_\tau(s,t) + \frac{5n}{\tau} \tag{1}$$

which completes the proof of Lemma 4.4. Let X^* contain all vertices in $\pi^*(s,t)$ that are NOT in L_{π_τ}: observe that by definition of V^*, every vertex $v \in X^*$ is either τ-heavy, or adjacent in G^* to a τ-heavy vertex. Note that $\mathbf{dist}^*(s,t) \leq |L_{\pi_\tau}| + |X^*|$, while $\mathbf{dist}_\tau(s,t) \geq |L_{\pi_\tau}|$ because all the vertices in L_{π_τ} are on $\pi_\tau(s,t)$. Thus, to prove Inequality 1, it suffices to show that

$$|X^*| < \frac{5n}{\tau} . \tag{2}$$

Let Y^* be the set containing every 5th vertex in X^*: that is, Y^* contains the vertex in X^* that is closest to s in G^*, the one that is 6th closest to s, the one that is 11th closest to s, and so on. Clearly, $|Y^*| \geq |X^*|/5$. We complete the proof of Equation 2, and hence of Lemma 4.4 as a whole, by arguing that

$$|Y^*| < \frac{n}{\tau} . \tag{3}$$

To prove Equation 3 above, for any vertex $v \in V_\tau$ we define $\text{BALL}(G^*, v, 2) \subseteq V^*$ to be the set containing v itself, the neighbors of v in G^*, and all vertices at distance 2 from v in G^*. Now, on the one hand, for every $v \in Y^*$ we have that $|\text{BALL}(G^*, v, 2)| > \tau$ because since $v \in Y^* \subset X^*$ we know that v is either itself τ-heavy or adjacent in G^* to a τ-heavy vertex, and so $\text{BALL}(G^*, v, 2)$ must contain that τ-heavy vertex as well as its $\geq \tau$ neighbors. On the other hand, if v and w are both in Y^* then $\text{BALL}(G^*, v, 2)$ and $\text{BALL}(G^*, w, 2)$ must be disjoint because otherwise there would be a path of length at most 4 between v and w in G^*, which contradicts the fact that the subpath of the shortest path $\pi^*(s,t)$ between v and w is of length at least 5. Thus, the total number of vertices among all of the $\text{BALL}(G^*, v, 2)$ for $v \in Y^*$ is strictly greater than $\tau|Y^*|$, but since there are only n vertices in the graph, we have $|Y^*| < n/\tau$, as desired. □

Now that we have shown that distances in G_τ are a close approximation to those in G, we show that these distances, as well as the graph G_τ itself, can be easily maintained.

Lemma 4.5 *Given a graph G subject to a sequence of edge deletions, and a positive integer threshold τ, we can maintain the graph G_τ in* total *time $O(m \log^2(n))$.*
Moreover, $\text{MAX-EDGES}(G_\tau) = O(n\tau + n\log(n))$.

PROOF. G_τ contains two types of edges: those incident to τ-light vertices, and those from a τ-heavy vertex to its component vertex c. The first types of edges are trivial to maintain, since they are simply a subset of the edges in G; when a deletion causes a τ-heavy vertex to become τ-light we must add all of its incident $O(\tau)$ edges to G_τ, but this transition can only happen a single time per vertex over the entire sequence of deletions because vertex degrees are only decreasing. This first type of edges thus requires $O(m)$ time to maintain, and only leads to $O(\min\{n\tau, m\})$ edge insertions into G_τ.

We now show how to maintain the edge from each τ-heavy vertex v to its component vertex c, where c corresponds to the connected component C in $G[\text{HEAVY}(\tau)]$ that contains v. First off, note that $G[\text{HEAVY}(\tau)]$ itself is easy to maintain because it is simply a subgraph of G. We can maintain connected components in $G[\text{HEAVY}(\tau)]$ by using a dynamic connectivity data structure (CDS) on the graph $G[\text{HEAVY}(\tau)]$. We use the CDS of Holm *et al.*[28], which is based on top trees. This CDS can process insertions and deletions into the graph with amortized update time of $O(\log^2(n))$. It is not hard to check that the top

trees used by their algorithm can be augmented to support more than just basic connectivity queries. In particular, their CDS can answer the following queries:

- **connected(u,v):** determines whether u and v are in the same connected component in the current graph. The query time is $O(\log(n))$.
- **size(v)**: returns the size of the connected component of v. The query time is $O(\log(n))$.
- **component(v)**: Returns a list of all the vertices in the same connected component as v. The query time is $O(\log(n)$ + number of vertices returned).

To maintain the graph G_τ as G changes, we will run the above CDS on the graph $G[\text{HEAVY}(\tau)]$. Note that $G[\text{HEAVY}(\tau)]$, like G, is only subject to edge deletions. When the adversary deletes an edge (u, v) in G where both u and v are τ-heavy, this edge must be deleted from $G[\text{HEAVY}(\tau)]$ as well, and this deletion is processed by the CDS in time $O(\log^2(n))$. Similarly, when a vertex $v \in V$ transitions from τ-heavy to τ-light, all of its incident edges must be deleted from $G[\text{HEAVY}(\tau)]$, and processed by the CDS. Each edge is deleted from $G[\text{HEAVY}(\tau)]$ at most once, so the total update time of the CDS is $O(m \log^2(n))$.

We must now show how to use the connectivity information maintained by the CDS to maintain the graph G_τ; in particular, how to maintain the edges from a τ-heavy vertex to its component vertex C. Whenever an edge $(u, v) \in G[\text{HEAVY}(\tau)]$ is deleted, we first query the CDS in $O(\log(n))$ time to check whether u and v are still part of the same connected component in $G[\text{HEAVY}(\tau)]$; if yes, the edges of G_τ do not change, and we are done. Otherwise, the deletion of (u, v) has caused the component to split into two. We now query CDS.size(u) and CDS.size(v) to determine in $O(\log(n))$ time which of the two parts is smaller. Say, wlog, that CDS.size(v) $\leq$ CDS.size(u). Let C be the original component that contained both u and v before the deletion of (u, v). Let C_v be the component containing v after the deletion. We can use CDS.component(v) to find all the vertices in C_v in time $O(\log(n) + |C_v|)$. Before the deletion, G_τ contained an edge from every vertex in C to the component vertex $c \in V_\tau \setminus V$. After the deletion, we add a new component vertex c_v to G_τ, and for every $w \in C_v$ we remove the edge (w, c) and add the edge (w, c_v). This takes time $O(|C_v|)$ and makes $O(|C_v|)$ edge changes to $G[\text{HEAVY}(\tau)]$. Amortized over all edge deletions in $G[\text{HEAVY}(\tau)]$ we have $\sum |C_v| \leq n \log(n)$ because edges in $G[\text{HEAVY}(\tau)]$ are only being deleted, so components are only splitting apart, and each time a vertex w is part of the smaller component C_v in a component split, we know that its component has shrunk by a factor of at least two.

Thus, the total time to process a deletion in $G[\text{HEAVY}(\tau)]$ is dominated by the $O(\log^2(n))$ update time of the CDS, yielding the desired $O(m \log^2(n))$ total update time. Moreover, from the bounds above, we see that at most $O(n\tau + n \log(n))$ edges are inserted into the graph; $O(n\tau)$ of the first type (edges incident to a τ-light vertex), and $O(n \log(n))$ of the second (edges from a τ-heavy vertex to its component vertex c). By Lemma 4.3, the number of edges in the initial G_τ is $O(n\tau)$, so $\text{MAX-EDGES}(G_\tau) = O(n\tau + n \log(n))$.

□

Lemma 4.6 *Given a graph G subject to a sequence of edge deletions, a positive integer threshold τ, and a pair of vertices u, v in G, the distance $\mathbf{dist}_\tau(u, v)$ never decreases as edges in G are deleted.*

PROOF. Say that the adversary deleted edge (x, y) in G. Note that any path in G_τ consists of a concatenation of subpaths of length 1 between vertices in V; each subpath is either an edge of weight 1 incident to a τ-light vertex, or two edges of weight 1/2 through a component vertex $c \in V_\tau \setminus V$. Thus, to show that distances in G_τ do not decrease, we show that for any pair of vertices $a, b \in V$ such that $\mathbf{dist}_\tau(a, b) = 1$ *after* the deletion of (x, y), we also had $\mathbf{dist}_\tau(a, b) = 1$ *before* the deletion of (x, y). We know that $\mathbf{dist}_\tau(a, b) = 1$ after the deletion if and only if edge (a, b) is in E, AND/OR a and b are in the same connected component in $G[\text{HEAVY}(\tau)]$. But either of these cases would clearly hold before the deletion of an edge as well, so we had $\mathbf{dist}_\tau(a, b) = 1$ before the deletion. □

Lemma 4.7 *Given a graph $G = (V, E)$ subject to a sequence of deletions, a fixed source s, a positive integer threshold τ, and a depth bound d, we can maintain the distance $\text{BOUND}_d(\mathbf{dist}_\tau(s, v))$ for all vertices v in a* total *update time of $O(m \log^2(n) + n \cdot d \cdot (\tau + \log(n))$.*

PROOF. We simply maintain the graph G_τ as edges in G are deleted, and run $\mathbf{ES}(G_\tau, s, d)$ (See Definition 2.3). By Lemma 4.5, we can maintain the graph G_τ in time $O(m \log^2(n))$. Moreover, by Lemma 4.5 we have $\text{MAX-EDGES}(G_\tau) = O(n\tau + n \log(n))$. This bound on $\text{MAX-EDGES}(G_\tau)$ implies that the total number of changes made to G_τ is $\Delta = O(n\tau + n \log(n))$. By Lemma 4.6 distances in G_τ never decrease, and all weights in G_τ are either 1/2 or 1, so by Lemma 2.4 and Corollary 2.6, the total running time of the ES-tree is $O(\text{MAX-EDGES}(G_\tau) \cdot d + \Delta)$, which is $O(n \cdot d \cdot (\tau + \log(n)))$. □

5. THE DECREMENTAL SSSP ALGORITHM

Our goal is to maintain approximate distances from a fixed source s. For every integer $i \in [1, \lfloor \log(n) \rfloor]$, let $\tau_i = \frac{n}{2^i}$, let $d_i = 2^i \cdot \frac{10}{\epsilon}$, and for every vertex v let

$$\widehat{\mathbf{dist}}_i(v) = \text{BOUND}_{d_i}(\mathbf{dist}_{\tau_i}(s, v))$$

Let $\widehat{\mathbf{dist}}(v) = \min_i\{\widehat{\mathbf{dist}}_i(v) + 5 \cdot 2^i\}$. When the adversary queries the distance to a vertex v, our algorithm returns $\widehat{\mathbf{dist}}(v)$.

The execution of the algorithm is simple. By Lemma 4.7, for any i we maintain $\widehat{\mathbf{dist}}_i(v)$ for all vertices v in total update time $O(m \log^2(n) + n \cdot d_i \cdot (\tau_i + \log(n)) = O(m \log^2(n) + n^2\epsilon^{-1} + n \log(n) d_i)$. Doing this for every i yields total update time $O(m \log^3(n) + n^2 \log(n)\epsilon^{-1})$ because $\sum_i d_i = O(n\epsilon^{-1})$. To maintain all the $\widehat{\mathbf{dist}}(v)$, for each vertex v we create a min-heap HEAP_v containing $\widehat{\mathbf{dist}}_i(v)$ for every i. The algorithm can access any $\widehat{\mathbf{dist}}(v)$ in $O(1)$ time by looking at the minimum of the heap, thus leading to an $O(1)$ query time. Maintaining the heaps is easy: each $\widehat{\mathbf{dist}}_i(v)$ can change at most $d_i = O(2^i\epsilon^{-1})$ times, so there will be at most $\sum_i d_i = n\epsilon^{-1}$ changes to each HEAP_v, and since each heap contain $O(\log(n))$ elements, a change requires $O(\log \log(n))$ time to process. Maintaining HEAP_v for all v thus requires total update time only $O(n^2 \log \log(n)\epsilon^{-1})$.

We now turn to proving that for any vertex v, $\mathbf{dist}(s, v) \leq \widehat{\mathbf{dist}}(s, v) \leq (1+\epsilon)\mathbf{dist}(s, v)$. The fact that $\mathbf{dist}(s, v) \leq \widehat{\mathbf{dist}}(s, v)$

follows directly from Lemma 4.4, because for every i

$$\begin{aligned}\mathbf{dist}(s,v) &\leq \mathbf{dist}_{\tau_i}(s,v) + \frac{5n}{\tau_i} \\ &\leq \widehat{\mathbf{dist}}_i(s,v) + \frac{5n}{\tau_i} \\ &= \widehat{\mathbf{dist}}_i(s,v) + 5\cdot 2^i \ .\end{aligned}$$

To prove that $\widehat{\mathbf{dist}}(s,v) \leq (1+\epsilon)\mathbf{dist}(s,v)$, we need to show that for *some* i, we have $\widehat{\mathbf{dist}}_i(s,v) + 5\cdot 2^i \leq (1+\epsilon)\mathbf{dist}(s,v)$. Let k be the largest index for which $\mathbf{dist}(s,v) \geq 2^k \cdot \frac{5}{\epsilon}$, so

$$2^k \cdot \frac{10}{\epsilon} \geq \mathbf{dist}(s,v) \geq 2^k \cdot \frac{5}{\epsilon} \ . \tag{4}$$

Now, by Lemma 4.4, $\mathbf{dist}_{\tau_k}(s,v) \leq \mathbf{dist}(s,v) \leq 2^k \cdot \frac{10}{\epsilon} = d_k$, so we have

$$\widehat{\mathbf{dist}}_k(s,v) = \textsc{bound}_{d_k}(\mathbf{dist}_{\tau_k}(s,v)) = \mathbf{dist}_{\tau_k}(s,v) \leq \mathbf{dist}(s,v) \ .$$

Thus,

$$\widehat{\mathbf{dist}}_k(s,v) + 5\cdot 2^k \leq \mathbf{dist}(s,v) + 5\cdot 2^k \leq (1+\epsilon)\mathbf{dist}(s,v) \ .$$

(The last inequality follows from $\mathbf{dist}(s,v) \geq 2^k \cdot \frac{5}{\epsilon}$ in Equation 4.)

6. THE INCREMENTAL SSSP ALGORITHM

In this section we sketch the modifications needed to make our algorithm incremental rather than decremental.

We first remind the reader that the Even-Shiloach algorithm works with the same bounds in the incremental setting. That is, Lemma 2.4 can be formulated as follows. Given a dynamic graph $G = (V, E)$ with positive integer weights, and s is a fixed source, and say that for every vertex v we are guaranteed that the distance $\mathbf{dist}(s,v)$ never increases due to an edge deletions. Then, the *total* update time of $\mathbf{ES}(G, s, d)$ over the entire sequence of edge updates is $O(m \cdot d + \Delta)$, where $m = \textsc{max-edges}(G)$, and Δ is the total number of edge changes.

In our incremental algorithm we invoke the incremental version of the Even-Shiloach algorithm described above. We will therefore need to make sure that distances never increases during edge deletions.

Similarly to the decremental case, we maintain the threshold graph G_τ described in Section 4. The only difference is that we need to maintain G_τ incrementally. Initially the graph is empty and all nodes are τ-light. As edges are added to G, some of these nodes may become τ-heavy (note that once a node becomes τ-heavy it will always remain τ-heavy as edges may only be added to G). We maintain the connected components of the τ-heavy nodes using again the result of Holm *et al.*[28] that works also in the incremental settings (with the same time bounds).

The graph G_τ is the same as in the decremental algorithm: for every connected component C in $G[\textsc{heavy}]$, G_τ contains a new node c that corresponds to this heavy connected component, with edges of weight $1/2$ from c to every vertex in C. As new nodes in G become τ-heavy, components in $G[\textsc{heavy}]$ can merge; once two connected components in $G[\textsc{heavy}]$ merge, their corresponding nodes c and c' are also merged by picking the smaller connected component, say c', and replacing all edges (c', v) with edges (c, v). Similarly to the analysis in Section 4 a node may belong to the smaller component at most $\log n$ times.

Since we maintain exactly the same threshold graph G_τ as in Section 4, the rest of the algorithm and correctness is identical to the decremental case. Moreover, using arguments analogous to those in Lemma 4.6, it is not hard to see that distances in G_τ never increase. Using the same analysis in the decremental case, we get that the asymptotic bound of the total update time of our incremental algorithm is the same as that of the decremental one.

7. CONCLUSIONS

In this paper we present the first *deterministic* decremental SSSP algorithm that goes beyond the Even-Shiloach bound of $O(mn)$ total update time. More precisely, we introduce a simple deterministic decremental SSSP algorithm for unweighted and undirected graphs with stretch $1+\epsilon$ and total update time of $\tilde{O}(n^2)$. For dense instances where $m = \Omega(n^{2-1/\sqrt{\log(n)}})$, our algorithm is also faster than all existing randomized algorithms.

We leave a couple of important open questions. The first is whether a deterministic algorithm can go beyond $O(n^2)$ total update time bound for sparse graphs. Or even more ambitiously, can we get the total update time all the way down to near linear, thus matching the randomized state of the art of Henzinger, Krinninger and Nanongkai [23]?

The second question is whether there exist better deterministic algorithms for decremental SSSP in *weighted* undirected graphs, where an update can either delete an edge or increase an edge weight. For exact distances, the state of the art is still the Even and Shiloach algorithm, which has total update time $O(mnW)$. Bernstein [9] showed that if we allow a $(1+\epsilon)$ approximation we can use scaling to reduce this to $O(mn\log(W))$. In this paper we achieve $O(n^2W)$: can we bring this down to $O(n^2\log(W))$? (For randomized algorithms, Henzinger *et al* achieve a close to optimal update time of $O(m^{1+o(1)}\log(W))$.)

Thirdly, is there a deterministic algorithm that goes beyond the $O(mn)$ total update time in *directed* unweighted graphs? Note that all three of the open problems above have no known solution even when we allow a large constant approximation ratio.

Finally, many dynamic shortest paths algorithms of all varieties are randomized and assume a non-adaptive oblivious adversary; a natural open question is to derandomize these algorithms. For example, all the decremental approximate APSP algorithms with total update time less than $O(mn)$ are randomized (e.g., [8, 38, 20, 1, 3]). In fact, as far as we know it is not known how to achieve a decremental dynamic APSP algorithm with less than $O(mn)$ total update time with *any* constant approximation. Showing such an algorithm or proving impossibility results is a very challenging and interesting open problem.

8. REFERENCES

[1] I. Abraham and S. Chechik. Dynamic decremental approximate distance oracles with $(1+\epsilon, 2)$ stretch. *CoRR*, abs/1307.1516, 2013.

[2] I. Abraham, S. Chechik, and C. Gavoille. Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 1199–1218, 2012.

[3] I. Abraham, S. Chechik, and K. 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.

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

[5] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 394–403, 2003.

[6] 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'02.

[7] S. Baswana and T. 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.

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

[9] A. 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.

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

[11] A. Bernstein and L. 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] S. Chechik. Compact routing schemes with improved stretch. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 33–41, 2013.

[13] S. Chechik. Approximate distance oracles with constant query time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 2014.

[14] S. Chechik. Approximate distance oracles with improved bounds. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing (STOC)*, pages 1–10, 2015.

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

[16] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *J. Comput. Syst. Sci.*, 72(5):813–837, 2006.

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

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

[19] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006. Announced at FOCS'01.

[20] M. Henzinger, S. Krinninger, and D. 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.

[21] M. Henzinger, S. Krinninger, and D. 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.

[22] M. Henzinger, S. Krinninger, and D. 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.

[23] M. Henzinger, S. Krinninger, and D. 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.

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

[25] M. Henzinger, S. Krinninger, D. Nanongkai, and T. 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.

[26] M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31(2):364–374, 2001.

[27] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997. Announced at STOC'94.

[28] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[29] V. 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.

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

[31] P. N. Klein and S. Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22(3):235–249, 1998. Announced at WADS'93.

[32] A. Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge,*

Massachusetts, USA, 5-8 June 2010, pages 121–130, 2010.

[33] D. Peleg and A. A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[34] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747, 1989.

[35] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[36] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34(6):1398–1431, 2005.

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

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

[39] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[40] M. Thorup. On RAM priority queues. *SIAM J. Comput.*, 30(1):86–109, 2000.

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

[42] M. 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.

[43] M. Thorup and U. Zwick. Compact routing schemes. In *SPAA*, pages 1–10, 2001.

[44] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.