Near-Optimal Decremental Hopsets with Applications

Jakub Łącki
Google Research, New York, NY, USA

Yasamin Nazari
Universität Salzburg, Austria

Abstract

Given a weighted undirected graph $G = (V, E, w)$, a hopset H of *hopbound* β and *stretch* $(1+\epsilon)$ is a set of edges such that for any pair of nodes $u, v \in V$, there is a path in $G \cup H$ of at most β hops, whose length is within a $(1+\epsilon)$ factor from the distance between u and v in G. We show the first efficient decremental algorithm for maintaining hopsets with a *polylogarithmic* hopbound. The update time of our algorithm matches the best known static algorithm up to polylogarithmic factors. All the previous decremental hopset constructions had a *superpolylogarithmic* (but subpolynomial) hopbound of $2^{\log^{\Omega(1)} n}$ [Bernstein, FOCS'09; HKN, FOCS'14; Chechik, FOCS'18].

By applying our decremental hopset construction, we get improved or near optimal bounds for several distance problems. Most importantly, we show how to decrementally maintain $(2k-1)(1+\epsilon)$-approximate all-pairs shortest paths (for any constant $k \geq 2$), in $\tilde{O}(n^{1/k})$ amortized update time[1] and $O(k)$ query time. This improves (by a polynomial factor) over the update-time of the best previously known decremental algorithm in the *constant* query time regime. Moreover, it improves over the result of [Chechik, FOCS'18] that has a query time of $O(\log\log(nW))$, where W is the aspect ratio, and the amortized update time is $n^{1/k} \cdot (\frac{1}{\epsilon})^{\tilde{O}(\sqrt{\log n})}$. For sparse graphs our construction nearly matches the best known static running time / query time tradeoff.

We also obtain near-optimal bounds for maintaining approximate multi-source shortest paths and distance sketches, and get improved bounds for approximate single-source shortest paths. Our algorithms are randomized and our bounds hold with high probability against an *oblivious* adversary.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic Algorithms, Data Structures, Shortest Paths, Hopsets

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.86

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2009.08416

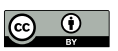
Funding *Yasamin Nazari*: This work was conducted in part while the author was an intern at Google and a PhD student at Johns Hopkins University. Supported in part by NSF award CCF-1909111 and by Austrian Science Fund (FWF) grant P 32863-N.

Acknowledgements The authors would like to thank Michael Dinitz and Sebastian Forster for the helpful discussions.

1 Introduction

Given a weighted undirected graph $G = (V, E, w)$, a hopset H of *hopbound* β and *stretch* $(1+\epsilon)$ (or, a $(\beta, 1+\epsilon)$-hopset) is a set of edges such that for any pair of nodes $u, v \in V$, there is a path in $G \cup H$ of at most β hops, whose length is within a $(1+\epsilon)$ factor from the distance between u and v in G (see Definition 5 for a formal statement).

[1] Throughout this paper we use the notation $\tilde{O}(f(n))$ to hide factors of $O(\text{polylog}\,(f(n)))$.



Hopsets, originally defined by [13], are widely used in distance related problems in various settings, such as parallel shortest path computation [13, 15, 18, 28], distributed shortest path computation [10, 17, 29], routing tables [16], and distance sketches [14, 16]. In addition to their direct applications, hopsets have recently gained more attention as a fundamental object (e.g. [1, 4, 17, 23]), and are known to be closely related to several other fundamental objects such as additive (or near-additive) spanners and emulators [19].

A key parameter of a hopset is its hopbound. In many settings, after constructing a hopset, we can approximate distances in a time that is proportional to the hopbound. For instance, in parallel or distributed settings a hopset with a hopbound of β allows us to compute approximate single-source shortest path in β parallel rounds (e.g. by using Bellman-Ford). For many applications, such as approximate APSP (all-pairs shortest paths), MSSP (multi-source shortest paths), computing distance sketches, and diameter approximation, where we require computing distances from *many* sources, we are interested in the regime where the hopbound is polylogairthmic. Indeed, we obtain improved (and in some cases near-optimal) bounds for several of these problems in decremental settings.

In this paper, we study the maintenance of hopsets in a dynamic setting. Namely, we give an algorithm that given a weighted undirected graph G maintains a hopset of G under edge deletions. Our algorithm covers a wide range of hopbound/update time/hopset size tradeoffs. Importantly, we get the first efficient algorithm for decrementally maintaining a hopset with a *polylogarithmic* hopbound. In this case, assuming G initially has m edges and n vertices, our algorithm takes $O(mn^{\rho})$ time, given any constant $\rho > 0$, and maintains a hopset of polylogarithmic hopbound and $1+\epsilon$ stretch. This matches (up to polylogarithmic factors) the running time of the best known static algorithm [17, 18] for computing a hopset with polylogarithimic hopbound and $(1+\epsilon)$ stretch.

▶ **Theorem 1.** *Given an undirected graph $G = (V, E)$ with polynomial weights[2], subject to edge deletions, we can maintain a $(\beta, 1+\epsilon)$-hopset of size $\tilde{O}(n^{1+\frac{1}{2^k-1}})$ in total expected update time $\tilde{O}(\frac{\beta}{\epsilon} \cdot (m + n^{1+\frac{1}{2^k-1}})n^{\rho})$, where $\beta = (O(\frac{\log n}{\epsilon} \cdot (k + 1/\rho)))^{k+1/\rho+1}$, $k \geq 1$ is an integer, $0 < \epsilon < 1$ and $\frac{2}{2^k-1} < \rho < 1$.*

In the decremental setting, to the best of our knowledge, the previous state-of-the art hopset constructions have a hopbound of $2^{\tilde{O}(\log^{3/4} n)}$ [21], or $(1/\epsilon)^{\tilde{O}(\sqrt{\log n})}$ [5, 12]. As a special case, by setting $\rho = (2^k - 1)^{-1} = \frac{\log\log n}{\sqrt{\log n}}$, we can maintain a hopset with hopbound $2^{\tilde{O}(\sqrt{\log n})}$ in $2^{\tilde{O}(\sqrt{\log n})}$ amortized time. More importantly, by setting ρ and k to a constant, we can maintain a hopset of *polylogarithmic* hopbound.

While hopsets are extensively studied in other models of computation (e.g. distributed and parallel settings), their applicability in dynamic settings is less understood. Examples of results utilizing hopsets include the state-of-the art decremental SSSP algorithm for undirected graphs by Henzinger, Krinninger and Nanongkai [21], and implicit hopsets considered in [5,12]. As stated, these decremental hopset algorithms as stated only provide a *superpolylogarithmic* hopbound. It may be possible (while not discussed) to use the hop-reduction techniques of [21] (inspired by a similar technique in [5]) to obtain a wider-range of tradeoffs, however to the best of our knowledge these techniques do not lead to near-optimal size/hopbound tradeoffs[3]. Hence our result constitutes the first near-optimal decremental algorithm for maintaining hopsets with in a wide-range of settings including polylogarithmic hopbound.

---

[2] If weights are not polynomial the $\log n$ factor will be replaced with $\log W$ in the hopbound, and a factor of $\log^2 W$ will be added to the update time, where W is the aspect ratio.

[3] In particular, in all regimes the algorithm of [21] gives a hopset with size that is super-linear in number

Discussion on hopset limitations and alternative techniques. In [1] it was shown that for a $(\beta, 1+\epsilon)$-hopset with size $n^{1+\frac{1}{2^k-1}-\delta}$ for any fixed k, ϵ and $\delta > 0$ we must have[4] $\beta = \Omega_k(\frac{1}{\epsilon})^k$. Their lower bound suggests that we cannot construct a $(\beta, 1+\epsilon)$-hopset of size $\tilde{O}(n)$ with $\beta = \operatorname{poly}\log(n)$ hopbound, implying that hopsets cannot be used for obtaining optimal time (i.e. polylog amortized time) for *sparse* graphs and *very small* ϵ. However when the graph is slightly denser ($|E| = n^{1+\Omega(1)}$), the approximation factor is slightly larger (see e.g. [4,15]), or we aim to compute distances from many sources (in APSP or MSSP), using hopsets may still lead to optimal algorithms. Indeed, we show that our decremental hopsets allow us to obtain a running time matching the best static algorithm (up to polylogarithmic factors) both in $(2k-1)$-APSP and $(1+\epsilon)$-MSSP. We leave it as an open problem if hopsets can be used to obtain linear time algorithms for SSSP with *larger* approximation factors (e.g. $\epsilon \geq 1$), since as stated, the lower bound of [1] does not apply in this case.

It is worth noting that in Theorem 12 we first give a decremental algorithm that maintains static hopsets of [18] that matches the size/hopbound tradeoff in the lower bound of [1]. However, as we will see, this algorithm has a large update time, and thus we propose a new hopset with slightly worse size/hopbound tradeoff that can be maintained much more efficiently. This efficient variant has additional polylogarithmic (in aspect ratio) factors in the hopbound relative to the existentially optimal construction.

Finally, for *single source* shortest path computation in other models recently algorithms based on continuous optimization techniques are proposed (e.g. [2,3,25]) that outperform algorithms based only on combinatorial objects such as hopsets/emulators. These optimization techniques lead to much better dependence on ϵ, but are less suitable when there are many sources, as the running time scales with the number of sources. Interestingly, the authors of [2] use low-hop combinatorial structures with *larger (polylogrithmic)* stretch as a subroutine in their continuous optimization framework. Hence understanding both combinatorial and optimization directions seems crucial for distance computation in general.

1.1 Applications of Our Decremental Hopsets

To illustrate applicability of our decremental hopset algorithm, we show how it yields improved algorithms for decremetanlly maintaining shortest paths from a fixed set S of sources. We consider different variants of the problem which differ in the size of S: the single-source shortest paths (SSSP) problem ($|S| = 1$), all-pairs shortest paths (APSP) problem ($S = n$, where n is the number of vertices of the input graph), as well as the multi-source shortest paths (MSSP) problem (S is of arbitrary size), which is a generalization of the previous two.

Near-Optimal approximate APSP. We give a new decremental algorithm for maintaining approximate all-pairs shortest paths (APSP) with constant query time.

▶ **Theorem 2** (Approximate APSP). *For any constant integer*[5] $k \geq 2$, *there is a data structure that can answer* $(2k-1)(1+\epsilon)$*-approximate distance queries in a given a weighted undirected graph* $G = (V, E, w)$ *subject to edge deletions. The total expected update time*

of edges m (e.g. m^{1+p} for a parameter p), while our hopset size is $O(n^{1+p})$ for some (other but similar) parameter p, which is a constant when the hopbound is polylogarithmic. Moreover, our techniques lead to near-optimal approximate APSP, whereas it is unclear how to get comparable bounds using techniques in [21], as they do not maintain Thorup Zwick-based clusters.

[4] Ω_k hides exponential a factor of roughly $1/(k2^k)$. As written in [1] they assume k is constant (and hence the sparse hopset regime is not covered), but they also indicate that a tighter analysis could change the exact relationship between k and ϵ and hence allow a better k dependence and covering the sparse case (see Theorem 4.6 and Remark 4.7 in [1]).

[5] The k here should not be confused with the parameter k in the hopset size.

over any sequence of edge deletions is $\tilde{O}(mn^{1/k})$ and the expected size of the data structure is $\tilde{O}(m + n^{1+1/k})$. Each query for the distance between two vertices is answered in $O(k)$ worst-case time.

Our result improves upon a decremental APSP algorithm by Chechik [12] in a twofold way. First, for constant k, our update time bound is better by a $(1/\epsilon)^{O(\sqrt{\log n})}$ factor. Second, we bring down the query time from $O(\log\log(nW))$ to constant. We note that in the area of distance oracles a major goal is to preprocess a data structure that can return a distance estimate in *constant* time [11, 27, 30, 36][6].

Our results match the best known static algorithm with the same tradeoff (up to $(1+\epsilon)$ in the stretch and polylog in time) by Thorup-Zwick [34] for sparse graphs. For dense graphs there have been improvements by [36] in static settings.

Prior to [12], Roditty and Zwick [31] gave an algorithm for maintaining Thorup-Zwick distance oracles in total time $\tilde{O}(mn)$, stretch $(2k-1)(1+\epsilon)$ and $O(k)$ query time for *unweighted graphs*. Later on, Bernstein and Roditty [9] gave a decremental algorithm for maintaining Thorup-Zwick distance oracles in $O(n^{2+1/k+o(1)})$ time using emulators also only for *unweighted graphs*.

Distance Sketches. Another application of our hopsets with polylogarithmic hopbound is a near-optimal decremental algorithm for maintaining distance sketches (or distance labeling); an important tool in the context of distance computation. The goal is to store a small amount of information, a sketch, for each node, such that the distance between any pair of nodes can be approximated only using their sketches (without accessing the rest of the graph). Distance sketches are particularly important in networks, and distributed systems [16, 32], and large-scale graph processing [14]. Their significance is that at query time we only need to access/communicate the small sketches rather than having to access the whole graph. This is specially useful for processing large data when queries happen more frequently than updates.

The Thorup-Zwick [34] algorithm can be used to obtain distance sketches of expected size $O(kn^{1/k})$ (for each node) that supports $(2k-1)$-approximate queries in $O(k)$ time (in static settings), and this is known to be tight assuming a well-known girth conjecture. Our approximate APSP data structure has the additional property that the information stored for each node is a distance sketch of expected size $O(kn^{1/k})$ that supports $(2k-1)(1+\epsilon)$-approximate queries. Hence we can maintain distance sketches that almost match the guarantees of the best static algorithm. More specifically, for a fixed size our algorithm matches the best known static construction up to a $(1+\epsilon)$-factor in the stretch and polyloagrithmic factors in the update time. In decremental settings, distance oracles of [34], and hence distance sketches with the guarantees described are studied by [9, 31], but our total update time of $\tilde{O}(mn^{1/k})$ (for constant $k \geq 2$) significantly improves over these results. In particular [31] maintains these distance sketches in a total update time of $\Omega(mn)$, and [9] requires total update time of $O(n^{2+1/k+o(1)})$.

Near-Optimal $(1+\epsilon)$-MSSP. Our next result is a near-optimal algorithm for multi-source shortest paths.

[6] We need to store the original graph in addition to the distance oracle in order to update the distances and maintain correctness, however we do *not* need the whole graph for *querying distances* as we will also point out in describing the applications in maintaining distance sketches.

▶ **Theorem 3** (MSSP). *There is a data structure which given a weighted undirected graph $G = (V, E)$* explicitly *maintains $(1+\epsilon)$-approximate distances from a set of s sources in G under edge deletions, where $0 < \epsilon < \frac{1}{2}$ is a constant. Assuming that $|E| = n^{1+\Omega(1)}$ and $s = n^{\Omega(1)}$, the total expected update time is $\tilde{O}(sm)$. The data structure is randomized and works against an oblivious adversary.*

We note that total update time matches (up to polylogarithmic factors) the running time of the best known *static* algorithm for computing $(1+\epsilon)$-approximate distances from s sources for a wide range of graph densities. While for very dense graphs, using algorithms based on fast matrix multiplication is faster, the running time of our decremental algorithm matches the best known results in the static settings (up to polylogarithmic factors) whenever $ms = n^{\delta}$, for a constant $\delta \in (1, 2.37)$.

In the dynamic setting, our algorithm improves upon algorithms obtained by using hopsets of Henzinger, Krinninger and Nanongkai [21], or emulators of Chechik [12], both of which give a total update time of $O(sm \cdot 2^{\tilde{O}(\log^{\gamma} n)}), 0 < \gamma < 1$ (for a constant γ). In particular, by maintaining a hopset with polylogarithmic hopbound in $\tilde{O}(sm)$ time, we can maintain approximate SSSP from each source in $\tilde{O}(m)$ time. In contrast, in [12, 21] with hopset of hopbound $2^{\tilde{O}(\log^{\gamma} n)}$ is maintained, which if one simply applies existing techniques, results in a total update time of $m2^{\tilde{O}(\log^{\gamma} n)}$. In the general case, i.e., for very sparse graphs, the update bound of our algorithm is $sm2^{\tilde{O}(\sqrt{\log n})}$, which is similar but slightly better than the bound obtained by [21], and slightly improves over dependence on ϵ over [12].

Improved bounds for $(1+\epsilon)$-SSSP. Finally, in order to better demonstrate how our techniques compare to previous work, we show that we can obtain a slightly improved bound for decremental single-source shortest paths.

▶ **Theorem 4.** *Given an undirected and weighted graph $G = (V, E)$, there is data structure for maintaining $(1+\epsilon)$-approximate distances from a source $s_0 \in V$ under edge deletions, where $0 < \epsilon < 1$ is a constant and $|E| = n \cdot 2^{\tilde{\Omega}(\sqrt{\log n})}$. The total expected update time of the data structure is $m \cdot 2^{\tilde{O}(\sqrt{\log n})}$. There is an additional factor of $O(\frac{1}{\epsilon})^{\frac{\sqrt{\log n}}{\log\log n}}$ in the running time for non-constant ϵ.*

The amortized update time of our algorithm over all m deletions is $2^{\tilde{O}(\sqrt{\log n})}$. This improves upon the state-of-the art algorithm of [21], whose amortized update time is $2^{\tilde{O}(\log^{3/4} n)}$. We note that the techniques of [12] can also be used to obtain $(1+\epsilon)$-SSSP in amortized update time $\tilde{O}(1/\epsilon)\sqrt{\log n}$. This is close to our update time, but we get a better bound with respect to the dependence on ϵ.

Recent developments on decremental shortest paths. Recently and after a preprint of this paper was published, a decremental *deterministic* $(1+\epsilon)$-SSSP also with amortized update time of $n^{o(1)}$ was proposed by [8]. Several other recent results have also focused on deterministic dynamic shortest path algorithms or algorithms that work against an *adaptive adversary* (e.g. [6–8, 20]) most of which also use hopsets or related objects such as emulators. Our work leaves an open problem on whether hopsets with small hopbound can also be maintained and utilized deterministically[7]. This could have applications in

[7] One possible direction is considering derandomization of Throup-Zwick based clustering in static settings [34] combined with our techniques.

deterministic approximate all-pairs shortest paths, which could in turn have implications in using decremental shortest path algorithms for obtaining faster algorithms in classic/static settings (e.g. see [26]).

Hopsets vs. emulators. A majority of the previous work on dynamic distance computation are based on sparse *emulators* (e.g. [5, 9, 12]). For a graph $G = (V, E)$, an emulator $H' = (V, E')$ is a graph such that for any pair of nodes $x, y \in V$, there is a path in H' that approximates the distance between x and y on G (possibly with both multiplicative and additive factors). While there are some similarities in algorithms for constructing these objects, their analysis is different. More importantly, their maintenance and utilization for dynamic shortest paths have significant differences. An emulator approximates distances without using the original graph edges and hence we can restrict the computation to a sparser graph, whereas for using hopsets we also need the edges in the original graph. On the other hand, hopsets allow one to only consider paths with few hops.

1.2 Preliminaries and Notation

Given a weighted undirected graph $G = (V, E, w)$, and a pair $u, v \in V$ we denote the (weighted) shortest path distance by $d_G(u, v)$. We denote by $d_G^{(h)}(u, v)$ the length of the shortest path between u and v among the paths that use at most h-hops, and call this the h-hop limited distance between u and v.

In this paper, we are interested in designing *decremental* algorithms for distance problems in weighted graphs. In the decremental setting, the updates are only edge deletions or weight increases. This is as opposed to an *incremental* setting in which edges can be inserted, or a *fully dynamic* setting, in which we have both insertions and deletions. Specifically, given a weighted graph $G = (V, E, w)$, we want to support the following operations: DELETE$((u, v))$, where $(u, v) \in E$, which removes the edge (u, v), DISTANCE(s, u), which returns an (approximate) distance between a source s and any $u \in V$, and INCREASE$((u, v), \delta)$, which increases the weight of the edge (u, v) by $\delta > 0$. While our results also allow handling weight increases, in stating our theorems for simplicity we use the term *total update time* to refer to a sequence of up to m deletions.

▶ **Definition 5.** *Let $G = (V, E, w)$ be a weighted undirected graph. Fix $d, \epsilon > 0$ and an integer $\beta \geq 1$. A $(d, \beta, 1+\epsilon)$-*hopset *is a graph $H = (V, E(H), w_H)$ such that for each $u, v \in V$, where $d_G(u, v) \leq d$, we have $d_G(u, v) \leq d_{G \cup H}^{(\beta)}(u, v) \leq (1+\epsilon) d_G(u, v)$. We say that β is the* hopbound *of the hopset and $1+\epsilon$ is the* stretch *of the hopset. We also use $(\beta, 1+\epsilon)$-hopset to denote a $(\infty, \beta, 1+\epsilon)$-hopset.*

We sometimes call a $(d, \beta, 1+\epsilon)$-hopset a *d-restricted hopset*, when the other parameters are clear. We also sometimes consider hopset edges added for a specific distance range $(2^j, 2^{j+1}]$, which we call a hopset for a single distance *scale*.

In analyzing dynamic algorithms we sometimes also use a time subscript t to denote a distance (or a weight) after the first t updates. In particular we use $d_{t,G}(u, v)$ to denote the distance between u and v after t updates, and similarly use $d_{t,G}^{(h)}(u, v)$ to denote h-hop limited distance between u and v at time t.

2 Overview of Our Algorithms

The starting point of our algorithm is a known static hopset construction [18, 23]. We first review this construction. As we shall see, maintaining this data structure dynamically directly would require update time of up to $O(mn)$. Our first technical contribution is another

hopset construction that captures some of the properties of the hopsets of [18, 23], but can be maintained efficiently in a decremental setting. We then explain how by hierarchically maintaining a sequence of data structures we can obtain a near-optimal time and stretch tradeoff.

2.1 Static Hopset of [16]

In this section we outline the (static) hopset construction of Elkin and Neiman [18][8] (which is similar to [23]). We will later give a new (static) hopset algorithm that utilizes some of the properties of this construction but with modifications that allows us to maintain a *similar* hopset dynamically.

▶ **Definition 6** (Bunches and clusters). *Let $G = (V, E, w)$ be a weighted, n-vertex graph, k be an integer such that $1 \leq k \leq \log\log n$ and $\rho > 0$. We define sets $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{k+1/\rho+1} = \emptyset$. Let $\nu = \frac{1}{2^k-1}$. Each set A_{i+1} is obtained by sampling each element from A_i with probability $q_i = \max(n^{-2^i \cdot \nu}, n^{-\rho})$.*

Fix $0 \leq i \leq k + 1/\rho + 1$ and for every vertex $u \in A_i \setminus A_{i+1}$, let $p(u) \in A_{i+1}$ be the node of A_{i+1}, which is closest to u, and let $d(u, A_{i+1}) := d(u, p(u))$ (assume $d(u, \emptyset) = \infty$). We call $p(u)$ the pivot *of u. We define a* bunch *of u to be a set $B(u) := \{v \in A_i : d(u, v) < d(u, A_{i+1})\}$. Also, we define a set $C(v)$, called the* cluster *of $v \in A_i \setminus A_{i+1}$, defined as $C(v) = \{u \in V : d(u, v) < d(u, A_{i+1})\}$.*

Note that if $v \in B(u)$ then $u \in C(v)$, but the converse does not necessarily hold. The way we define the bunches and clusters here follows [18], but differs slightly from the definitions in [31, 34], where each vertex has a separate bunch and cluster defined for each level i (and stores the union of these for all levels).

The clusters are *connected* in a sense that if a node $u \in C(v)$ then any node z on the shortest path between v and u is also in $C(v)$. This property is important for bounding the running time (as also noted in [31, 34]):

▷ Claim 7. Let $u \in C(v)$, and let $z \in V$ be on a shortest path between v and u. Then $z \in C(v)$.

Proof. Let $v \in A_i$. If $z \notin C(v)$ then by definition $d(z, A_{i+1}) \leq d(v, z)$. On the other hand, since z is on the shortest path between u and v: $d(u, A_{i+1}) \leq d(z, u) + d(z, A_{i+1}) \leq d(u, z) + d(z, v) = d(u, v)$, which contradicts the fact that $u \in C(v)$. ◁

The hopset is then obtained by adding an edge (u, v) for each $u \in A_i \setminus A_{i+1}$ and $v \in B(u) \cup \{p(u)\}$, and setting the weight of this edge to be $d(u, v)$. These distances can be computed by maintaining the clusters.

▶ **Lemma 8** ([18, 23]). *Let $G = (V, E, w)$ be a weighted, n-vertex graph, k be an integer such that $1 \leq k \leq \log\log n$ and $0 < \rho, 0 < \epsilon < 1$. Assume the sets A_i and bunches are defined as in Definition 6. Define a graph $H = (V, E_H, w_H)$, such that for each $u \in A_i \setminus A_{i+1}$ and $v \in B(u) \cup \{p(u)\}$, we have an edge $(u, w) \in E_H$ with weight $d_G(u, v)$. Then H is a $(\beta, 1 + \epsilon)$-hopset of size $O(n^{1+\frac{1}{2^k-1}})$, where $\beta = (O(\frac{k+1/\rho}{\epsilon})^{k+1/\rho+1}$.*

[8] In [18] two algorithms with different sampling probabilities are given, where one removes a factor of k in the size. This factor does not impact our overall running time, so we will use the simpler version.

For reference we sketch a proof of the hopset properties in the full version. Our main result is based on a new construction consisted of a hierarchy of hopsets. Our dynamic hopset requires a new *stretch* analysis as estimates on the shortest paths are obtained from different data structures, but the *size* analysis is basically the same.

While we are generally interested in a hopset that is not much denser than the input, as we will see the running time (both in static and dynamic settings) is mainly determined by the number of clusters a node belongs to, rather than the size of the hopset. Moreover, unlike an emulator, for computing the distances using a hopset, we also need to consider the edges in G, and a small hopbound is the key to efficiency rather than the sparsity.

The hopset of [18] has some structural similarities to the emulators of [35]. One main difference is that the sampling probabilities are adjusted (lower-bounded by $n^{-\rho}$) to allow for efficient construction of these hopsets in various models, at the cost of slightly weaker size/hopbound tradeoffs. This adjustment is also crucial for our efficient decremental algorithms. Inspired by the construction described, in Section 2.2 we describe a *new static* hopset algorithm, and later in Section 2.3 we adapt it to decremental settings.

2.2 New static hopset based on path doubling and scaling

As a warm-up, before moving to our new dynamic hopset construction, we provide a simple static hopset and explain why we expect to maintain such a structure more efficiently than the structure in Section 2.1 in *dynamic settings*. Our main contribution is to maintain a dynamic hopset *efficiently* using ideas in the simple algorithm described in this section.

At a high level, computing one of the main components of the hopset of Lemma 8 involves multiple single-source shortest paths computations. Maintaining single-source shortest paths is easy in the decremental setting, if we limit ourselves to paths of low length (or allow approximation). Namely, assuming integer edge weights, one can maintain single source shortest paths up to length d under edge deletions in total $O(md)$ time.

If we simply modified the construction of the hopset of Lemma 8, and computed shortest paths up to length d instead of shortest paths of unbounded length, we would obtain a d-restricted hopset. We describe this idea in more detail in Section 3, where we show that an adaptation of the techniques by [31] allows us to maintain a d-restricted hopset in deceremental settings, in total time $O(dmn^{\rho})$, for a parameter $0 < \rho < \frac{1}{2}$. However, for large d such a running time is prohibitive. In order to address this challenge, in this section we describe a static hopset, which can be computed using shortest path explorations up to only a polylogarithmic depth, yet can be used to approximate arbitrarily large distances. In the next sections we leverage this property to maintain a similar hopset in the decremental setting efficiently. This will require overcoming other obstacles, notably the fact that the dynamic shortest path problems that we need to solve are not decremental.

Path doubling. Assume that we are given a procedure $\textsc{Hopset}(G, \beta, d, \epsilon)$ that constructs a $(d, \beta, 1+\epsilon)$ hopset. In Section 3, we provide such an algorithm that uses only shortest path computation up to polylogarithmic depth. We argue that by applying the $\textsc{Hopset}(G, \beta, d, \epsilon)$ procedure *repeatedly* we can compute a full hopset, and in this process by utilizing the previously added hopset edges we can restrict our attention to short-hop paths only.

More formally, we construct a sequence of graphs $H_0, \ldots, H_{\log W}$, such that H_j is a hopset that handles pairs of nodes with distance in range $[2^{j-1}, 2^j)$, for $0 \leq j \leq \log W$. This implies that $\bigcup_{r=0}^{j} H_r$ is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. Note that for $0 \leq j \leq \log \beta$ we can set $H_j = \emptyset$, since G covers these scales. We would like to use $G \cup_{r=0}^{j-1} H_r$ to construct H_j based on the following observation that has been previously used in other (static) models (e.g. parallel hopsets of Cohen [13]).

Consider $u, v, d_G(u,v) \in [2^{j-1}, 2^j)$, and let π be the shortest path between u and v in G. Then π can be divided into three segments π_1, π_2 and π_3, where π_1 and π_3 have length at most 2^{j-1} and π_2 consists of a single edge. But we know there is a path in $G \cup_{r=0}^{j-1} H_r$ with at most β-hops that approximates each of π_1 and π_2. Hence for constructing H_j we can compute *approximate* shortest paths by restricting our attention to paths of consisting of at most $2\beta+1$ *hops* in $G \cup_{r=0}^{j-1} H_r$.

This idea, which we call path doubling, has been previously used in hopset constructions in distributed/parallel models (e.g. [13,17,18]), but to the best of our knowledge this is the first use of this approach in a dynamic setting. Applying this idea in parallel/distributed settings is relatively straight-forward, since having bounded hop paths already leads to efficient parallel shortest path explorations (e.g. by using Bellman-Ford). However, utilizing it efficiently in dynamic settings is more involved for several reasons: we have to simultaneously maintain the clusters (including their connectivity property), apply a scaling idea on the whole structure, and handle insertions in our hopset algorithm and its analysis.

But first we describe a *scaling idea* that at a high-level allows to go from *h-hop-bounded* explorations to h-depth bounded explorations on a scaled graph.

Scaling. We review a scaling algorithm that allows us to utilize the path doubling idea. Similar scaling techniques are used in dynamic settings [5,9,21] for single-source shortest paths, but as we will see, using the scaling idea in our setting is more involved since it has to be carefully combined with other components of our construction.

This idea can summarized in the following scaling scheme due to Klein and Subramanian [24], which, roughly speaking, says that finding shortest paths of length $\in [2^{j-1}, 2^j)$ and at most ℓ hops, can be (approximately) reduced to finding paths of length at most $O(\ell)$ in a graph with in integral weights. This is done by a rounding procedure that adds a small *additive* term of roughly $\frac{\epsilon_0 w(e)}{\ell}$ to each edge e. Then for a path of ℓ hops the overall stretch will be $(1+\epsilon_0)$.

▶ **Lemma 9** ([24]). *Let $G=(V,E,w)$ be a weighted undirected graph. Let $R \geq 0$ and $\ell \geq 1$ be integers and $\epsilon_0 > 0$. We define the* scaled graph *to be a graph $\textsc{Scale}(G, R, \epsilon_0, \ell) := (V, E, \hat{w})$, such that $\hat{w}(e) = \lceil \frac{w(e)}{\eta(R,\epsilon_0)} \rceil$, where $\eta(R,\epsilon_0) = \frac{\epsilon_0 R}{\ell}$. Then for any path π in G such that π has at most ℓ hops and weight $R \leq w(\pi) \leq 2R$ we have,*

- $\hat{w}(\pi) \leq \lceil 2\ell/\epsilon_0 \rceil$,
- $w(\pi) \leq \eta(R,\epsilon_0) \cdot \hat{w}(\pi) \leq (1+\epsilon_0) w(\pi)$.

Similar scaling ideas have been used in the h-SSSP algorithm for maintaining approximate shortest paths [5]. The algorithm maintains a collection of trees and to return a distance estimate, it finds the tree that best approximates a given distance. But we note that in utilizing the scaling techniques in our final dynamic hopset construction we cannot simply maintain a *disjoint set of* bounded hop shortest path trees. We need to maintain the whole structure of the hopset on the scaled graphs together: firstly, based on definition of bunches in Lemma 8, nodes keep on leaving and joining clusters, so we cannot simply maintain a set of shortest trees from a fixed set of roots. We need to maintain the *connectivity* of clusters as described in Section 2.1 at the same time as maintaining the shortest path trees. Additionally, while we are maintaining distances over the set of clusters we also need to handle insertions introduced by smaller scales.

To maintain these efficiently, we need to apply the scaling to the whole structure, including the hopset edges added so far. But when we utilize the smaller scale hopset edges (for applying path doubling) insertions or distance decreases are introduced. As we will see, handling insertions at the same time the clusters (and the corresponding distances) are updated makes the stretch/hopbound analysis more involved.

Next we combine the scaling with the path doubling techniques. The path doubling property states that we can restrict our attention to $(2\beta+1)$-hop limited shortest path computation, and the scaling idea ensures that such $(2\beta+1)$-*hop* bounded paths in $G \cup \bigcup_{r=0}^{j-1} H_r$ correspond to paths bounded *in depth* by $d = \lceil \frac{2\ell}{\epsilon_0} \rceil = O(\frac{\beta}{\epsilon_0})$ in the scaled graph $G_{scaled} = \textsc{Scale}(G \cup \bigcup_{r=0}^{j-1} H_r, 2^j, \epsilon_0, 2\beta+1)$. Informally, this mean it is enough to construct shortest path trees up to depth ℓ on the scaled graphs in our hopset construction.

We can now summarize our new static hopset construction in Algorithm 1. Similarly to Scale, for a graph $G = (V, E, w)$ we define $\textsc{Unscale}(G, R, \epsilon, \ell)$ to be a graph (V, E, w'), where for each $e \in E$, $w'(e) = \eta(R, \epsilon) \cdot w(e)$. In static settings, the procedure $\textsc{Hopset}(G, \beta, d, \epsilon)$ for constructing a d-restricted hopset can be performed by running a $(\beta, 1+\epsilon)$-hopset construction algorithm in which the shortest path explorations are restricted to depth ℓ. In Section 3 we describe a decremental algorithm for this procedure, and describe how it leads to a $(d, \beta, 1+\epsilon)$-restricted hopset. Note that we can set $\beta = \text{poly} \log n$, and so the shortest path explorations can be bounded by a polylogairthmic value.

Algorithm 1 Simple static hopset.

for $j = 1$ *to* $\lceil \log W \rceil$ **do**
  $G_{scaled} := \textsc{Scale}(G \cup \bigcup_{r=0}^{j-1} H_r, 2^j, \epsilon_0, 2\beta+1)$
  $\hat{H} := \textsc{Hopset}(G_{scaled}, \beta, \lceil \frac{2(2\beta+1)}{\epsilon_0} \rceil, \epsilon)$
  $H_j := \textsc{Unscale}(\hat{H}, 2^j, \epsilon_0, 2\beta+1))$
  $H := H \cup H_j$

It is not hard to see that in such a static construction, three different approximation factors are combined in each scale: a $(1+\epsilon)$-stretch due to the Hopset procedure, a $(1+\epsilon_1)$-factor from the restricted hopset, and a $(1+\epsilon_0)$-factor due to scaling. This is summarized in the following lemma.

▶ **Lemma 10.** *Let G be a graph and H be a $(d, \beta, 1+\epsilon_1)$ hopset of G. Let $G_{scaled} = \textsc{Scale}(G \cup H, d, \epsilon_0, 2\beta+1)$ and let $H' = \textsc{Unscale}(\textsc{Hopset}(G_{scaled}, d, \lceil \frac{2(2\beta+1)}{\epsilon_0} \rceil, \epsilon_2))$. Then $H \cup H'$ is a $(2d, \beta, (1+\epsilon)(1+\epsilon_1)(1+\epsilon_0))$ hopset of G.*

Obtaining such a guarantee in dynamic settings is going to be more involved, since we also need to handle insertions, and at the same time ensure that the update time remains small. Moreover the stretch analysis will require combining estimates obtained by different procedures.

2.3 Near-Optimal Decremental Hopsets

In this section we describe how we can overcome the challenges of the dynamic settings in order to maintain a decremental hopset in near-optimal update-time.

The first step of our algorithm is constructing a d-restricted version of the hopset described in Section 2.1. As discussed, for this we can use the techniques by [31] to maintain a $(d, \beta, 1+\epsilon)$-hopset in $\tilde{O}(dmn^{\rho})$ total update time, where $0 < \rho < \frac{1}{2}$. Now in order to remove the time dependence on d, we use the path doubling and scaling ideas described as follows: we maintain this data structure on a sequence of scaled graphs simultaneously, and argue that this data structure gives us a hopset on G after *unscaling* the edge weights.

Sequence of restricted hopsets. Similiar to Section 3.1, our decremtnal algorithm maintains the sequence of graphs $H_0, \ldots, H_{\log W}$, where for each $0 \leq j \leq \log W$, $\bigcup_{r=0}^{j} H_r$ is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. For *each scale* we show the following:

▶ **Lemma 11.** *Consider a graph $G = (V, E, w)$ subject to edge deletions, and parameters $0 < \epsilon < 1, \rho < \frac{1}{2}$. Assume that we have maintained $\bar{H}_j := H_1, ..., H_j$, which is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. Then given the sequence of changes to G and $\bar{H}_j$, we can maintain a graph H_{j+1}, such that $\bar{H}_j \cup H_{j+1}$ is a $(2^{j+1}, \beta, (1+\epsilon)^{j+1})$-hopset of G. This restricted hopset can be maintained in $\tilde{O}((m+\Delta)n^\rho \cdot \frac{\beta}{\epsilon})$ total time, where m is the initial size of G, and Δ is the number of edges inserted to $\bar{H}_j$ over all updates, $\beta = (\frac{1}{\epsilon \cdot \rho})^{O(1/\rho)}$.*

Note that the lemma does not hold for *any* restricted hopset, and in dynamic settings we need to use special properties of our construction to prove this.

To prove this lemma we use the techniques of [31] to maintain the clusters. For obtaining near-optimal update time, we combine this algorithm with the path doubling and scaling ideas described earlier. However, in order to utilize these ideas, we need to deal with the fact that inserting hopset edges from smaller scales introduces *insertions*.

Handling insertions. In addition to maintaining clusters and distances decrementally, in our final construction we need to handle edge *insertions*. This is because we run it on a graph $G \cup \bigcup_{r=0}^{j-1} H_r$ (after applying scaling of Lemma 9). While edges of G can only be deleted, new edges are added to the H that we need to take into account for obtaining faster algorithms.

At a high-level, the algorithm of Roditty and Zwick [31] decrementally maintains a collection of single-source shortest path trees (up to a bounded depth) using the Even-Shiloach algorithm (ES-tree) [33] at the same time as maintaining a clustering. To handle edge insertions, we modify the algorithm to use the *monotone* ES-tree idea proposed by [21,22].

The goal of a monotone ES-tree is to support edge insertions in a limited way. Namely, whenever an edge (u, v) is inserted and the insertion of the edge causes a distance decrease in the tree, we do *not* update the currently maintained distance estimates. Still the inserted edge may impact the distance estimates in later stages by preventing some estimates from increasing after further deletions.

While it is easy to see that this change keeps the running time roughly the same as in the decremental setting, analyzing the correctness is a nontrivial challenge. This is because the existing analyses of a monotone ES-tree work under specific structural assumptions and do not generalize to any construction. Specifically, while [21] analyzed the stretch incurred by running monotone ES-trees on a hopset, the proof relied on the properties of the specific hopset used in their algorithm. Since the hopset we use is quite different, we need a different analysis, which combines the static hopset analysis, with the ideas used in [21], and also take into account the stretch incurred due to the fact that the restricted hopsets are maintained on the scaled graphs. Note also that our main hopset is not a simply a decremental maintenance of hopsets of [16], as our estimates are obtained from a *sequence of hopsets* and insertions in one scale introduce insertions in the next scale. This is why we need a new argument and cannot simply rely on arguments in [21] and [16].

Putting it together. We now go back to the setting of Lemma 11, and use a procedure similar to Algorithm 1. Given a 2^j-restricted hopset $\bar{H}_j = H_1 \cup ... \cup H_j$ for distances up to 2^j, we can now construct a graph G^j by applying the scaling of Lemma 9 to $G \cup \bar{H}_j$ and setting $R = 2^j$, $\ell = 2\beta + 1$. Then we can efficiently maintain an ℓ-restricted hopset on G^j. Then by Lemma 11 we can use this to update H_{j+1}. Importantly, ℓ is independent of R, and

thus we can eliminate the factor R to get $\tilde{O}(\beta mn^{\rho})$ total update time. Our final algorithm is a hierarchical construction that maintains the restricted hopsets on scaled graphs and the original graph simultaneously.

Stretch and hopbound analysis. As discussed, applying the path-doubling idea to the hopset analysis is straightforward in static settings (and can be to some extent separated from the rest of the analysis) as is the case in [18]. However in our adapted decremental hopset algorithm this idea needs to be combined with the properties of the monotone ES tree idea and the fact that distance estimate are obtained from a sequence of hopsets on the scaled graph. In particular, in our stretch analysis we need to divide paths into smaller segments, such that the length of some segments is obtained from smaller iterations i, and the length of some segments are obtained from this combination of monotone ES tree estimates based on path doubling and scaling. We need a careful analysis to show that the stretch obtained from these different techniques combine nicely, which is based on a threefold inductive analysis:

1. An induction on i, the iterations of the base hopset, which controls the sampling rate and the resulting size and hopbound tradeoffs.
2. An induction on the scale j, which allows us to cover all ranges of distances $[2^j, 2^{j+1}]$ by maintaining distances in the appropriate scaled graphs.
3. An induction on time t that allows us to handle insertions by using the estimates from previous updates in order to keep the distances monotone.

The overall stretch argument needs to deal with several error factors in *addition to* the base hopset stretch. First, the error incurred by using hopsets for smaller scales, which we deal with by maintaining our hopsets by setting $\epsilon' = \frac{\epsilon}{\log n}$. This introduces polylogarithmic factors in the hopbound. The second type of error comes from the fact that the restricted hopsets are maintained for scaled graphs, which implies the clusters are only approximately maintained on the original graph. This can also be resolved by further adjusting ϵ'. Finally, since we use an idea similar to the monotone ES tree of [21, 22], we may set the level of nodes in each tree is to be larger than what it would be in a static hopset. But we argue that the specific types of insertions in our algorithm will still preserve the stretch. At a high-level this is because in case of a decrease we use an estimate from time $t-1$, which we can show inductively has the desired stretch. We note that while the monotone ES tree is widely used, we always need a different construction-specific analysis to prove the correctness.

Technical differences with previous decremental hopsets. We note that while the use of monotone ES tree and the structure of the clusters in our construction are similar to [21], our algorithm has several important technical differences. First, our hopset algorithm is based on different base hopset with a polylogarithmic hopbound, which as noted is crucial for obtaining near-optimal bounds in most of our applications. Additionally, we use a different approach to maintain the hopset efficiently by using path doubling and maintaining restricted hopsets on a sequence of *scaled graphs*. Among other things, in [21] a notion of *approximate ball* is used that is rather more lossy with respect to the hopbound/stretch tradeoffs. By maintaining restricted hopsets on scaled graphs, we are also effectively preserving approximate clusters/bunches with respect to the original graph, but as explained earlier, the error accumulation combines nicely with the path-doubling idea. Moreover, [21] uses an edge sampling idea to bound the update time, which we can avoid by utilizing the sampling probability adjustments in [18], and the ideas in [31]. Finally, our algorithm is based on maintaining the clusters up to a low hop, whereas they directly maintain bunches/balls.

2.4 Applications in Decremental Shortest Paths

Our algorithms for maintaining approximate distances under edge deletions are as follows. First, we maintain a $(\beta, 1+\epsilon)$-hopset. Then, we use the hopset and Lemma 9 to reduce the problem to the problem of approximately maintaining short distances from a single source. For our application in MSSP and APSP the best update time is obtained by setting the hopbound to be polylgarithmic whereas for SSSP the best choice is $\beta = 2^{\tilde{O}(\sqrt{\log n})}$. Using this idea for SSSP and MSSP mainly involves using the monotone ES tree ideas described earlier. Maintaining the APSP distance oracle is slightly more involved but uses the same techniques as in our restricted hopset algorithm. This algorithm is based on maintaining Thorup-Zwick distance oracle [34] more efficiently. At a high-level, we maintain *both* a $(\beta, 1+\epsilon)$-hopset and Thorup-Zwick distance oracle simultaneously, and balance out the time required for these two algorithms. The hopset is used to improve the time required for maintaining the distance oracle from $O(mn)$ (as shown in [31]) to $O(\beta mn^{1/k})$, but with a slightly weaker stretch of $(2k-1)(1+\epsilon)$. Querying distances is then the same as in the static algorithm of [34], and takes $O(k)$ time. In the full version of this paper, we explain how our hopset can be used for applications in approximate shortest paths and distance sketches.

3 Decremental Hopset

In this section we describe two decremental hopset algorithms with different tradeoffs. The starting-point of our constructions are the static hopsets described in Section 2.1. But in order to get an efficient dynamic algorithm, we need to modify this construction in several ways. First we explain how we can adapt ideas by Roditty-Zwick [31] to obtain an algorithm for computing a d-restricted hopset. The total running time of this algorithm is $O(dmn^{\rho})$ (where $\rho < 1$ is a constant). While existentially this construction matches the state-of-the-art static hopsets with respect to size and hopbound tradeoffs, the update time is undesirable for large values of d, and thus in Section 3.1 we explain how we can remove this dependence from the running time at the cost of a slightly worst hopbound guarantee.

Maintaining a restricted hopset. We start by adapting the decremental algorithm by [31] that maintains the Thorup-Zwick distance oracles [34] with stretch $(2k-1)$ for pairs of nodes within distance d in $\tilde{O}(dmn^{1/k})$ total time, but we use it to obtain a d-restricted hopset. In particular, using ideas in [31], and by restricting the shortest path trees up to depth d, we can maintain a variant of the hopset defined in Lemma 8 in which the hopset guarantee only holds for nodes within distance d. In the full version, we describe how we adapt the algorithm of [31] to our settings to prove the following theorem.

▶ **Theorem 12.** *Fix $\epsilon > 0, k \geq 2$ and $\rho \leq 1$. Given a graph $G = (V, E, w)$ with integer and polynomial weights, subject to edge deletions we can maintain a $(d, \beta, 1+\epsilon)$-hopset, with $\beta = O\left(\left(\frac{1}{\epsilon} \cdot (k + 1/\rho)\right)^{k+1/\rho+1}\right)$ in $O(d(m + n^{1+\frac{1}{2^k-1}})n^{\rho})$ total time. The size/hopbound guarantee holds with high probability against an oblivious adversary.*

This algorithm has a large update time for d-restricted hopsets, when d is large. Next we show how we can eliminate this update time dependence on d, which is the main technical component of this work.

3.1 Decremental hopsets with improved update time

Next we provide a new hopset algorithm that is based on maintaining these restricted hopsets on a sequence of scaled graphs, and show how this improves the update time, in exchange for a small (polylogarithmic) loss in the hopbound.

Recall that our algorithm maintains a sequence of graphs $H_0, \ldots, H_{\log W}$, where for each $1 \leq j \leq \log W$, $H_0 \cup \ldots \cup H_j$ is a 2^j-restricted hopset of G. Instead of computing each H_j separately, we use $G \cup \bigcup_{r=0}^{j-1} H_r$ to construct H_j. The first technical challenge is making the running time independent of the distance bound 2^j, which is what we would get by directly using the algorithm of [31]. We observe that at the cost of some small approximation errors, any path of length $\in [2^{j-1}, 2^j)$ in G can be approximated by a path of at most $2\beta+1$ hops in $G \cup \bigcup_{r=0}^{j-1} H_r$. This relies on having the 2^j-restricted hopset $\bar{H}_j$, which allows us to maintain the hopset $\bar{H}_{j+1}$.

Second, while G is undergoing deletions, H_j may be undergoing edge *insertions* incurred by restricted hopset edges added for smaller scales, which we discuss next. All the missing details in this section can be found in the full-version of this paper.

Handling edge insertions. We handle edge insertions by combining of the monotone ES-tree algorithm [21] (and further used in the hopset construction of [22]). We summarize this idea and the relevant properties in the full version. As stated earlier, the algorithm itself is a simple extension of the Even-Schiloach tree. At a high-level we maintain an ES tree for each cluster and when an insertion causes the level of a node in an ES tree to decrease, we ignore the insertion and keep the same level. The more challenging aspect of using the monotone ES tree idea is proving the correctness (stretch), as this does not extend to all types of insertions but only for insertion with certain inductive structural properties. That is why even though this idea is widely used, we always need a construction-specific proof of correctness. In Theorem 14 we prove that specifically for the insertions in our final hopset algorithm the use of monotone ES tree does not violate our stretch argument.

Path doubling and scaling. We first state the path doubling idea more formally for a *static hopset* in the following lemma. However for utilizing this idea dynamically we need to combine it with other structural properties of our hopsets and the two algorithms described above.

▶ **Lemma 13.** *Given a graph $G = (V, E)$, $0 < \epsilon_1 < 1$, the set of $(\beta, 1+\epsilon_1)$-hopsets $H_r, 0 \leq r < j$ for each distance scale $(2^r, 2^{r+1}]$, provides a $(1+\epsilon_1)$-approximate distance for any pair $x, y \in V$, where $d(x, y) \leq 2^{j+1}$ using paths with at most $2\beta+1$ hops.*

This implies that it is enough to compute $(2\beta+1)$-hop limited distances in restricted hopsets for *each* scale. For using this idea in dynamic settings we have to deal with some technicalities. We should show that we can combine the rounding with the modification needed for handling insertions.

A hierarchy of restricted hopsets. We define a scaled graph using Lemma 9 as follows: $G^j := \textsc{Scale}(G \cup \bigcup_{r=0}^{j} H_r, 2^j, \epsilon_2, 2\beta+1)$. Here we set $R = 2^j, \ell = 2\beta+1$, and ϵ_2 is a parameter that we tune later. We first describe the operations performed on this scaled graph. We then explain how we can put things together for all scales to get the desired guarantees. The key insight for scaling $G \cup \bigcup_{r=0}^{j} H_r, 2^j$ is that we can obtain H_{j+1} by computing an $O(\ell)$-restricted hopset of G^j (using the algorithm of Lemma 11) and scaling back the weights of the hopset edges.

In addition to the graph G undergoing deletions, our decremental algorithm maintains the following data structures for each $1 \leq j \leq \log W$:

- The set $\bar{H}_j = \bigcup_{r=0}^{j} H_r$, union of all hopset edges for distance scales up to $[2^j, 2^{j+1}]$.
- The scaled graphs $G^1, ..., G^j$.
- Data structure obtained by constructing an $O(\beta/\epsilon_2)$-restricted hopset on G^j using Theorem 12 for the appropriate parameter $\epsilon_2 < 1$. We denote this data structure by D_j.

We update the data structures described as follows: we maintain d-restricted hopsets for $d = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ starting on $j = 0, ..., \log W$ in increasing order of j to compute hopset edges H_j. After processing all the changes in scaled graph G^j, we add the inserted edges to G^{j+1}. Then we process the changes in G^{j+1} by computing a d-restricted hopset again and repeat until all distance scales of covered. As described, when the distances increase a node may join a new cluster which will lead to a set of insertions in H and in turn insertions in a sequence of graphs G^j. Note that we need to update both the restricted hopsets on the scaled graphs (denoted by D_j) and the hopset H_j for G obtained by scaling back the distances using Lemma 9. A pseudocode of this algorithm and its running time analysis can be found in the full version.

Hopset stretch and hopbound. We next show the stretch and hopbound of the hopset algorithm described for a single-scale by combining properties of the monotone ES-tree algorithm with the static hopset argument and the rounding framework.

▶ **Theorem 14.** *Given a graph $G = (V, E)$, and $0 \leq \epsilon_2 < 1$, assume that we have maintained a $(2^j, \beta, (1+\epsilon_j))$-restricted hopset $\bar{H}_j$, and let H_{j+1} be the hopset obtained by running the above algorithm on $G \cup \bar{H}_j$. Fix $0 < \delta \leq \frac{1}{8(k+1/\rho+1)}$, and consider a pair $x, y \in V$ where $d_{t,G}(x,y) \in [2^j, 2^{j+1}]$. Then for $0 \leq i \leq k + 1/\rho + 1$, either of the following conditions holds:*

1. $d^{((3/\delta)^i)}_{G \cup \bar{H}_{j+1}}(x,y) \leq (1+8\delta i)(1+\epsilon_j)(1+\epsilon_2) d_{t,G}(x,y)$ *or,*
2. *There exists $z \in A_{i+1}$ such that,*

$$d^{((3/\delta)^i)}_{G \cup \bar{H}_{j+1}}(x,z) \leq 2(1+\epsilon_j)(1+\epsilon_2) d_{t,G}(x,y).$$

Moreover, by maintaining a monotone ES tree on G^{j+1} up to depth $\lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$, and applying the rounding in Lemma 9, we can maintain $(1+\epsilon_{j+1})$-approximate single-source distances up to distance 2^{j+2} from a fixed source s on G, where $1+\epsilon_{j+1} = (1+\epsilon_j)(1+\epsilon_2)^2(1+\epsilon)$ and $\beta = (3/\delta)^{k+1/\rho+1}$.

Proof. The stretch argument is based on a threefold induction on i, j-th scale, and time t. By fixing i, j, t, and a source node s, we show that there is a $(1+\epsilon_j)$-stretch path between s and any other node with β hops (or if we are using previous scale $2\beta+1$-hops) such that each segment of this path has the desired stretch based on the inductive claim on one of these three factors. At a high level induction on i and j follows from static properties of our hopset. To show that bounded depth monotone ES tree maintains the approximate distances, we note that any segment of the path undergoing an insertion consistent of a single shortcut and the weight on such an edge is a distance estimate between its endpoints.

It is easy to see that we never underestimate distances. Roughly speaking, we either obtain an estimate from rounding estimates obtained from smaller scales, which is an upper bound on the original estimate, or we ignore a distance decrease.

We use a double induction on i and time t, and also rely on distance computed up to scaled graph G^j. First, using these distance estimates for smaller scales, we argue that when we add an edge to $\bar{H}_{j+1}$ it has the desired stretch. Let $L_{t,j}(u,v)$ denote the level of node v

in the tree rooted at u after running the monotone ES tree up to depth $D = \lceil \frac{2(2\beta+1)}{\epsilon} \rceil$ on graph G^j. This proof is based on a cyclic argument: assuming we have correctly maintained distances up to a given scale using our hopset, we show how we can compute the distances for the next scale. In particular, we first assume that based on the theorem conditions we are given $\bar{H}_j$ and have maintained all the clusters and the corresponding distances in $G^1, ..., G^j$ with stretch $1 + \epsilon_j$. This lets us analyze H_{j+1}. Then to complete the argument, we show how given the hopsets of scale $[2^j, 2^{j+1}]$, we can compute approximate SSSP distances for the next scale based on the monotone ES tree on G^{j+1}.

First, in the following claim, we observe that the edge weights inserted in the latest scale have the desired stretch by using our inductive assumption that all the shortest path trees on each cluster on $G^1, ..., G^j$ are approximately maintained. We use such distance to add edges in each cluster to construct H_{j+1}, and we observe the following about the weights on these edges:

▶ **Observation 15.** *Let $v \in B(u)$ such that $d_{t,G}(u, v) \leq 2^{j+1}$. Consider an edge (u, v) added to H_{j+1} after running the algorithm on $G^1, ..., G^j$ for $D = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ rooted at node v. Let $w_{j+1}(u, v) := \min_{r=1}^{j} \eta(2^r, \epsilon_2) L_r(u, v)$, that is the unscaled edge weight. Then we have $d_{t,G}(u, v) \leq w_{j+1}(u, v) \leq (1 + \epsilon_j)(1 + \epsilon_2) d_{t,G}(u, v)$.*

This claim implies that the weights of hopset edges assigned by the algorithm correspond to approximate distance of their endpoints. Let $d_{t,j}(x, y) := \min_{r=1}^{j} \eta(2^r, \epsilon_2) L_{t,j}(x, y)$ which would be the estimate we obtain by for distance between x and y after scaling back distances on G^j. In other words this is the hop-bounded distance after running monotone ES tree on G^j and scaling up the weights.

For any time t and the base case of $i = 0$, we have three cases. If $y \in B(x)$ then edge (x, y) is in the hopset H_{j+1}, and by Observation 15 the weight assigned to this edge is at most $(1 + \epsilon_j)(1 + \epsilon_2) d_{t,G}(x, y)$. In this case the first condition of the theorem holds. Otherwise if $x \in A_1$, then $z = x$ trivially satisfies the second condition. Otherwise we have $x \in A_0 / A_1$, and by setting $z = p(x)$ we know that there is an edge $(x, z) \in \bar{H}_j$ such that $d_{t,j}(x, z) \leq (1 + \epsilon_2) d_{G \cup \bar{H}_j}(x, y)$ (by definition of $p(x)$ and using the same argument as above). Hence the second condition holds.

By inductive hypothesis assume the claim holds for i. Consider the shortest path $\pi(x, y)$ between x and y. We divide this path into $1/\delta$ segments of length at most $\delta d_{t,G}(x, y)$ and denote the a-th segment by $[u_a, v_a]$, where u_a is the node closest to x (first node of distance at least $a\delta d_{t,G}(x, y)$) and v_a is the node furthest to x on this segment (of distance at most $(a + 1)\delta d_{t,G}(x, y)$).

We then use the induction hypothesis on each segment. First consider the case where for all the segments the first condition holds for i, then there is a path of $(3/\delta)^i (1/\delta) \leq (3/\delta)^{i+1}$ hops consisted of the hopbounded path on each segment. We can show that this path satisfies the first condition for $i + 1$. In other words,

$$d^{((3/\delta)^{i+1})}_{t,G \cup \bar{H}_{j+1}}(x, y) \leq \sum_{a=1}^{1/\delta} d^{((3/\delta)^i)}_{t,G \cup \bar{H}_{j+1}}(u_a, v_a) + d^{(1)}_{t,G}(v_a, u_{a+1}) \leq (1 + 8\delta i)(1 + \epsilon_j)(1 + \epsilon_2) d_{t,G}(x, y)$$

Next, assume that there are at least two segments for which the first condition does not hold for i. Otherwise, if there is only one such segment a similar but simpler argument can be used. Let $[u_l, v_l]$ be the first such segment (i.e. the segment closest to x, where u_l is the first and v_l is the last node on the segment), and let $[u_r, v_r]$ be the last such segment.

First by inductive hypothesis and since we are in the case that the second condition holds for segments $[u_l, z_l]$ and $[u_r, v_r]$, we have,

- $d^{((3/\delta)^i)}_{t,G\cup\bar{H}_{j+1}}(u_l, z_l) \leq 2(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(u_l, v_l)$, *and,*
- $d^{((3/\delta)^i)}_{t,G\cup\bar{H}_{j+1}}(v_r, z_r) \leq 2(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(u_r, v_r)$

Again, we consider two cases. First, in case $z_r \in B(z_l)$ (or $z_l \in C(z_r)$), we have added a single hopset edge $(z_r, z_l) \in \bar{H}_{j+1}$. Note that $d_{t,G}(z_r, z_l) \leq 2^{j+1}$, since $d_{t,G}(z_r, z_l) \leq d_{t,G}(x, y) \leq 2^{j+1}$. Hence by Observation 15 the weight we assign to (z_r, z_l) is at most $(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(z_r, z_l)$.

On the other hand, by triangle inequality, and the above inequalities (which are based on the induction hypothesis) we get,

$$d^{(1)}_{\bar{H}_{j+1}}(z_l, z_r) \leq (1+\epsilon_2)(1+\epsilon_j)d_G(z_l, z_r) \tag{1}$$

$$\leq (1+\epsilon_2)(1+\epsilon_j)[d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l, z_l) + d_G(u_l, v_r) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(z_r, v_r)] \tag{2}$$

By applying the inductive hypothesis on segments before $[u_l, v_l]$, and after $[u_r, v_r]$, we have a path with at most $(3/\delta)^i$ for each of these segments, satisfying the first condition for the endpoints of the segment. Also, we have a $2(3/\delta)^i + 1$-hop path going through u_l, z_l, z_r, v_r that satisfies the first condition for u_l, v_r.

Putting all of these together, we argue that there is a path of hopbound $(3/\delta)^{i+1}$ satisfying the first condition. In particular, we have (the subscript t is dropped in the following),

$$d^{(3/\delta)^{(i+1)}}_{G\cup\bar{H}_{j+1}}(x, y) \leq \sum_{a=1}^{l-1}[d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_a, v_a) + d^{(1)}_G(v_a, u_{a+1})] + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l, z_l) \tag{3}$$

$$+ d^{(1)}_{\bar{H}_{j+1}}(z_l, z_r) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(z_r, v_r) + d^{(1)}_G(v_r, u_{r+1}) \tag{4}$$

$$+ \sum_{a=r+1}^{1/\delta}[d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_a, v_a) + d^{(1)}_G(v_a, u_{a+1})] \tag{5}$$

$$\leq (1+8\delta i)(1+\epsilon_j)(1+\epsilon_2)[d_G(x, u_l) + d_G(v_r, y)] + d_G(u_l, v_r) \tag{6}$$

$$+ (1+\epsilon_2)(1+\epsilon_j)[2d_G(u_l, z_l) + 2d_G(z_r, v_r)] \tag{7}$$

$$\leq (1+\epsilon_2)(1+\epsilon_j)[8\delta d_G(x, y) + (1+8\delta i)d_G(x, y)] \tag{8}$$

$$\leq (1+8\delta(i+1))(1+\epsilon_2)(1+\epsilon_j)d_G(x, y) \tag{9}$$

In the first inequality we used the induction on i for each segment, and triangle inequality. In the second inequality we are using the fact that nodes u_j, v_j for all j are on the shortest path between x and y in G, and we are replacing $d^{(1)}_{\bar{H}_{j+1}}(z_l, z_r)$ with inequality 2. In line 8 we used the fact that the length of each segment is at most $\delta \cdot d_G(x, y)$. Hence we have shown that the first condition in the lemma statement holds.

Finally, consider the case where $z_r \notin B(z_l)$. If $z_l \notin A_{i+2}$, we consider $z = p(z_l)$, where $z_l \in A_{i+2}$. We now claim that this choice of z satisfies the second lemma condition.

We have added the edge (z_l, z) to the hopset. Since $z_r \notin B(z_l)$, we have $d_{t-1,G}(z_l, p(z_l)) \leq d_{t-1,G}(z_l, z_r) \leq d_{t,G}(x, y) \leq 2^{j+1}$. Therefore we can use Observation 15 on the edge $(z_l, p(z_l))$.

$$d^{(3/\delta)^{(i+1)}}_{G\cup\bar{H}_{j+1}}(x,y) \leq \sum_{a=1}^{l-1}[d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_a,v_a) + d^{(1)}_{G}(v_a,u_{a+1})] \tag{10}$$

$$+ d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) + (1+\epsilon_2)(1+\epsilon_j)d^{(1)}_{\bar{H}_{j+1}}(z_l,z) \tag{11}$$

$$\leq (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d_G(x,u_l) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) \tag{12}$$

$$+ (1+\epsilon_2)(1+\epsilon_j)d_{\bar{H}_{j+1}}(z_l,z_r) \tag{13}$$

$$\leq (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d_G(x,u_l) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) \tag{14}$$

$$+ (1+\epsilon_2)(1+\epsilon_j)[2d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(z_l,u_l) + d_G(u_l,v_r) + d^{(3/\delta)^i}_{G\cup\bar{H}_{j+1}}(v_r,z_r)] \tag{15}$$

$$\leq (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(x,v_r) + 6\delta(1+\epsilon_j)d_G(x,y) \tag{16}$$

$$\leq 2(1+\epsilon_2)(1+\epsilon_j)d_G(x,y) \tag{17}$$

In the last inequality we used the fact that we set $\delta < \frac{1}{8(k+1/\rho+1)}$ and thus $8\delta i < 1$. The only remaining case is when $z_\ell \in A_{i+2}$, in which case a similar reasoning follows by setting $z = z_l$.

Next, we need to prove that after adding hopset edges H_{j+1} we can maintain approximate single-source shortest path distances from a given source s to conclude the proof of this theorem. For this we need to use scaling again, and by Lemma 9 an additional $(1+\epsilon_2)$-factor will be added to the stretch. This enables us to show that Observation 15 can be used for the next scale, i.e. that we can set the weights for the next scale by maintaining the clusters and $(1+\epsilon_{j+1})$ approximate distance rooted at a source s when we have $d(s,v) \in [2^{j+1}, 2^{j+2}]$, and hence close the inductive cycle in the argument. This argument uses a similar type of case-by-case analysis as the above argument combined with path-doubling. We omit this argument here due to space limitations. The complete proof can be found in the full version of this paper. ◀

Theorem 14 allows us to hierarchically use the restricted hopsets for smaller scales to compute the distance for larger scales, that is in turn used to update the hopset edges in the larger scales. Finally, for getting the final stretch and hopbound we set the parameters $\epsilon' = \frac{\epsilon}{6\log W}, \epsilon_2 = \epsilon'$ (error incurred by rounding), and $\delta = \frac{\epsilon}{8(k+1/\rho+1)}$ (details can be found in the full version). Putting it all together we get the following theorem:

▶ **Theorem 16.** *The total update time in each scaled graph G^j, $1 \leq j \leq \log W$, over all deletions is $\tilde{O}((\beta/\epsilon')(n^{1+\frac{1}{2^k-1}} + m)n^\rho)$, and hence the total update time for maintaining $(\beta, 1+\epsilon)$-hopset with hopbound $\beta = O(\frac{\log W}{\epsilon} \cdot (k+1/\rho))^{k+1/\rho+1}$ is $\tilde{O}(\frac{\beta}{\epsilon} \cdot mn^\rho \cdot \log W)$.*

4 Applications in Decremental Approximate Shortest Path

In the full version of this paper, we use our hopsets to maintain approximate shortest paths and distance sketches. At a high level, we maintain a $(\beta, 1+\epsilon)$-hopset using the appropriate parameter settings in Theorem 16. The applications in $(1+\epsilon)$-SSSP and $(1+\epsilon)$-MSSP are straightforward extensions of Theorem 14.

In our approximate APSP data structure we simultaneously maintain a $(\beta, 1+\epsilon)$-hopset for by setting β to be polylogarithmic and a Thorup-Zwick distance oracle [34]. Our algorithm for maintaining distance oracles of [34] is similar to the restricted hopset algorithm, combined

with the rounding procedure in Lemma 9 that allows us to maintain clusters up to $O(\beta/\epsilon)$ hops. One main difference between these algorithms is in the information/distances stored and the fact that the sampling probabilities stay fixed in case of distance oracles. In order to maintain $(2k-1)(1+\epsilon)$-approximate APSP, we set the parameters ρ and k in such a way that updating the hopset and the distance oracle are roughly the same. By maintaining the distance oracle, querying distances is the same as in the static algorithm of [34], and takes $O(k)$ time, which is constant when k is constant.

References

1. Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. *SIAM Journal on Computing*, 47(6):2203–2236, 2018.
2. Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 322–335, 2020.
3. Ruben Becker, Sebastian Forster, Andreas Karrenbauer, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. *SIAM Journal on Computing*, 50(3):815–856, 2021.
4. Uri Ben-Levy and Merav Parter. New (α, β) spanners and hopsets. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1695–1714. SIAM, 2020.
5. Aaron Bernstein. Fully dynamic (2+ ε) approximate all-pairs shortest paths with fast query and close to linear update time. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 693–702. IEEE, 2009.
6. Aaron Bernstein. Deterministic partially dynamic single source shortest paths in weighted graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, page 44. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2017.
7. Aaron Bernstein and Shiri Chechik. Deterministic partially dynamic single source shortest paths for sparse graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 453–469. SIAM, 2017.
8. Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time. In *62 Annual IEEE Symposium on Foundatios of Computer Science (FOCS 2022)*, 2021.
9. 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*, pages 1355–1365. SIAM, 2011.
10. Keren Censor-Hillel, Michal Dory, Janne H Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 74–83, 2019.
11. Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 654–663, 2014.
12. Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 170–181. IEEE, 2018.
13. Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 2000.
14. Michael Dinitz and Yasamin Nazari. Massively parallel approximate distance sketches. *OPODIS*, 2019.
15. Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Almost shortest paths and pram distance oracles in weighted graphs. *arXiv preprint*, 2019. `arXiv:1907.11422`.

16 Michael Elkin and Ofer Neiman. Near-optimal distributed routing with low memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2018.

17 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 2019.

18 Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in rnc. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 333–341, 2019.

19 Michael Elkin and Ofer Neiman. Near-additive spanners and near-exact hopsets, a unified view. *arXiv preprint*, 2020. `arXiv:2001.07477`.

20 Maximilian Probst Gutenberg and Christian Wulff-Nilsen. Deterministic algorithms for decremental approximate shortest paths: Faster and simpler. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2522–2541. SIAM, 2020.

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

22 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the o(mn) barrier and derandomization. *SIAM Journal on Computing*, 45(3):947–1006, 2016.

23 Shang-En Huang and Seth Pettie. Thorup–zwick emulators are universally optimal hopsets. *Information Processing Letters*, 142:9–13, 2019.

24 Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 1997.

25 Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 308–321, 2020.

26 Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 121–130, 2010.

27 Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. *Journal of the European Mathematical Society*, 9(2):253–275, 2007.

28 Gary L Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015.

29 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the ACM Symposium on Theory of computing*. ACM, 2014.

30 Liam Roditty, Mikkel Thorup, and Uri Zwick. Deterministic constructions of approximate distance oracles and spanners. In *International Colloquium on Automata, Languages, and Programming*, pages 261–272. Springer, 2005.

31 Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 499–508. IEEE, 2004.

32 Atish Das Sarma, Michael Dinitz, and Gopal Pandurangan. Efficient distributed computation of distance sketches in networks. *Distributed Computing*, 28(5):309–320, 2015.

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

34 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

35 Mikkel Thorup and Uri Zwick. Spanners and emulators with sublinear distance errors. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 802–809, 2006.

36 Christian Wulff-Nilsen. Approximate distance oracles with improved preprocessing time. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 202–208. SIAM, 2012.