$(1+\epsilon)$-Approximate Shortest Paths in Dynamic Streams

Michael Elkin
Ben-Gurion University of the Negev, Beer-Sheva, Israel

Chhaya Trehan
London School of Economics & Political Science, UK

Abstract

Computing approximate shortest paths in the dynamic streaming setting is a fundamental challenge that has been intensively studied. Currently existing solutions for this problem either build a sparse multiplicative spanner of the input graph and compute shortest paths in the spanner offline, or compute an exact single source BFS tree. Solutions of the first type are doomed to incur a stretch-space tradeoff of $2\kappa-1$ versus $n^{1+1/\kappa}$, for an integer parameter κ. (In fact, existing solutions also incur an extra factor of $1+\epsilon$ in the stretch for weighted graphs, and an additional factor of $\log^{O(1)} n$ in the space.) The only existing solution of the second type uses $n^{1/2-O(1/\kappa)}$ passes over the stream (for space $O(n^{1+1/\kappa})$), and applies only to unweighted graphs.

In this paper we show that $(1+\epsilon)$-approximate single-source shortest paths can be computed with $\tilde{O}(n^{1+1/\kappa})$ space using just *constantly* many passes in unweighted graphs, and polylogarithmically many passes in weighted graphs. Moreover, the same result applies for multi-source shortest paths, as long as the number of sources is $O(n^{1/\kappa})$. We achieve these results by devising efficient dynamic streaming constructions of $(1+\epsilon,\beta)$-spanners and hopsets.

On our way to these results, we also devise a new dynamic streaming algorithm for the 1-sparse recovery problem. Even though our algorithm for this task is slightly inferior to the existing algorithms of [26, 11], we believe that it is of independent interest.

2012 ACM Subject Classification Theory of computation → Streaming models; Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Shortest paths; Theory of computation → Sparsification and spanners

Keywords and phrases Shortest Paths, Dynamic Streams, Approximate Distances, Spanners, Hopsets

Digital Object Identifier 10.4230/LIPIcs.APPROX/RANDOM.2022.51

Category APPROX

Related Version *Extended Version*: `https://arxiv.org/abs/2107.13309` [20]

Funding *Michael Elkin*: This research was supported by ISF grant 2344/19.

1 Introduction

Processing massive graphs is an important algorithmic challenge. This challenge is being met by intensive research effort. One of the most common theoretical models for addressing this challenge is the *semi-streaming* model of computation [22, 2, 36]. In this model, edges of an input n-vertex graph $G=(V,E)$ arrive one after another, while the storage capacity of the algorithm is limited. Typically it should be close to linear in the number of *vertices*, n. One usually allows space of $\tilde{O}(n)$, though it is often relaxed to $n^{1+o(1)}$, sometimes to $O(n^{1+\rho})$, for an arbitrarily small constant parameter $\rho>0$, or even to $O(n^{1+\eta_0})$, for some fixed constant η_0, $0<\eta_0<1$. Generally, the model allows several passes over the stream, and the objective is to keep both the number of passes and the space complexity of the algorithm in check.

The model comes in two main variations. In the first one, called *static* or *insertion-only* model [22], the edges can only arrive, and never get deleted. If the algorithm employs multiple passes, then the streams of edges observed on these passes may be permutations of one another, but are otherwise identical. In the more general *dynamic* (also known as *turnstile*) streaming setting [2], edges may either arrive or get deleted.

1.1 Distances in the Streaming Model

An important thread of the literature on dynamic streaming algorithms for graph problems is concerned with computing *distances* and constructing *spanners* and *hopsets*. This is also the topic of the current paper. For a pair of parameters $\alpha \geq 1$, $\beta \geq 0$, given an undirected graph $G = (V, E)$, a subgraph $G' = (V, H)$ of G is said to be an *(α, β)-spanner* of G, if for every pair $u, v \in V$ of vertices, it holds that $d_{G'}(u, v) \leq \alpha \cdot d_G(u, v) + \beta$, where d_G and $d_{G'}$ are the distance functions of G and G', respectively. A spanner with $\beta = 0$ is called a *multiplicative* spanner and one with $\alpha = 1$ is called an *additive* spanner. There is another important variety of spanners called *near-additive* spanners for which $\beta \geq 0$ and $\alpha = 1 + \epsilon$, for an arbitrarily small $\epsilon > 0$. The near-additive spanners are mostly applicable to *unweighted* graphs, even though there are some recent results about weighted near-additive spanners [14].

Spanners are very well-studied from both combinatorial and algorithmic viewpoints. It is well-known that for any parameter $\kappa = 1, 2, \ldots$, and for any n-vertex graph $G = (V, E)$, there exists a $(2\kappa - 1)$-spanner with $O(n^{1+1/\kappa})$ edges, and this bound is nearly-tight unconditionally, and completely tight under Erdős-Simonovits girth conjecture [38, 4]. The parameter $2\kappa - 1$ is called the *stretch* parameter of the spanner. Also, for any pair of parameters, $\epsilon > 0$ and $\kappa = 1, 2, \ldots$, there exists $\beta = \beta_{EP} = \beta(\kappa, \epsilon)$, so that for every n-vertex undirected graph $G = (V, E)$, there exists a $(1 + \epsilon, \beta)$-spanner with $O_{\kappa,\epsilon}(n^{1+1/\kappa})$ edges [18]. The additive term $\beta = \beta_{EP}$ in [18] behaves as $\beta(\kappa, \epsilon) \approx \left(\frac{\log \kappa}{\epsilon}\right)^{\log \kappa}$, and this bound is the state-of-the-art. A lower bound of $\Omega(\frac{1}{\epsilon \cdot \log \kappa})^{\log \kappa}$ for it was shown in [1].

Given an n-vertex weighted undirected graph $G = (V, E, \omega)$ and two parameters $\epsilon > 0$ and $\beta = 1, 2, \ldots$, a graph $G' = (V, H, \omega')$ is called a $(1 + \epsilon, \beta)$-*hopset* of G, if for every pair of vertices $u, v \in V$, we have

$$d_G(u, v) \leq d^{(\beta)}_{G \cup G'}(u, v) \leq (1 + \epsilon) \cdot d_G(u, v).$$

Here $d^{(\beta)}_{G \cup G'}(u, v)$ stands for β-bounded distance (See Definition 3) between u and v in $G \cup G'$. The parameter β is called the *hopbound* of the hopset G'.

Just like spanners, hopsets are a fundamental graph-algorithmic construct. They are extremely useful for computing approximate shortest distances and paths in various computational settings, in which computing shortest paths with a limited number of hops is significantly easier than computing them with no limitation on the number of hops. A partial list of these settings includes streaming, distributed, parallel and centralized dynamic models. Cohen [10] showed that for any undirected weighted n-vertex graph G, and parameters $\epsilon > 0$, $\rho > 0$, and $\kappa = 1, 2, \ldots$, there exists a $(1 + \epsilon, \beta_C)$-hopset with $\tilde{O}(n^{1+1/\kappa})$ edges, where $\beta_c = \left(\frac{\log n}{\epsilon}\right)^{O\left(\frac{\log \kappa}{\rho}\right)}$. Elkin and Neiman [16] improved Cohen's result, and constructed hopsets with *constant* hopbound. Specifically, they showed that for any $\epsilon > 0$, $\kappa = 1, 2, \ldots$, and any n-vertex weighted undirected graph, there exists a $(1 + \epsilon, \beta_{EN})$-hopset with $\tilde{O}(n^{1+1/\kappa})$ edges, and $\beta_{EN} = \beta_{EP} \approx (\frac{\log \kappa}{\epsilon})^{\log \kappa}$. The lower bound of [1], $\beta = \Omega(\frac{1}{\epsilon \cdot \log \kappa})^{\log \kappa}$ is applicable to hopsets as well. Generally, hopsets (see [10, 28, 16]) are closely related to near-additive spanners. See a recent survey [17] for an extensive discussion on this relationship.

Most of the algorithms for computing (approximate) distances and shortest paths in the streaming setting compute a sparse spanner, and then employ it for computing exact shortest paths and distances in it offline, i.e., in the post-processing, after the stream is over [23, 12, 7, 21, 15, 3, 32, 24, 25]. Specifically, in the dynamic streaming model, algorithms for computing approximately shortest paths (with space $\tilde{O}(n^{1+1/\kappa})$, for a parameter $\kappa = 1, 2, \ldots$), can be divided into two categories. The algorithms in the first category build a sparse multiplicative $(2\kappa-1)$-spanner, and they provide a *multiplicative* stretch of at least $2\kappa-1$ [3, 32, 24, 25]. Moreover, due to existential lower bounds for spanners, this approach is doomed to provide stretch of at least $\frac{4}{3}\kappa$ [35]. The algorithms in the second category compute *exact single source* shortest paths in *unweighted* graphs, but they employ $n^{1/2-O(1/\kappa)}$ passes [9, 13]. In the current paper, we partially fill in the gap between these two extremes, and devise a dynamic streaming $(1+\epsilon)$-approximate SSSP algorithm with space $\tilde{O}(n^{1+1/\kappa})$ that uses $\left(\frac{\kappa}{\epsilon}\right)^{\kappa(1+o(1))}$ passes. Table 1 summarizes the existing algorithms for the problem of computing approximate shortest paths in streaming setting on unweighted graphs, and compares them to our results.

Table 1 Prior Work on the Problem.

Citation	Model	Stretch	Space	No. of Passes	Technique
[23]	Static	$(2\kappa+1)$	$\tilde{O}(n^{1+1/\kappa})$	1	Multiplicative Spanner
[12, 7]	Static	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	1	Multiplicative Spanner
[21, 15]	Static	$(1+\epsilon, \beta_{EN})$	$\tilde{O}(n^{1+1/\kappa})$	β_{EN}	Near-Additive Spanner
[29]	Static	$(1+\epsilon)$	$n^{1+o(1)} \cdot O(\frac{\log\Lambda}{\epsilon})$	$2^{O(\sqrt{\log n \log\log n})} = n^{o(1)}$	Hopsets
[16]	Static	$(1+\epsilon)$	$\tilde{O}(n^{1+\rho})$, for $\rho > 0$	$\left(\frac{\log n}{\epsilon\cdot\rho}\right)^{\frac{1}{\rho}(1+o(1))}$	Hopsets
[13]	Static	Exact	$O(n \cdot p)$, for $1 \le p \le n$	$O(n/p)$	Hopsets
[3]	Dynamic	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	κ	Multiplicative Spanner
[24]	Dynamic	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	$\lfloor\kappa/2\rfloor+1$	Multiplicative Spanner
[3]	Dynamic	$O(\kappa^{\log_2 5})$	$\tilde{O}(n^{1+1/\kappa})$	$O(\log\kappa)$	Multiplicative Spanner
[25]	Dynamic	$O(\kappa^{\log_2 3})$	$\tilde{O}(n^{1+1/\kappa})$	$O(\log\kappa)$	Multiplicative Spanner
[9]	Dynamic	Exact	$\tilde{O}(n+p^2)$, for $1 \le p \le n$	$\tilde{O}(n/p)$	Exact BFS
(This Paper)	Dynamic	$(1+\epsilon)$	$\tilde{O}(n^{1+1/\kappa})$	Constant (unweighted graphs) and polylog (weighted graphs)	Near-Additive Spanners and Hopsets

The algorithms of [23, 12, 7] apply to unweighted graphs, but they can be extended to weighted graphs by running many copies of them in parallel, one for each weight scale. Let $\Lambda = \Lambda(G)$ denote the *aspect ratio* of the graph, i.e., $\Lambda = \frac{max_{u,v\in V} d_G(u,v)}{min_{u,v\in V} d_G(u,v)}$. Also, let $\epsilon \ge 0$ be a slack parameter. Then by running $O(\frac{\log\Lambda}{\epsilon})$ copies of the algorithm for unweighted graphs and taking the union of their outputs as the ultimate spanner, one obtains a one-pass static streaming algorithm for $2(1+\epsilon)\kappa$-spanner with $\tilde{O}(n^{1+\frac{1}{\kappa}} \cdot (\log\Lambda)/\epsilon)$ edges [19].

In [21], the authors devised static streaming algorithms for building $(1+\epsilon, \beta_{EZ})$-spanners where, $\beta_{EZ} = \beta_{EZ}(\epsilon, \rho, \kappa) = \left(\frac{\log\kappa}{\epsilon\cdot\rho}\right)^{O(\frac{\log\kappa}{\rho})}$, for any parameters $\epsilon, \rho > 0$ and $\kappa = 1, 2, \ldots$. This result was improved in [15], where a static streaming algorithm with similar properties, but with $\beta = \beta_{EN} = \left(\frac{\log\kappa\rho+1/\rho}{\epsilon}\right)^{\log\kappa\rho+1/\rho}$ was devised. The algorithms of [21, 15] directly give rise to β-pass static streaming algorithms with space $\tilde{O}(n^{1+\rho})$ for $(1+\epsilon, \beta)$-APASP (*All Pairs Almost Shortest Paths*) in unweighted graphs where $\beta(\rho) \approx (1/\rho)^{(1/\rho)(1+o(1))}$. They can also be used for producing purely multiplicative $(1+\epsilon)$-approximate shortest paths and distances in $O(\beta/\epsilon)$ passes and $\tilde{O}(n^{1+\rho})$ space from up to $n^{\rho(1-o(1))}$ designated sources to all other vertices. There are also a number of additional *not* spanner-based static streaming algorithms for computing approximate shortest paths. Henzinger et al. [29] and Elkin and Neiman [16] devised $(1+\epsilon)$-approximate *single*-source shortest paths (henceforth, SSSP) algorithms for weighted graphs, that are based on *hopsets*.

Recently Chang et al. [9] devised a *dynamic streaming* algorithm for this problem in *unweighted* graphs. Their algorithm uses $\tilde{O}(n/p)$ passes (for parameter $1 \leq p \leq n$ as above) and space $\tilde{O}(n+p^2)$ for the SSSP problem, and space $\tilde{O}(|S|n+p^2)$ for the $S \times V$ approximate shortest path computation.

Ahn et al. [3] devised the first *dynamic streaming* algorithm for computing approximate distances. Their algorithm computes a $(2\kappa-1)$-spanner (for any $\kappa = 1, 2, \ldots$) with $\tilde{O}(n^{1+1/\kappa})$ edges (and the same space complexity) in κ passes over the stream. This bound was recently improved by [24]. Their algorithm computes a spanner with the same properties using $\lfloor \kappa/2 \rfloor + 1$ passes. Ahn et al. [3] also devised an $O(\log \kappa)$-pass algorithm for building $O(\kappa^{\log_2 5})$-spanner with size and space complexity $\tilde{O}(n^{1+1/\kappa})$. This bound was recently improved by [25], whose algorithm produces $O(\kappa^{\log_2 3})$-spanner with the same pass and space complexities, and the same size.

Another dynamic streaming algorithm was devised by Kapralov and Woodruff [32]. It produces a $(2^\kappa - 1)$-spanner with $\tilde{O}(n^{1+1/\kappa})$ edges (and space usage) in two passes. Kapralov et al. [25] improved the stretch parameter of the spanner to $2^{\frac{\kappa+3}{2}} - 3$, with all other parameters the same as in the results of [32]. Kapralov et al. [25] also devised a general tradeoff in which the number of passes can be between 2 and κ, and the stretch of the spanner decreases gracefully from exponential in κ (where the number of passes is 2) to $2\kappa - 1$ (when the number of passes is κ). They have also devised a single pass algorithm with stretch $\tilde{O}(n^{\frac{2}{3}(1-1/\kappa)})$. As was mentioned above, most of these spanner-based algorithms provide a solution for $(2\kappa-1)$-APASP for unweighted graphs with space $\tilde{O}(n^{1+1/\kappa})$ and the number of passes equal to that of the spanner-construction algorithm. Like their static streaming counterparts [23, 12, 7], they can be extended to weighted graphs, at the price of increasing their stretch by a factor of $1+\epsilon$ (for an arbitrarily small parameter $\epsilon > 0$), and their space usage by a factor of $O\left(\frac{\log \Lambda}{\epsilon}\right)$.

1.2 Our Results

In the current paper, we present the first dynamic streaming algorithm for SSSP with stretch $1+\epsilon$, space $\tilde{O}(n^{1+1/\kappa})$, and *constant* (as long as ϵ and κ are constant) number of passes for unweighted graphs. For weighted graphs, our number of passes is *polylogarithmic* in n. Specifically, the number of passes of our SSSP algorithm is $(\frac{\kappa}{\epsilon})^{\kappa(1+o(1))}$ for unweighted graphs, and $\left(\frac{(\log n)\kappa}{\epsilon}\right)^{\kappa(1+o(1))}$ for weighted ones. Moreover, within the same complexity bounds, our algorithm can compute $(1+\epsilon)$-approximate $S \times V$ shortest paths from $|S| = n^{1/\kappa}$ designated sources. Moreover, in *unweighted* graphs, *all* pairs almost shortest paths with stretch $(1+\epsilon, \left(\frac{\kappa}{\epsilon}\right)^\kappa)$ can also be computed within the same space and number of passes. (That is, paths and distances with multiplicative stretch $1+\epsilon$ and additive stretch $\left(\frac{\kappa}{\epsilon}\right)^\kappa$.) Note that our multiplicative stretch $(1+\epsilon)$ is dramatically better than $(2\kappa-1)$, exhibited by algorithms based on multiplicative spanners [3, 32, 24, 25], but this comes at a price of at least exponential increase in the number of passes. Nevertheless, our number of passes is *independent of* n, for unweighted graphs, and depends only polylogarithmically on n for weighted ones.

1.3 Technical Overview

We devise algorithms for building near-exact hopsets and near-additive spanners in dynamic streaming model. These structures help us compute almost shortest paths in the input graph. The following theorem summarizes the resource usage and properties of our hopset construction.

▶ **Theorem 1** (Theorem 9 in Section 3.4). *For any n-vertex graph $G(V,E,\omega)$ with aspect ratio Λ, $2 \leq \kappa \leq (\log n)/4$, $1/\kappa \leq \rho \leq 1/2$ and $0 < \epsilon < 1$, our dynamic streaming algorithm computes whp, a $(1+\epsilon,\beta)$ hopset H with expected size $O(n^{1+1/\kappa} \cdot \log n)$ and the hopbound β given by, $\beta = O\left(\frac{(\log \kappa\rho + 1/\rho)\log n}{\epsilon}\right)^{\log \kappa\rho + 1/\rho}$. It does so by making $O(\beta \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream and using $O(n \cdot \log^3 n \cdot \log \Lambda)$ bits of space in the first pass and $O(\frac{\beta}{\epsilon} \cdot \log^2 1/\epsilon \cdot n^{1+\rho} \cdot \log^5 n)$ bits of space (respectively $O(\frac{\beta^2}{\epsilon} \cdot \log^2 1/\epsilon \cdot n^{1+\rho} \cdot \log^5 n)$ bits of space for path-reporting hopset) in each of the subsequent passes.*

The hopset is then used (in Section 4) to compute almost shortest paths in weighted graphs.

In the extended version, we also show a similar algorithm for constructing near-additive spanners. This result is summarized in the following theorem.

▶ **Theorem 2.** *For any unweighted graph $G(V,E)$ on n vertices, parameters $0 < \epsilon < 1$, $\kappa \geq 2$, and $\rho > 0$, our dynamic streaming algorithm computes a $(1+\epsilon,\beta)$-spanner with $O_{\epsilon,\kappa,\rho}(n^{1+1/\kappa})$ edges, in $O(\beta)$ passes using $O(n^{1+\rho}\log^4 n)$ space with high probability, where β is given by, $\beta = \left(\frac{\log \kappa\rho + 1/\rho}{\epsilon}\right)^{\log \kappa\rho + 1/\rho}$.*

Our algorithms for spanner and hopset construction extend the results of [15, 16] from the static streaming setting to dynamic streaming one. The algorithms of [15, 16], like their predecessor, the algorithm of [18], are based on the superclustering-and-interconnection (henceforth, SAI) approach. Our algorithms in the current paper also fall into this framework. Algorithms that follow the SAI approach proceed in phases, and in each phase they maintain a partial partition of the vertex set V of the graph. Some of the clusters of G are selected to create superclusters around them. This is the superclustering step. Clusters that are not superclustered into these superclusters are then *interconnected* with their nearby clusters. The main challenge in implementing this scheme in the dynamic streaming setting is in the interconnection step. Indeed, the superclustering step requires a single and rather shallow BFS exploration, and implementing depth-d BFS in unweighted graphs in d passes over the dynamic stream can be done in near-linear space (See, e.g., [3, 9]). For the weighted graphs, we devise a routine for performing an approximate Bellman-Ford exploration up to a given hop-depth d, using d passes and $\tilde{O}(n)$ space.

On the other hand, the interconnection step requires implementing simultaneous BFS (Bellman-Ford for the weighted case) distance explorations originated at multiple sources. A crucial property that enabled [15, 16] to implement it in the static streaming setting is that one can argue that with high probability, not too many distance explorations traverse any particular vertex. Let us denote by N, an upper bound on the number of explorations (traversing any particular vertex). In the dynamic streaming setting, however, at any point of the stream, there may well be much more than N explorations that traverse a specific vertex $v \in V$, based on the stream of updates observed so far. Storing data about all these explorations would make the space requirement of the algorithm prohibitively large.

To resolve this issue (and a number of related similar issues), we incorporate a *sparse recovery* routine into our algorithms. Sparse recovery is a fundamental and well-studied primitive in the dynamic streaming setting [26, 11, 30, 5]. It is defined for an input which is a stream of (positive and negative) updates to an n-dimensional vector $\vec{a} = (a_1, a_2, \ldots, a_n)$. In the *strict* turnstile setting, which is sufficient for our application, ultimately each coordinate a_i (i.e., at the end of the stream) is non-negative, even though negative updates are allowed and intermediate values of coordinates may be negative. In the *general* turnstile model coordinates of the vector $\vec{a}$ may be negative at the end of the stream as well. The *support* of $\vec{a}$, denoted $supp(\vec{a})$, is defined as the set of its non-zero coordinates. For a parameter

s, an *s-sparse recovery* routine returns the vector $\vec{a}$, if $|supp(\vec{a})| \leq s$, and returns failure otherwise. (It is typically also allowed to return failure with some small probability $\delta > 0$, given to the routine as a parameter, even if $|supp(\vec{a})| \leq s$.)

Most of sparse recovery routines are based on 1-sparse recovery, i.e., the case $s = 1$. The first 1-sparse recovery algorithm was devised by Ganguly [26], and it applies to the strict turnstile setting. The space requirement of the algorithm of [26] is $O(\log n)$. The result was later extended to the general turnstile setting by Cormode and Firmani [11] (See also [37]). We devise an alternative streaming algorithm for this basic task in the strict turnstile setting. The space complexity of our algorithm is $O(\log n)$, like that of [26]. The processing time-per-item of [26]'s algorithm is however $O(1)$, instead of $polylog(n)$ of our algorithm.[1] Nevertheless, we believe that our new algorithm for this task is of independent interest.

In the current paper paper we analyze our algorithm in terms of the aspect ratio Λ of the input graph, given by $\Lambda = \frac{max_{u,v\in V} d_G(u,v)}{min_{u,v\in V} d_G(u,v)}$. (All dependencies are polylogarithmic in Λ.) In the extended version [20], we also show that [34]'s weight reduction (see also [16]) can be implemented in the dynamic streaming model. As a result, we replace all appearances of $\log \Lambda$ in the hopset's size, hopbound and number of passes of our construction by $O(\log n)$. However, the space complexity of our algorithm still mildly depends on $\log \Lambda$. Specifically, it is $\tilde{O}(n^{1+\rho})+\tilde{O}(n)\cdot\log\Lambda$. In all existing dynamic streaming algorithms for computing multiplicative spanners or computing approximate shortest paths in weighted graphs [3, 32, 24, 25], both the spanner's size and the space requirements are $\tilde{O}(n^{1+1/\kappa}\cdot\log\Lambda)$. But using our weight reduction in conjunction with these algorithms, one can produce spanners of size $\tilde{O}(n^{1+1/\kappa})$ (without dependence on $\log\Lambda$), using space $\tilde{O}(n^{1+1/\kappa})+\tilde{O}(n)\cdot\log\Lambda$. However, the number of passes increases by an additive term of 1. Completely eliminating the dependence on $\log\Lambda$ from these results is left as an open problem.

1.4 Outline

The rest of the paper is organized as follows. Section 2 provides necessary definitions and concepts. Section 3 presents an algorithm for constructing hopsets with constant hopbound. All the missing proofs and a more thorough analysis of our hopset construction are available in the extended version [20]. Section 4 shows how we use the algorithm of Section 3 to compute $(1+\epsilon)$-approximate shortest paths in weighted graphs. The spanner construction algorithm is very similar to the hopset construction algorithm. The spanner construction algorithm and details on its usage in computing approximate shortest paths in unweighted graphs are available in the extended version [20] of this paper. The subroutine for performing a limited depth Bellman-Ford exploration in the input graph is described in Appendix B. Appendix appears after the references.

2 Preliminaries

2.1 Streaming Model

In the *dynamic streaming* model of computation, the set of vertices V of the input graph is known in advance and the edge set E is revealed one at a time. The edges can be added as well as removed. For a weighted input graph, the stream S arrives as a sequence of

[1] If the algorithm knows in advance the dimension n of the vector $\vec{a}$ and is allowed to compute during preprocessing, before seeing the stream, a table of size n, then our algorithm can also have $O(1)$ processing time per update. This scenario occurs in dynamic streaming graph algorithms, including those discussed in the current paper.

edge updates $S = \langle s_1, s_2, \cdots \rangle$, where $s_t = (e_t, eSign_t, eWeight_t)$, where e_t is the edge being updated and $eWeight_t$ is its weight. The $eSign_t \in \{+1, -1\}$ value of an update indicates whether the edge e_t is to be added or removed. A value of $+1$ indicates addition and a value of -1 indicates removal.

For a weighted undirected graph $G = (V, E, \omega)$, we assume that the edge weights are scaled so that the minimum edge weight is 1. Let $maxW$ denote the maximum edge weight $\omega(e)$, $e \in E$. For a non-edge $(u, v) \notin E$, we define $\omega((u, v)) = \infty$.

▶ **Definition 3.** *Given a weighted graph $G(V, E, \omega)$, a positive integer parameter t, and a pair $u, v \in V$ of distinct vertices, a* t**-bounded** u**-**v ***path*** *in G is a path between u and v that contains no more than t edges (also known as* hops*), and* t**-bounded distance** *between u and v in G denoted $d_G^{(t)}(u, v)$ is the length of a shortest t-bounded u-v path in G.*

2.2 Samplers, Hash Functions and Vertex Encodings

The main technical tool in our algorithms is a space-efficient sampling technique which enables us to sample a single vertex or a single edge from an appropriate subset of the vertex set or the edge set of the input graph, respectively. Algorithms for sampling from a dynamic stream are inherently randomized and often use hash functions as a source of randomness. Appendix A is devoted to hash functions.

2.3 Vertex Encodings

We assume that the vertices have unique IDs from the set $\{1, \ldots, n\}$. The maximum possible ID (which is n) of a vertex in the graph is denoted by $maxVID$. The binary representation of the ID of a vertex v can be obtained by performing a name operation $name(v)$. For an integer $k \geq 1$, $[k]$ denotes the set $\{1, 2, \ldots, k\}$.

We also need standard definitions of convex combination, convex hull and a convexly independent set. We will use the following *CIS*-based encoding for the vertices of the graph:
CIS Encoding Scheme $\boldsymbol{\nu}$: We assign a unique code in $\mathbb{Z}^2$ to every vertex $v \in V$. The encoding scheme works by generating a set of n convexly independent integer vectors in $\mathbb{Z}^2$. Specifically, our encoding scheme uses as its range, the extremal points of the convex hull of $Ball_2(R) \cap \mathbb{Z}^2$, where $Ball_2(R)$ is a two-dimensional disc of radius R centered at origin. A classical result by [31], later refined by [6], states that the number of extremal points of the convex hull of a set of integer points of a disc of radius R is $\Theta(R^{2/3})$. We set $R = \Theta(n^{3/2})$ to allow for all the possible $n = \Theta(R^{2/3})$ vertices to be encoded in $O(\log n)$ bits. The encoding of any vertex v can be obtained by performing an encoding operation denoted by $\nu(v)$.

We prove the following lemma in extended version which will be be useful in Section 3.2 to detect if the sampling procedure succeeded in sampling exactly one vertex from a desired subset of the set V.

▶ **Lemma 4.** *Let $c_1, c_2, \cdots, c_n$ be non-negative integer coefficients of a linear combination of a set $P = \{p_1, p_2, \cdots p_n\}$ of n convexly independent points in $\mathbb{Z}^2$ such that $\frac{\sum_{j=1}^{n} c_j \cdot p_j}{\sum_{j=1}^{n} c_j} = p_i$, for some $p_i \in P$. Then $c_j = 0$ for every $j \neq i$, and $c_i \neq 0$.*

3 Hopsets with Constant Hopbound in Dynamic Streaming Model

We adapt the insertion-only streaming algorithm of [16] for hopset construction to work in the dynamic streaming setting. The algorithm is based on the SAI (Superclustering and Interconnection) approach. (See [16] for more details.) The main ingredient of both

the superclustering and interconnection steps is a set of Bellman-Ford explorations (*B-F explorations*, henceforth) up to a given distance in the input graph from a set of chosen vertices. The insertion-only streaming algorithm of [16] identifies all the edges spanned by $\Theta(\beta)$ iterations of certain B-F explorations up to a distance δ from a set of chosen vertices, by making $\Theta(\beta)$ passes through the stream. Other parts of the hopset construction, such as identifying the vertices of the graph from which to perform B-F explorations and subsequently adding edges corresponding to certain paths traversed by these explorations to the hopset, are performed offline.

We devise a technique to perform a given number of iterations of a B-F exploration from a set of chosen vertices and up to a given distance in the graph in the dynamic streaming setting, and as in [16], perform the rest of the work offline. The difference however is that in the dynamic streaming setting, we do not perform an exact and deterministic B-F exploration (as in [16]). A randomized algorithm for performing an approximate B-F exploration originated at a subset of source vertices in a weighted graph, that succeeds whp, is described in Appendix B. We use this algorithm as a subroutine in the superclustering step of our main algorithm. The interconnection step is more challenging and involves performing multiple simultaneous B-F explorations in a weighted graph, each from a separate source vertex. Here, each vertex in the graph needs to identify all the B-F explorations it is a part of, and to find its (approximate) distance to the source of each such exploration. Due to the dynamic nature of the stream, a given vertex may find itself on a lot more explorations than it finally ends up belonging to. This can be dealt with by combining a delicate encoding/decoding scheme for the IDs of exploration sources with a space-efficient sampling technique.

In the following section, we provide an overview of our hopset construction algorithm.

3.1 Overview

Our hopset construction algorithm takes as input an n-vertex weighted undirected graph $G = (V, E, \omega)$ with aspect ratio Λ, and parameters $0 < \epsilon' < 1/10$, $\kappa = 1, 2, \ldots$ and $1/\kappa < \rho < 1/2$, and produces as output a $(1+\epsilon', \beta')$-hopset of G. The hopbound parameter β' is a function of ϵ', Λ, κ, ρ and is given by $\beta' = O\left(\frac{\log \Lambda}{\epsilon'} \cdot (\log \kappa\rho + 1/\rho)\right)^{\log \kappa\rho + 1/\rho}$.

Let $k = 0, 1, \ldots, \lceil \log \Lambda \rceil - 1$. Given two parameters $\epsilon > 0$ and $\beta = 1, 2, \ldots$, a set of weighted edges H_k on the vertex set V of the input graph is said to be a $(1+\epsilon, \beta)$-*hopset for the scale* k or a *single-scale hopset*, if for every pair of vertices $u, v \in V$ with $d_G(u, v) \in (2^k, 2^{k+1}]$ we have that

$$d_G(u, v) \leq d^{(\beta)}_{G_k}(u, v) \leq (1+\epsilon) \cdot d_G(u, v),$$

where $G_k = (V, E \cup H_k, \omega_k)$ and $\omega_k(u, v) = min\{\omega(u, v), \omega_{H_k}(u, v)\}$, for every edge $(u, v) \in E \cup H_k$. Let $\epsilon > 0$ be a parameter that will be determined later in the sequel. (See Section 3.3.) Set also $\ell = \lfloor \log \kappa\rho \rfloor + \lceil \frac{\kappa+1}{\kappa\rho} \rceil - 1$. Let $\beta = (1/\epsilon)^\ell$.

The algorithm constructs a separate $(1+\epsilon, \beta)$-hopset H_k for every scale $(2^0, 2^1], (2^1, 2^2], \ldots, (2^{\lceil \log \Lambda \rceil - 1}, 2^{\lceil \log \Lambda \rceil}]$ one after another. For $k \leq \lfloor \log \beta \rfloor - 1$, we set $H_k = \phi$. We can do so because for such a k, it holds that $2^{k+1} \leq \beta$, and for every pair of vertices u, v with $d_G(u, v) \leq 2^{k+1}$, the original graph G itself contains a shortest path between u and v that contains at most β edges. (We remark that after rescaling, we will have $\beta' = \beta$. See Section 3.3.) In other words, $d_G(u, v) = d^{(\beta)}_G(u, v)$. Denote $k_0 = \lfloor \log \beta \rfloor$ and $k_\Lambda = \lceil \log \Lambda \rceil - 1$. We construct a hopset H_k for every $k \in [k_0, k_\Lambda]$.

During the construction of the hopset H_k for some $k \geq k_0$, we need to perform explorations from certain vertices in V up to distance $\delta \leq 2^{k+1}$ in G. An exploration up to a given distance from a certain vertex in G may involve some paths with up to $n-1$ hops. This can take up to $O(n)$ passes through the stream. We overcome this problem by using the hopset edges $H^{(k-1)} = \bigcup_{k_0 \leq j \leq k-1} H_j$ for constructing hopset H_k. The hopset H_k has to take care of all pairs of vertices u, v with $d_G(u,v) \in (2^k, 2^{k+1}]$, whereas the edges in $E \cup H^{(k-1)}$ provide a $(1+\epsilon_{k-1})$-approximate shortest path with up to β hops, for every pair u, v with $d_G(u,v) \leq 2^k$. The value of ϵ_{k-1} will be specified later in the sequel. (See Section 3.3.) Denote by $G^{(k-1)}$ the graph obtained by adding the edge set $H^{(k-1)}$ to the input graph G. Instead of conducting explorations from a subset $S \subseteq V$ up to distance $\delta \leq 2^{k+1}$ in the input graph G, we perform $2\beta+1$ iterations of B-F algorithm on the graph $G^{(k-1)}$ up to distance $(1+\epsilon_{k-1}) \cdot \delta$. The following lemma from [16] provides a justification for this approach.

▶ **Lemma 5** ([16]). *For $u, v \in V$ with $d_G(u,v) \leq 2^{k+1}$, the following holds:*

$$d^{(2\beta+1)}_{G^{(k-1)}}(u,v) \leq (1+\epsilon_{k-1}) \cdot d_G(u,v) \tag{1}$$

3.2 Constructing H_k

We now proceed to the construction of the hopset H_k for the scale $(2^k, 2^{k+1}]$, for some $k \in [k_0, k_\Lambda]$. We start by initializing the hopset H_k as an empty set and proceed in phases $0, 1, \ldots, \ell$. All phases of our algorithm except for the last one consist of two steps, a superclustering step and an interconnection step. In the last phase, we go directly to the interconnection step. Throughout the algorithm, we build clusters of nearby vertices. The input to phase $i \in [0, \ell]$ is a set of clusters P_i, a distance threshold parameter δ_i and a degree parameter $\deg_i$. For phase 0, the input P_0 is a partition of the vertex set V into singleton clusters. Every cluster created by our algorithm has a designated *center* vertex. We denote by r_C the center of cluster C. In particular, each singleton cluster $C = \{v\}$ is centered around v. For a cluster C, we define $Rad(C) = max\{d_{G(C)}(r_C, v) \mid v \in C\}$. For a set of clusters P_i, $Rad(P_i) = \underset{C \in P_i}{max}\{Rad(C)\}$.

The degree threshold parameter deg_i of phase i is used to define the sampling probability with which the centers of clusters in P_i are selected to grow superclusters around them. We partition the phases of the algorithm into two stages based on how the degree parameter grows in each stage. The two stages are the *exponential growth stage* and the *fixed grown stage*. In the *exponential growth stage*, which consists of phases $0, 1, \ldots, i_0 = \log\lfloor \kappa\rho \rfloor$, we set $deg_i = n^{\frac{2^i}{\kappa}}$. In the *fixed growth stage*, which consists of phases $i_0+1, i_0+2, \ldots, i_1 = i_0 + \lceil \frac{\kappa+1}{\kappa\rho} \rceil$, we set $deg_i = n^\rho$. Observe that for every index i, we have $deg_i \leq n^\rho$.

The distance threshold parameter increases by a factor of $1/\epsilon$ in every phase. The sequence of the distance threshold parameters for the centralized construction as defined in [16] is given by $\alpha = \alpha^{(k)} = \epsilon^\ell \cdot 2^{k+1}$, $\delta_i = \alpha(1/\epsilon)^i + 4R_i$, where $R_0 = 0$ and $R_{i+1} = R_i + \delta_i = \alpha(1/\epsilon)^i + 5R_i$ for $i \geq 0$. Here α can be perceived as a unit of distance. To adjust for the fact that explorations are performed on the graph $G^{(k-1)}$, and not on the input graph G, we multiply all the distance thresholds δ_i by a factor of $1+\epsilon_{k-1}$, the stretch guarantee of the graph $G^{(k-1)}$. We further modify this sequence to account for the fact that our B-F explorations in the dynamic stream are not exact and incur a multiplicative error. Throughout the construction of H_k, we set the multiplicative error of every approximate B-F Exploration we perform to $1+\chi$, for a parameter $\chi > 0$ which will be determined later (in Section 3.3). Therefore we multiply all the distance thresholds by a factor of $1+\chi$. We define $R'_i = (1+\chi) \cdot (1+\epsilon_{k-1}) R_i$ and $\delta'_i = (1+\chi) \cdot (1+\epsilon_{k-1})\delta_i$ for every $i \in [0, \ell]$. In the centralized setting, R_i serves as an upper bound on the radii of the input clusters of phase i. As a result of rescaling, R'_i becomes the new upper bound on the radii of input clusters of phase i.

Superclustering. The phase i begins by sampling each cluster $C \in P_i$ independently at random with probability $1/deg_i$. Let S_i denote the set of sampled clusters. We now have to conduct (approximate) distance exploration up to depth δ'_i in $G^{(k-1)}$ rooted at the set $CS_i = \bigcup_{C \in S_i}\{r_C\}$. By Lemma 5, this can be achieved by $2\beta+1$ iterations of B-F algorithm on the graph $G^{(k-1)}$. For this, we invoke the approximate B-F exploration algorithm of Appendix B on graph $G^{(k-1)}$ with set CS_i as the set S of source vertices and parameters $\eta = 2\beta+1$, $\zeta = \chi$. We slightly modify the algorithm of Appendix B and then invoke the modified version. In the modified version, at the end of each pass through the stream, for every vertex $v \in V$, we scan through the edges incident to v in the set $H^{(k-1)}$ and update its distance estimate $\hat{d}(v)$ as:

$$\hat{d}(v) = \min\{\hat{d}(v), \min_{(v,w)\in H^{(k-1)}}\{\hat{d}(w) + \omega_{H^{(k-1)}}(v,w)\}\}.$$

The parent of v, $\hat{p}(v)$ is also updated accordingly. Note that this modification does not affect the space complexity, stretch guarantee or the success probability of the algorithm of Appendix B. This provides us with a $(1+\chi)$-approximation of $d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i)$, for all $v \in V$. Hence, by Theorem 14, an invocation of modified version of approximate B-F algorithm of Appendix B during the the superclustering step of phase i generates whp, an approximate B-F exploration of the graph $G^{(k-1)}$, rooted at the set $CS_i \subseteq V$ in $2\beta+1$ passes. It outputs for every $v \in V$ an estimate $\hat{d}(v)$ of its distance to set CS_i such that:

$$d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i) \leq \hat{d}(v) \leq (1+\chi)\cdot d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i). \tag{2}$$

Moreover, the set of parent variables $\hat{p}(v)$ of every $v \in V$ with $\hat{d}(v) < \infty$ spans a forest F of $G^{(k-1)}$ rooted at the set of sampled centers CS_i.

For every cluster center $r_{C'}$, $C' \in P_i \setminus S_i$, such that $\hat{d}(r_{C'}) \leq \delta'_i$, the algorithm adds an edge $(r_C, r_{C'})$ of weight $\hat{d}(r_{C'})$ to the hopset H_k, where r_C is the root of the tree in F to which $r_{C'}$ belongs. We also create a supercluster rooted at r_C which contains all the vertices of C' as above. Note that if $d_G(r_C, r_{C'}) \leq \delta_i$, then by equations (1) and (2), $\hat{d}(r_{C'}) \leq (1+\chi)\cdot(1+\epsilon_{k-1})d_G(r_C, r_{C'}) = \delta'_i$. Therefore, edge $(r_C, r_{C'})$ will be added to the hopset and the cluster C' will be superclustered into a cluster centered at r_C.

Interconnection. Next we describe the interconnection step of each phase $i \in \{0, 1, \ldots, \ell\}$. Let U_i be the set of clusters of P_i that were not superclustered in phase i. Let $CU_i = \bigcup_{C\in U_i}\{r_C\}$. In the interconnection step of phase $i \geq 0$, we want to connect every cluster $C \in U_i$ to every other cluster $C' \in U_i$ that is close to it. To do this, we want to perform $2\beta+1$ iterations of a $(1+\chi)$-approximate B-F exploration from every cluster center $r_C \in CU_i$ *separately* in $G^{(k-1)}$. These explorations are, however, conducted to a bounded depth (in terms of number of hops), and to bounded distance. Specifically, the hop-depth of these explorations will be at most $2\beta+1$, while the distance to which they are conducted is roughly $\delta_i/2$. For every cluster center $r_{C'}$, $C' \in U_i$ within distance $\delta_i/2$ from another center r_C in G, we want to add an edge $e = (r_C, r_{C'})$ of weight at most $(1+\chi)\cdot d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$ to the hopset H_k. To do so, we turn to the stream to find an estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C)$ for every $v \in V$ and every center $r_C \in U_i$. We cannot afford to invoke the algorithm of Appendix B multiple times in parallel to conduct a separate exploration from every center r_C in CU_i, due to space constraints. As shown in [16] (See Lemmas 3.2 and 3.3 of [16]), the following lemma holds in the interconnection step of our (single-scale) hopset construction:

▶ **Lemma 6** ([15]). *For any vertex $v \in V$, the expected number of explorations that visit v in the interconnection step of phase i is at most deg_i. Moreover, for any constant c_1', every vertex v is explored by at most $c_1' \cdot \ln n \cdot \deg_i$ explorations in phase i with probability at least $1 - 1/n^{c_1'-1}$.*

Specifically, if one conducts B-F explorations to depth at most $\delta_i'/2$ in $G^{(k-1)}$ to hop-depth at most $2\beta+1$, then, whp, every vertex is traversed by at most $O(deg_i \ln n)$ explorations. Therefore, we devise a randomized technique to efficiently identify for every $v \in V$, the sources of all the explorations it gets visited by in phase i. Moreover, for every vertex $v \in V$ with a non-empty subset $U_i^v \subseteq U_i$ of explorations that visit v, we find for every cluster $C \in U_i^v$, an estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C)$.

Throughout the interconnection step of phase i, we maintain for every vertex $v \in V$, a set $LCurrent_v$ (called *estimates list* of v) of sources of B-F explorations that visited v so far. Each element of $LCurrent_v$ is a tuple $(s, \hat{d}(v,s))$, where s is the center of some cluster in U_i, and $\hat{d}(v,s)$ is the current estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$. For any center $s' \in CU_i$, for which we do not yet have a tuple in $LCurrent_v$, $\hat{d}(v,s')$ is implicitly defined as ∞. Initially, the estimates lists of all the vertices are empty, except for the centers of clusters in U_i. The estimates list of every center $r_C \in CU_i$ is initialized with a single element $(r_C, 0)$ in it. The interconnection step of phase i is carried out in $2\beta+1$ sub-phases. Next, we describe the purpose of each of the $2\beta+1$ sub-phases of the interconnection step and the way they are carried out.

Sub-phase p of the interconnection step. Denote $\zeta' = \frac{\chi}{2\cdot(2\beta+1)}$. Our goal is to ensure that by the end of sub-phase p, for every vertex $v \in V$ and every exploration source $s \in CU_i$ with a p-bounded path to v in $G^{(k-1)}$, there is a tuple $(s, \hat{d}(v,s))$ in the estimates list $LCurrent_v$ such that

$$d^{(p)}_{G^{(k-1)}}(v,s) \leq \hat{d}(v,s) \leq (1+\zeta')^p \cdot d^{(p)}_{G^{(k-1)}}(v,s).$$

To accomplish this, in every sub-phase p, we search for every vertex $v \in V$, a *better* (smaller than the current value of $\hat{d}(v,s)$) estimate (if exists) of its $(2\beta+1)$-bounded distance to every source $s \in CU_i$, by keeping track of edges $e = (u,v)$ incident to v in $G^{(k-1)}$. In each of the $2\beta+1$ sub-phases, we make two passes through the stream. For a given vertex $v \in V$, an exploration source $s \in CU_i$ is called an *update candidate* of v in sub-phase p, if a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ is available in sub-phase p through some edge $e = (u,v)$ on the stream. (Recall that the current estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s')$ for some source $s' \in CU_i$ for which we do not yet have an entry in $LCurrent_v$ is ∞.) We go through the edge set $H^{(k-1)}$ offline at the end of every sub-phase and update all our estimates lists with the best available estimates in $H^{(k-1)}$.

In the first pass of sub-phase p, we identify for every $v \in V$, all of v's update candidates in sub-phase p. All of these update candidates are added to a list called the *update list* of v, denoted $LUpdate_v$. Each element of $LUpdate_v$ is a tuple $(s, range, r)$, where s is the ID of an exploration source in CU_i for which a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ is available, $range$ is the distance range $I = (low, high]$ in which the better estimate is available, and r is the number of vertices $u \in \Gamma_G(v)$, such that $\hat{d}(u,s) + \omega(u,v) \in range$. The second pass of sub-phase p uses the update list of every vertex $v \in V$ to find a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$, for every update candidate s in $LUpdate_v$. The new better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ for every source s in $LUpdate_v$ is then used to update the estimates list $LCurrent_v$ of v.

First pass of sub-phase p of phase i. By Lemma 6,, the number of update candidates of v in any sub-phase of interconnection step of phase i is at most $c_1' \cdot \ln n \cdot deg_i$ whp. (Recall that all the explorations are restricted to distance at most $\delta_i'/2$.) We denote $\mathcal{N}_i = c_1' \cdot \ln n \cdot deg_i$

and $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$, where $c_4 \geq 1$ is a sufficiently large positive constant. At a high level, in the first pass of every sub-phase, we want to recover, for every vertex $v \in V$, a vector (containing sources of explorations that visit v in sub-phase p) with at most $\mathcal{N}_i$ elements in its support. In other words, we want to perform an s-sparse recovery for every vertex $v \in V$, where $s = \mathcal{N}_i$. We do so by performing multiple simultaneous invocations of a procedure called *FindNewCandidate*, for every $v \in V$. The pseudocode for procedure *FindNewCandidate* is given in Algorithm 1. The procedure *FindNewCandidate* enables us to sample an update candidate s of v (if exists), with a better (than the current) estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in a specific distance range.

Algorithm 1 Pseudocode for procedure $FindNewCandidate$.

Procedure FindNewCandidate(v, h, I)

▷ Initialization

$slots \leftarrow \emptyset$ ▷ An array with $\lambda = \lceil \log\ n \rceil$ elements indexed from 1 to λ.

▷ Each element of slots is a tuple $(sCount, sNames)$. For a given index $1 \leq \tau \leq \lambda$, fields $sCount$ and $sNames$ of $slots[\tau]$ can be accessed as $slots[\tau].sCount$ and $slots[\tau].sNames$, respectively.

▷ $slots[\tau].sCount$ counts the update candidates u seen by v with hash values $h(u) \in [2^\tau]$. It is set to 0 initially.

▷ $slots[\tau].sNames$ is an encoding of the names of candidate sources seen by v with hash values in $[2^\tau]$. It is set to ϕ initially.

▷ Update Stage

while (there is some update $(e_t,\ eSign_t,\ eWeight_t)$ in the stream) **do**
  if (e_t is incident on v and some $u \in V$) **then**
    for each $(s, \hat{d}(u,s)) \in LCurrent_u$ **do**
      if $((\hat{d}(u,s) + eWeight_t) \in I$ and
        $\hat{d}(u,s) + eWeight_t < \hat{d}(v,s))$ **then**
        $\tau \leftarrow \lceil \log h(s) \rceil$
        repeat ▷ Update $slots[\tau]$ for all $\lceil \log h(s) \rceil \leq \tau \leq \lambda$
          $slots[\tau].sCount \leftarrow slots[\tau].sCount + eSign_t$
          $slots[\tau].sNames \leftarrow slots[\tau].sNames + \nu(s) \cdot eSign_t$
          ▷ The function ν is described in Section 2.3.
          ▷ The addition in line 12 is a vector addition.
          $\tau = \tau + 1$
        until $\tau > \lambda$

▷ Recovery Stage

if ($slots$ vector is empty) **then**
  return (ϕ, ϕ)
else if ($\exists$ index τ s.t. $\frac{slots[\tau].sNames}{slots[\tau].sCount} = \nu(s)$ for some s in V) **then**
  return $(s, slots[\tau].sCount)$
else
  return $(\perp, \perp)$

For every vertex $v \in V$, we divide the possible range of better estimates of v's $(2\beta+1)$-bounded distances to its update candidates, into sub-ranges on a geometric scale. We then invoke the procedure *FindNewCandidate* repeatedly in parallel to perform an $\mathcal{N}_i$-sparse recovery for v on every sub-range. Specifically, we divide the search space of potential better estimates, $[1, \delta'_i/2]$, into sub-ranges $I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$, for $j \in \{0, 1, \ldots, \gamma\}$, where $\gamma = \lceil \log_{1+\zeta'} \delta'_i/2 \rceil - 1$. For $j = 0$, we make the sub-range $I_0 = \left[(1+\zeta')^0, (1+\zeta')^1\right]$ closed to include the value 1. Note that we are only interested in distances at most $\delta'_i/2$. Therefore we restrict our search for distance estimates to the range $[1, \delta'_i/2]$.

In more detail, we make for for each $v \in V$ and for each sub-range I_j, $\mu_i = \Theta(\mathcal{N}_i \cdot \ln n)$ *attempts* in parallel. In a specific attempt for a given vertex v and a given sub-range I_j, we make a single call to procedure *FindNewCandidate* which samples an update candidate

s (if exists) of v with a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in the sub-range I_j. Henceforth, we will refer to an update candidate s of a vertex v with a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in a given distance range I, as the *update candidate of v in the range I*. A single call to procedure *FindNewCandidate* succeeds only with a constant probability. Hence multiple parallel calls are required to boost the probability of success. In the extended version we show that making μ_i parallel attempts ensures that a specific update candidate s for some input vertex v and a distance subrange I_j gets sampled whp (assuming that it exists).

The procedure *FindNewCandidate* takes as input the ID of a vertex, a hash function h chosen at random from a family of pairwise independent hash functions and an input range $I = (low, high]$. (The input range may be closed as well.) A successful invocation of *FindNewCandidate* for an input vertex v and a distance range I returns a tuple (s, c_s), where s is the ID of an update candidate of v in the range I, and c_s is the number of edges $(v,u) \in E$ such that $\hat{d}(u,s) + \omega(v,u) \in I$. If there is no update candidate of v in the input range I, procedure *FindNewCandidate* returns a tuple (ϕ, ϕ). If there are update candidates of v in the input range, but procedure *FindNewCandidate* fails to isolate an ID of such a candidate, it returns $(\perp, \perp)$. Before we start making our attempts in parallel, we sample uniformly at random a set of functions H_p ($|H_p| = \mu_i$) from a family of pairwise independent hash functions $h : \{1, \ldots, maxVID\} \to \{1, \ldots, 2^\lambda\}$, where $\lambda = \lceil \log maxVID \rceil = \lceil \log n \rceil$. Then, for every vertex $v \in V$ and every distance sub-range I_j, $j \in \{0, 1, \ldots, \gamma\}$, we make μ_i parallel calls to procedure *FindNewCandidate*(v, h, I_j), one call for each $h \in H_p$.

Procedure FindNewCandidate. Procedure *FindNewCandidate* uses the input hash function h to sample for the input vertex v, an update candidate of v in the input range I. Let $d^{(I)}_v$ be the number of update candidates of v in the input range I. If we knew the exact value of $d^{(I)}_v$, we could sample every new update candidate witnessed by v with probability $1/d^{(I)}_v$ to extract exactly one of them in expectation. However, all we know about $d^{(I)}_v$ is that it is at most deg_i in expectation (Lemma 6) and at most $O(deg_i \cdot \ln n)$ whp. We therefore sample every new update candidate seen by v on a range of probabilities. We use an array *slots* of λ elements, indexed by *slot-levels* from 1 to $\lambda = \lceil \log n \rceil$, to implement sampling on a range of probabilities. We want a given update candidate s to be sampled into slot-level τ with probability $1/2^{\lambda-\tau}$. When $d^{(I)}_v \approx 2^{\lambda-\tau}$, with a constant probability there is exactly one exploration source that gets mapped to $slots[\tau]$.

Each element of *slots* is a tuple $(sCount, sNames)$. The field $sCount$ of element at slot-level τ counts the new update candidates seen by v with hash values in $[2^\tau]$. It is set to 0 initially. The field $sName$ of element at slot-level τ is an encoding of the names of candidate sources seen by v with hash values in $[2^\tau]$. It is set to ϕ initially.

The update stage of the procedure for an input vertex v and an input distance range I (See lines 4-16 of Algorithm 1) proceeds as follows. For every update $(e_t,\ eSign_t,\ eWeight_t)$ to an edge e_t incident to v and some vertex u, we look at every exploration source s in the estimates list $LCurrent_u$ of u, (see line 6 of Algorithm 1) and check whether the distance estimate of v to s via edge $e_t = (v,u)$ is better than the current value of $\hat{d}(v,s)$, and whether it falls in the input distance range I. (See line 8 of Algorithm 1.) If this is the case, then, we sample s (on a range of probabilities) by updating the elements of *slots* array from slot-levels $\lceil \log h(s) \rceil$ to λ. (See Lines 11 and 12.) We use the CIS-based encoding scheme ν described in Section 2.3 to encode the names of the exploration sources we sample, and use Lemma 4 to check (See line 19 of Algorithm 1), if we have successfully isolated the ID of a single update candidate in the desired distance range.

We need to make sure that for some $1 \leq \tau \leq \lambda$, only one exploration source will get mapped to $slots[\tau]$. By Corollary 13 (see Appendix A), only one exploration source gets mapped to $slots[\tau]$ for $\tau = \lambda - \lceil \log d_v^{(I)} \rceil - 1$, with at least a constant probability. Therefore, a single call to *FindNewCandidate* succeeds with at least a constant probability. For a vertex $v \in V$, if there are no update candidates of v in sub-phase p, all the calls to procedure *FindNewCandidate* in all the attempts return (ϕ, ϕ). For every such vertex, we do not need to add anything to its update list $LUpdate_v$. At the end of the first pass, if no invocation of procedure *FindNewCandidate* returns as error, we extract for every vertex $v \in V$ and every distance range I_j ($j \in \{0, 1, \ldots, \gamma\}$), all the distinct update candidates of v in the range I_j sampled by μ_i attempts made for v and sub-range I_j. For a given update candidate s of v, let $j = j_{v,s}$ be the smallest index in $\{0, 1, \ldots, \gamma\}$, such that a tuple (s, c_s) (for some $c_s > 0$) is returned by a call to procedure $FindNewCandidate(v, h, I_j)$. We add a tuple (s, I_j, c_s) to the list of update candidates $LUpdate_v$ of v. Recall that the set $LUpdate_v$ of vertex v contains tuples $(s, range, r_s)$, where s is the ID of an update candidate of v, $range$ is the distance range in which a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, s)$ lies, and r is the number of edges $(u, v) \in \Gamma_G(v)$ such that $\hat{d}(u, s) + \omega(u, v) \in range$.

Second pass of sub-phase j of phase i. The second pass of sub-phase p starts with the update lists $LUpdate_v$ of every $v \in V$. We find for every tuple $(s, range, r)$ in $LUpdate_v$, a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, s)$ in the sub-range $range$, by invoking procedure *Guess-Distance* (described in Appendix B.1) $O(\log n)$ times. We sample uniformly at random a set of $c_1 \log_{7/8} n = O(\log n)$ pairwise independent hash functions H'_p from the family $h : \{1, \ldots, maxVID\} \to \{1, 2, \ldots, 2^\lambda\}$ ($\lambda = \lceil \log n \rceil$), to be used by invocations of procedure *GuessDistance*.

Note that the current estimate $\hat{d}(v, s)$ of input vertex v's distance to its update candidate s is either available in its estimates list $LCurrent_v$ or is implicitly set to ∞. The latter happens if v has not yet been visited by the exploration rooted at source s. At the end of the second pass, we have the results of all the invocations of procedure *GuessDistance*, for a given vertex v corresponding to the tuple $(s, range, r) \in LUpdate_v$. We update the corresponding tuple $(s, \hat{d}(v, s))$ in the estimates list $LCurrent_v$ of v with the minimum value returned by any invocation of *GuessDistance* for vertex v. If an entry corresponding to s is not present in the estimates list $LCurrent_v$ at this stage (i.e., $\hat{d}(v, s) = \infty$ as above), then we add a new tuple to the estimates list of v. Finally, the updates lists of all the vertices are cleared to be re-used in the next sub-phase. At the end of second pass of sub-phase p, we go through the edges of the lower level hopsets and check for each $v \in V$ whether a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, s)$ for any source $s \in CU_i$ is available through one of the hopset edges. If this is the case, then we update the estimates lists accordingly.

Finally, after $2\beta + 1$ sub-phases of the interconnection step of phase i, we go through the estimates list of every center $r_C \in CU_i$ to check for every center $r'_C \in CU_i$, whether, there is a tuple $(r'_C, \hat{d}(r_C, r'_C)) \in LCurrent_{r_C}$ and $\hat{d}(r_C, r'_C) \leq \delta'_i/2$. Then, for every such center r'_C found, we add an edge (r_C, r'_C) of weight $\hat{d}(r_C, r'_C)$ into hopset H_k.

Next, we analyze the properties of our final hopset H.

3.3 Size, Stretch and Hopbound Anlaysis

Size: The size of our hopset H is the same as that of the insertion-only algorithm of [16], since we follow the same criteria (as in [16]), when deciding which cluster centres to connect via a hopset edge during our construction. Thus, the overall size of the hopset produced by our construction is $O(n^{1+1/\kappa} \cdot \log \Lambda)$ in expectation.

Stretch and Hopbound. Recall that ϵ_k is the value such that the graph $G^{(k)}$ (which is a graph obtained by adding the edges of hopset $H^{(k)} = \bigcup_{k_0 \le j \le k} H_j$ to the input graph G) provides stretch at most $1+\epsilon_k$. Also, recall that $k_0 = \lfloor \log \beta \rfloor$ and $k_\Lambda = \lceil \log \Lambda \rceil$. Write $c_5 = 2$. We need the following lemma from [16] regarding the stretch of a single scale hopset H_k, $k \in [k_0, k_\Lambda]$ produced by the insertion-only algorithm. We refer the reader to Lemma 3.10 and preamble of Theorem 3.11 of [16] for the proof.

▶ **Lemma 7** ([16]). *Let $x, y \in V$ be such that $2^k \le d_G(x,y) \le 2^{k+1}$, then it holds that*

$$d^{(h_\ell)}_{G \cup H_k}(x,y) \le (1+\epsilon_{k-1})(1+16 \cdot c_5 \cdot \ell \cdot \epsilon) d_G(x,y), \tag{3}$$

and $h_\ell = O(\frac{1}{\epsilon})^\ell$ is the hopbound. (Here c_5 is a fixed constant.)

Rescaling. Define $\epsilon'' = 16 \cdot c_5 \cdot \ell \cdot \epsilon$. Therefore, the stretch of a single scale hopset H_k, $k \in [k_0, k_\Lambda]$, produced by the insertion-only algorithm of [16] becomes $(1+\epsilon_{k-1})(1+\epsilon'')$. After rescaling, the hopbound h_ℓ becomes $O(\frac{\ell}{\epsilon''})^\ell$. Recall that $\ell = \ell(\kappa, \rho) = \lfloor \log(\kappa\rho) \rfloor + \lceil \frac{\kappa+1}{\rho\kappa} \rceil - 1 \le \log(\kappa\rho) + \lceil 1/\rho \rceil$, is the number of phases of our single-scale hopset construction. It follows that the hopbound of the insertion-only algorithm is

$$\beta_{EN} = O\left(\frac{\log \kappa\rho + 1/\rho}{\epsilon''}\right)^{\log \kappa\rho + 1/\rho}. \tag{4}$$

Observe that for $k = k_0$, graph $G^{(k-1)}$ is the input graph G itself, since H_k for all $k < k_0$ is an empty set. (See Section 3.1 for details.) Therefore, $1+\epsilon_{k-1}$ for $k = k_0$ is equal to 1. It follows therefore that the stretch $1+\epsilon_k = 1+\epsilon_{k_{EN}}$, of the insertion-only algorithm follows the following sequence: $1 + \epsilon_{k_{0\,EN}} = (1+\epsilon'')$ and for the higher scales, $1 + \epsilon_{k+1_{EN}} = (1+\epsilon'') \cdot (1 + \epsilon_{k_{EN}})$.

The stretch of our single scale hopset construction (Section 3.2) for any scale $(2^k, 2^{k+1}]$, $k_0 \le k \le k_\Lambda$ is $(1+\chi)$ times the stretch of the corresponding hopset produced by the insertion-only algorithm. We set $\chi = \epsilon''$. Incorporating the additional stretch incurred by our algorithm into the stretch analysis of [16], we get the following lemma about the stretch of our dynamic streaming algorithm.

▶ **Lemma 8.** *For $k \in [k_0, k_\Lambda]$, we have*

$$1 + \epsilon_{k_0} = (1+\epsilon'')^2$$
$$1 + \epsilon_k = (1+\epsilon'')^2(1+\epsilon_{k-1}) \text{ for } k > k_0$$

Observe that Lemma 8 implies that the overall stretch of our hopset H is at most $(1+\epsilon'')^{2\log\Lambda}$. Recall that the desired stretch of our hopset construction is $1+\epsilon'$ (see Section 3.1), where $\epsilon' > 0$ is an input parameter of our algorithm. We set $\epsilon'' = \frac{\epsilon'}{4 \cdot \log \Lambda}$, and it follows that our overall stretch is $\left(1 + \frac{\epsilon'}{4 \log \Lambda}\right)^{2 \log \Lambda} \le 1 + \epsilon'$. It follows that $\epsilon = \frac{\epsilon'}{64 \cdot c_5 \cdot \ell \cdot \log \Lambda}$.

Plugging in $\epsilon'' = \frac{\epsilon'}{4 \cdot \log \Lambda}$ in (4), we get the following expression for the hopbound of our dynamic streaming hopset:

$$\beta' = O\left(\frac{\log \Lambda}{\epsilon'}(\log \kappa\rho + 1/\rho)\right)^{\log \kappa\rho + 1/\rho}. \tag{5}$$

Also recall that we had defined $\beta = (\frac{1}{\epsilon})^\ell$ for using $2\beta+1$ as the hop-depth of our explorations. After the two rescaling steps as above, we get that $\beta = \beta'$.

3.4 Summary

A detailed analysis of the resource usage of the hopset construction algorithm is provided in extended version. In the extended version we also generalize our result to path-reporting hopsets, and integrate it with a weight reduction that eliminates most of the dependencies on $\log \Lambda$. We summarize the results below.

▶ **Theorem 9.** *For any n-vertex graph $G(V,E,\omega)$ with aspect ratio Λ, $2 \le \kappa \le (\log n)/4$, $1/\kappa \le \rho \le 1/2$ and $0 < \epsilon' < 1$, our dynamic streaming algorithm computes whp, a $(1+\epsilon',\beta')$ hopset H with expected size $O(n^{1+1/\kappa} \cdot \log n)$ and the hopbound β' given by*

$$\beta' = O\left(\frac{(\log \kappa\rho + 1/\rho)\log n}{\epsilon'}\right)^{\log \kappa\rho + 1/\rho}$$

It does so by making $O(\beta' \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream and using $O(n \cdot \log^3 n \cdot \log \Lambda)$ bits of space in the first pass and $O(\frac{\beta'}{\epsilon'} \cdot \log^2 1/\epsilon' \cdot n^{1+\rho} \cdot \log^5 n)$ bits of space (respectively $O(\frac{\beta'^2}{\epsilon'} \cdot \log^2 1/\epsilon' \cdot n^{1+\rho} \cdot \log^5 n)$ bits of space for path-reporting hopset) in each of the subsequent passes.

4 $(1+\epsilon)$-Approximate Shortest Paths

Consider the problem of computing $(1+\epsilon)$-approximate shortest paths (henceforth $(1+\epsilon)$-ASP) for all pairs in $S \times V$, for a subset S, $|S| = s$, of designated source vertices, in a weighted undirected n-vertex graph $G = (V,E,\omega)$ with aspect ratio Λ. Let $\epsilon, \rho > 0$ be parameters, and assume that $s = O(n^\rho)$. Our dynamic streaming algorithm for this problem computes a path-reporting $(1+\epsilon,\beta)$-hopset H of G with $\beta = O(\frac{\log n}{\epsilon\rho})^{1/\rho}$ using our hopset construction algorithm, with $\kappa = 1/\rho$. Once the hopset H has been computed, we conduct $(1+\epsilon)$-approximate Bellman-Ford explorations in $G \cup H$ to depth β from all the sources of S. (See the algorithm from Appendix B.) By Theorem 14, this requires $O(\beta)$ passes of the stream, and space $O(|S| \cdot n \cdot poly(\log n, \log \Lambda))$, and results in $(1+\epsilon)$-approximate distances $d^{(\beta)}_{G\cup H}(s,v)$, for all $(s,v) \in S \times V$. (Note that following every pass over G, we do an iteration of Bellman-Ford over the hopset H *offline*, as H is stored by the algorithm.) In addition, for every pair $(s,v) \in S \times V$, we also get the parent of v on the exploration rooted at source s. We compute the path $\pi_{G\cup H}(s,v)$ between s and v in graph $G \cup H$ from these parent pointers. The path-reporting property of our hopset H enables us to replace any hopset edge $e = (x,y) \in H$ on the path $\pi_{G\cup H}(s,v)$ with a corresponding path $\pi_G(x,y)$ in G. In the extended version we also argue that these replacements can be performed using $\tilde{O}(n^{1+\rho})$ space. By definition of the hopset, we have $d_G(s,v) \le d^{(\beta)}_{G\cup H}(s,v) \le (1+\epsilon) \cdot d_G(s,v)$, and the estimates $\hat{d}(s,v)$ computed by our approximate Bellman-Ford algorithm satisfy $d^{(\beta)}_{G\cup H}(s,v) \le \hat{d}(s,v) \le (1+\epsilon) \cdot d^{(\beta)}_{G\cup H}(s,v)$. Thus, we have, $d_G(s,v) \le \hat{d}(s,v) \le (1+\epsilon)^2 \cdot d_G(s,v)$. By rescaling $\epsilon' = 3\epsilon$, we obtain $(1+\epsilon)$-approximate $S \times V$ paths, the total space complexity of the algorithm is $O(n^{1+\rho} \cdot poly(\log n, \log \Lambda))$, and the number of passes is $poly(\log n)$. We derive the following theorem:

▶ **Theorem 10.** *For any parameters $\epsilon, \rho > 0$, and any n-vertex undirected weighted graph $G = (V,E,\omega)$ with polynomial in n aspect ratio, and any set $S \subseteq V$ of n^ρ distinguished sources, $(1+\epsilon)$-ASP for $S \times V$ can be computed in dynamic streaming setting in $\tilde{O}(n^{1+\rho})$ space and $\log^{\frac{1}{\rho}+O(1)} n = polylog(n)$ passes.*

References

1 Amir Abboud and Greg Bodwin. The 4/3 additive spanner exponent is tight. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 351–361, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897518.2897555.

2 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 459–467, USA, 2012. Society for Industrial and Applied Mathematics.

3 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'12, pages 5–14, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2213556.2213560.

4 Ingo Althofer, Gautam Das, David Dobkin, and Deborah A Joseph. Generating sparse spanners for weighted graphs. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1989.

5 Khanh Do Ba, Piotr Indyk, Eric Price, and David P. Woodruff. Lower bounds for sparse recovery. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1190–1197. SIAM, 2010. doi:10.1137/1.9781611973075.95.

6 Antal Balog and Imre Bárány. On the convex hull of the integer points in a disc. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, pages 162–165, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/109648.109666.

7 Surender Baswana. Streaming algorithm for graph spanners – single pass and constant processing time per edge. *Information Processing Letters*, 106(3):110–114, 2008. doi:10.1016/j.ipl.2007.11.001.

8 J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. doi:10.1016/0022-0000(79)90044-8.

9 Yi-Jun Chang, Martin Farach-Colton, Tsan-sheng Hsu, and Meng-Tsung Tsai. Streaming complexity of spanning tree computation. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 34:1–34:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.STACS.2020.34.

10 Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *STOC '94*, 1994.

11 Graham Cormode and D. Firmani. A unifying framework for ?0-sampling algorithms. *Distributed and Parallel Databases*, 32:315–335, 2013.

12 Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2), March 2011. doi:10.1145/1921659.1921666.

13 Michael Elkin. Distributed exact shortest paths in sublinear time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 757–770. ACM, 2017. doi:10.1145/3055399.3055452.

14 Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Improved weighted additive spanners. *CoRR*, abs/2008.09877, 2020. arXiv:2008.09877.

15 Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Trans. Algorithms*, 15(1), November 2018. doi:10.1145/3274651.

16 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 48(4):1436–1480, 2019. doi:10.1137/18M1166791.

17 Michael Elkin and Ofer Neiman. Near-additive spanners and near-exact hopsets, A unified view. *Bull. EATCS*, 130, 2020. URL: http://bulletin.eatcs.org/index.php/beatcs/article/view/608/624.

18 Michael Elkin and David Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 173–182, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/380752.380797.

19 Michael Elkin and Shay Solomon. Fast constructions of light-weight spanners for general graphs. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 513–525. SIAM, 2013. doi:10.1137/1.9781611973105.37.

20 Michael Elkin and Chhaya Trehan. $(1+\epsilon)$-approximate shortest paths in dynamic streams. *CoRR*, abs/2107.13309, 2021. arXiv:2107.13309.

21 Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\epsilon, \beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385, 2006. doi:10.1007/s00446-005-0147-2.

22 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming*, pages 531–543, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

23 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2009. doi:10.1137/070683155.

24 Manuel Fernandez, David P. Woodruff, and Taisuke Yasuda. Graph spanners in the message-passing model. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 77:1–77:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ITCS.2020.77.

25 Arnold Filtser, Michael Kapralov, and Navid Nouri. *Graph Spanners by Sketching in Dynamic Streams and the Simultaneous Communication Model*, pages 1894–1913. SIAM, 2021. doi:10.1137/1.9781611976465.113.

26 S. Ganguly. Counting distinct items over update streams. *Theor. Comput. Sci.*, 378:211–222, 2007.

27 David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015. arXiv:1509.06464.

28 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 725–736, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

29 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 489?498, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897518.2897638.

30 Piotr Indyk, Eric Price, and David P. Woodruff. On the power of adaptivity in sparse recovery. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 285–294. IEEE Computer Society, 2011. doi:10.1109/FOCS.2011.83.

31 Vojtěch Jarník. Über die gitterpunkte auf konvexen kurven. *Mathematische Zeitschrift*, 24:500–518, 1926.

32 Michael Kapralov and David Woodruff. Spanners and sparsifiers in dynamic streams. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 272–281, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2611462.2611497.

33 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. *CoRR*, abs/1502.03320, 2015. arXiv:1502.03320.

34 Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997. doi:10.1006/jagm.1997.0888.

35 Felix Lazebnik and Vasiliy A. Ustimenko. Some algebraic constructions of dense graphs of large girth and of large size. In Joel Friedman, editor, *Expanding Graphs, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, May 11-14, 1992*, volume 10 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 75–93. DIMACS/AMS, 1992. doi:10.1090/dimacs/010/07.

36 Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014. doi:10.1145/2627692.2627694.

37 Morteza Monemizadeh and David P. Woodruff. 1-pass relative-error $\mathrm{l_p}$-sampling with applications. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 1143–1160. SIAM, 2010. doi:10.1137/1.9781611973075.92.

38 David Peleg and Alejandro A. Schaffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. doi:10.1002/jgt.3190130114.

A Hash Functions

A hash function h maps elements from a given input domain to an output domain of bounded size. Ideally, we would like to draw our hash function randomly from the space of all possible functions on the given input/output domain. However, since we are concerned about the space used by our algorithm, we will rely on hash functions with limited independence. A family of functions $H = \{h : \mathcal{U} \to [m]\}$, from a universe $\mathcal{U}$ to $[m]$, for some positive integers m and k, is said to be *k-wise independent*, if it holds that, when h is chosen uniformly at random from H then for any k distinct elements $x_1, x_2, \cdots, x_k \in \mathcal{U}$, and any k elements $z_1, z_2, \cdots, z_k \in [m]$, $x_1, x_2, \cdots, x_k$ and mapped by h to $z_1, z_2, \cdots, z_k$ with probability $1/m^k$, i.e., as if they were perfectly random. Such functions can be described more compactly, but are sufficiently random to allow formal guarantees to be proven. The following lemma summarizes the space requirement of limited independence hash functions:

▶ **Lemma 11** ([8]). *A function drawn from a family of k-wise independent hash functions can be encoded in $O(k \log n)$ bits.*

Specifically, we will be using *pairwise independent* hash functions.

The following lemma, a variant of which has also been proved in [27, 33] in a different context, is proved in the extended version.

▶ **Lemma 12.** *Let $h : \mathcal{U} \to [2^\lambda]$ be a hash function sampled uniformly at random from a family of pairwise independent hash functions $\mathcal{H}$. If we use h to hash elements of a given set $\mathcal{S} \subseteq \mathcal{U}$ such that $|\mathcal{S}| = s$, then a specific element $d \in \mathcal{S}$ hashes to the set $[2^t], t = \lambda - \lceil \log s \rceil - 1$ and no other element of $\mathcal{S}$ does so with probability at least $\frac{1}{8s}$.*

Lemma 12 implies the following corollary:

▶ **Corollary 13.** *Let $h : \mathcal{U} \to [2^\lambda]$ be a hash function sampled uniformly at random from a family of pairwise independent hash functions $\mathcal{H}$. If we use h to hash elements of a given set $\mathcal{S} \subseteq \mathcal{U}$ with $|\mathcal{S}| = s$, then exactly one element in $\mathcal{S}$ hashes to the set $[2^t]$, $t = \lambda - \lceil \log s \rceil - 1$, with probability at least $\frac{1}{8}$.*

B Approximate Bellman-Ford Explorations

In this section, we describe an algorithm for performing a given number of iterations of an approximate B-F (Bellman-Ford) exploration from a given subset $S \subseteq V$ of *source* vertices in a weighted undirected graph $G(V, E, \omega)$ with aspect ratio Λ. For a given vertex $v \in V$ and a set $S \subseteq V$, the t-bounded distance between v and S in G, denoted $d_G^{(t)}(v, S)$, is the length of a shortest t-bounded path between v and some $s \in S$ (See Definition 3) such that $d_G^{(t)}(v, s) = \min\{d_G^{(t)}(s', v) \mid s' \in S\}$.

Given an n-vertex weighted graph $G(V, E, \omega)$, a set $S \subseteq V$ of vertices, an integer parameter $\eta > 0$ and an error parameter $\zeta \geq 0$, an (η, ζ)-B-F exploration of G rooted at S outputs for every vertex $v \in V$, a $(1+\zeta)$-approximation of its η-bounded distance to to the set S.

Throughout the execution of our algorithm, we maintain two variables for each vertex $v \in V$. One of them is a current estimate of v's η-bounded distance to set S, denoted $\hat{d}(v)$, and the other is the ID of v's neighbour through which it gets its current estimate, denoted $\hat{p}(v)$, and called the *parent* of v. We start by initializing $\hat{d}(s) = 0$, $\hat{p}(s) = \perp$, for each $s \in S$ and $\hat{d}(v) = \infty$, $\hat{d}(v) = \perp$ for each $v \in V \setminus S$. As the algorithm proceeds, $\hat{d}(v)$ and $\hat{p}(v)$ values of every vertex $v \in V \setminus S$ are updated to reflect the current best estimate of $d_G^{(\eta)}(v, S)$. The final value of $\hat{d}(v)$ for each $v \in V$ is such that $d_G^{(\eta)}(v, S) \leq \hat{d}(v) \leq (1+\zeta) \cdot d_G^{(\eta)}(v, S)$, and the final value of $\hat{p}(v)$ for each $v \in V$ contains the ID of v's parent on the forest spanned by (η, ζ)-B-F exploration of G rooted at the set S. The algorithm proceeds in phases, indexed by p, $1 \leq p \leq \eta$. We make one pass through the stream in each phase.

Phase p. In every phase, we search for every vertex $v \in V \setminus S$, a *better* (smaller than the current value of $\hat{d}(v)$) estimate (if exists) of its η-bounded distance to the set S, by keeping track of updates to edges $e = (v, u)$ incident to v. Specifically, we divide the search space of potential better estimates, $[1, 2 \cdot \Lambda]$, into sub-ranges $I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$, for $j \in \{0, 1, \ldots, \gamma\}$, where $\gamma = \lceil \log_{1+\zeta'} 2 \cdot \Lambda \rceil - 1$ and ζ' is set to $\zeta/2\eta$ for technical reasons. For $j = 0$, we make the sub-range $I_0 = \left[(1+\zeta')^0, (1+\zeta')^1\right]$ closed to include the value 1. Recall that we are doing a $(1+\zeta)$-approximate B-F exploration (and not an exact one). Due to this, some of the better estimates we get in a given phase may be between Λ and $(1+\zeta) \cdot \Lambda \leq 2 \cdot \Lambda$, where Λ is the aspect ratio of the input graph. We therefore keep our search space from 1 to 2Λ instead of Λ. In more detail, we make for for each $v \in V \setminus S$, γ *guesses*, one for each sub-range. In a specific guess for a vertex v corresponding to sub-range $\left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$ for some j, we make multiple simultaneous calls to a randomized procedure called *GuessDistance* which samples an edge (if exists) between v and some vertex u such that

$$\hat{d}(u) + \omega(v, u) \in I_j.$$

The exact number of calls we make to procedure *GuessDistance* in each guess will be specified later in the sequel. The smallest index $j \in [0, \gamma]$, for which the corresponding guess denoted $Guess_v^{(j)}$ successfully samples an edge which gives a distance estimate better than the current estimate of v, is chosen to update $\hat{d}(v)$.

The pseudocode for procedure *GuessDistance* is given in Algorithm 2.

The procedure *GuessDistance* takes as input the ID of a vertex, a hash function h chosen at random from a family of pairwise independent hash functions and an input range $I = (low, high]$. The input range may be closed as well.

A successful invocation of procedure *GuessDistance* for an input vertex x and input range I, returns a tuple $(dist, parent)$, (if there is at least one edge (x, y) in G such that $\hat{d}(y) + \omega(x, y) \in I$, and (∞, ∞) otherwise), where $dist$ is an estimate of x's η-bounded

Algorithm 2 Pseudocode for Procedure *GuessDistance*.

Procedure GuessDistance(x, h, I) ▷ Initialization
$slots \leftarrow \emptyset$ ▷ An array with λ elements indexed from 1 to λ, where $\lambda = \lceil \log n \rceil$.
▷ Each element of slots is a tuple $(xCount, xDist, xName)$. For a given index $1 \leq \tau \leq \lambda$, fields $xCount$, $xDist$ and $xName$ of $slots[\tau]$ can be accessed as $slots[\tau].xCount$, $slots[\tau].xDist$ and $slots[\tau].xName$, respectively.
▷ $slots[\tau].xCount$ is the number of sampled edges (x, y) with $h(y) \in [2^\tau]$. Initially, it is set to 0.
▷ $slots[\tau].xDist$ is the distance estimate for x provided by an edge (x, y) with $h(y) \in [2^\tau]$. Initially, it is set to 0.
▷ $slots[\tau].xName$ is encoding of the names of the endpoints y of sampled edges (x, y) with $h(y) \in [2^\tau]$. Initially, it is set to ϕ.
▷ Update Stage
while (there is some update $(e_t,\ eSign_t,\ eWeight_t)$ in the stream) **do**
  if (e_t is incident on x and some y such that $\hat{d}(y) + eWeight_t \in I$) **then**
    $\tau \leftarrow \lceil \log h(y) \rceil$
    repeat ▷ Update $slots[\tau]$ for all $\lceil \log h(y) \rceil \leq \tau \leq \lambda$.
      $slots[\tau].xCount \leftarrow slots[\tau].xCount + eSign_t$
      $slots[\tau].xDist \leftarrow slots[\tau].xDist + (\hat{d}(y) + eWeight_t) \cdot eSign_t$
      $slots[\tau].xName \leftarrow slots[\tau].xName \bigoplus name(y)$ ▷ $\bigoplus$ stands for bitwise XOR.
      $\tau = \tau + 1$
    until $\tau > \lambda$
▷ Recovery Stage
if ($slots$ array is empty) **then**
  return (∞, ∞)
else if ($\exists$ index $\tau \mid slots[\tau].xCount = 1$) **then**
  return $(slots[\tau].xDist, slots[\tau].xName)$
else
  return $(\perp, \perp)$

distance to the set S in the range I, and *parent* is the *parent* of x in the forest spanned by (η, ζ)-B-F exploration of G rooted at the set S. The procedure *GuessDistance* may fail to return (with a constant probability) a distance estimate in the desired range, even when such an estimate exists. It returns an error, denoted by $(\perp, \perp)$, in that case.

Before we start making calls to procedure *GuessDistance*, we sample uniformly at random a set of functions H_p of size $c_1 \log_{8/7} n$ from a family of pairwise independent hash functions $h : \{1, \ldots, maxVID\} \to \{1, \ldots, 2^\lambda\}$, where $\lambda = \lceil \log n \rceil$ and c_1 is an appropriate constant. For every guess for a given vertex $x \in V \setminus S$ and a given subrange I_j, we make $|H_p|$ parallel calls to procedure *GuessDistance*, one for each $h \in H_p$, to get an estimate of $d_G^{(\eta)}(x, S)$ in the given subrange. The multiple parallel calls are required since a single call to procedure *GuessDistance* succeeds only with a constant probability, while we need to succeed with high probability.

Additionally, before we start the phase p, we create for each $v \in V \setminus S$, a copy $\hat{d}'(v)$ of its current distance estimate $\hat{d}(v)$. Any update to the distance estimate of a vertex v during phase p is made to its *shadow* distance estimate $\hat{d}'(v)$. On the other hand, the variable $\hat{d}(v)$ for vertex $v \in V \setminus S$ remains unchanged during the execution of phase p. At the end of phase p, we update $\hat{d}(v)$ as $\hat{d}(v) = \hat{d}'(v)$. The purpose of using the shadow variable is to avoid any issues arising due to simultaneous reading from and writing to the distance estimate variable of a vertex by multiple parallel calls to procedure *GuessDistance*.

B.1 Procedure GuessDistance

For a given vertex x, and a given distance range I, let $y \in \Gamma_G(x)$ be such that

$$\hat{d}(y) + \omega(x,y) \in I.$$

In what follows, we will refer to such a vertex $y \in \Gamma_G(x)$ as a *candidate neighbour* and the corresponding edge (x,y) as a *candidate edge* in the range I. For a given vertex x, let $c_x^{(p,j)}$ be the number of candidate neighbours of x in the sub-range I_j. A call to procedure *GuessDistance* for vertex x with input range $I = I_j$ works by sampling a *candidate* neighbour with probability $\frac{1}{c_x^{(p,j)}}$. We use the input hash function h to assign hash values to the candidate edges in the range $\{1,\ldots,2^\lambda\}$, where $\lambda = \lceil \log n \rceil$. We only know an upper bound of n and not the exact value of $c_x^{(p,j)}$. Therefore, we try to guess $c_x^{(p,j)}$ on a geometric scale of values $2^{\lambda-\tau}$, $\tau = 1,2,\ldots,\lambda$, and sample every candidate neighbour on a range of probabilities corresponding to our guesses of $c_x^{(p,j)}$.

To implement sampling on a range of probabilities, we use an array *slots* of λ elements indexed by *slot-levels* from 1 to λ. Every new candidate neighbour y witnessed by x is assigned a hash value $h(y)$ by h. In every element of *slots*, we maintain a tuple $(xCount, xDist, xName)$, and $xCount$, $xDist$ and $xName$ of $slots[\tau]$ can be accessed as $slots[\tau].xCount$, $slots[\tau].xDist$ and $slots[\tau].xName$, respectively. The variable $xCount \in \mathbb{Z}$ at slot-level τ maintains the number of candidate neighbours with hash values in $[2^\tau]$. It is initialized to 0 at the beginning of the stream. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^\tau]$ appears on the stream, $slots[\tau].xCount$ is updated by adding the $eSign_t$ value of e_t to its current value. The variable $xDist$ at slot-level τ is an estimate of η-bounded distance of x limited to the input distance range I provided by edge (x,y) with $h(y) \in [2^\tau]$. Initially, it is set to 0. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^\tau]$ appears on the stream, $slots[\tau].xDist$ is updated by adding the value of the expression $(\hat{d}(y) + eWeight_t) \cdot eSign_t$ to its current value. (Recall that it is initialized as 0.) The variable $xName$ is encoding of the names of endpoints y of the sampled edges (x,y) with $h(y) \in [2^\tau]$. It is set to ϕ initially. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^\tau]$ appears on the stream, $slots[\tau].xName$ is updated by performing a bitwise XOR of its current value with $name(y)$.

At the end of the stream, if the *slots* array is empty, then there are no candidate neighbours in $\Gamma_G(x)$ and the procedure *GuessDistance* returns (∞,∞). If there is a slot-level τ such that $slots[\tau].xCount = 1$, then only one candidate neighbour is mapped to slot-level τ. In this case, $slots[\tau].xDist$ gives us an estimate of x's η-bounded distance to the set S in the input distance range I, and $slots[\tau].xName$ gives us the name of x's parent on the forest spanned by the (η,ζ)-B-F exploration of G rooted at set S. Indeed, if no smaller scale estimate will be discovered, the vertex recorded in $slots[\tau].xName$ will become the parent of x in the forest. The procedure *GuessDistance* returns $(slots[\tau].xDist, slots[\tau].xName)$. If the *slots* vector is not empty but there is no slot level with $xCount = 1$, then the procedure *GuessDistance* has failed to find a distance estimate in the input range I for x, and thus it returns an error $(\perp,\perp)$.

If the input vertex x has some candidate neighbours in the input distance range, we need to make sure that for some $1 \le \tau \le \lambda$, only one candidate neighbour will get mapped to $slots[\tau]$. By Corollary 13, only one of the $c_x^{(p,j)}$ candidate neighbours gets mapped to the set $[2^\tau]$, for $\tau = \lambda - \lceil \log c_x^{(p,j)} \rceil - 1$, with at least a constant probability. Therefore, a single invocation of procedure *GuessDistance* for a given vertex x and a given distance range succeeds with at least a constant probability. Since we are running $|H_p|$ parallel invocations of procedure *GuessDistance* for a given input vertex x and a given distance range I, we pick the output of a successful invocation of procedure *GuessDistance* as an estimate for x in the

input range. In the case that all the invocations of *GuessDistance* in a guess return an error, the algorithm terminates with an error. In the extended version, we show that when the set H_p is appropriately sized, the event of all the invocations of procedure *GuessDistance* in a given guess failing has a very low probability.

Once all the $\gamma = O(\frac{\log \Lambda}{\zeta'})$ guesses for a given vertex x have completed their execution without failure, we pick the smallest index j for which the corresponding guess $guess_x^{(j)}$ has returned a finite (non-failure) value, and compare this value with $\hat{d}(x)$. If this value gives a better estimate than the current value of $\hat{d}(x)$, we update the corresponding shadow variable $\hat{d}'(x)$, and the parent variable $\hat{p}(x)$. At the end of phase p, if the algorithm has not terminated with an error, for every vertex $x \in V \setminus S$, we update its current distance estimate variable with the value in the corresponding shadow variable as $\hat{d}(x) = \hat{d}'(x)$.

A detailed analysis of the algorithm is available in the extended version. The following theorem summarizes the results.

▶ **Theorem 14.** *For a sufficiently large positive constant c, given an integer parameter η, an error parameter ζ, an input graph $G(V, E, \omega)$, and a subset $S \subseteq V$, our distance exploration algorithms performs, with probability at least $1 - \frac{1}{n^c}$, a $(1+\zeta)$-approximate Bellman-Ford exploration of G rooted at the set S to depth η, and outputs for every $v \in V$, an estimate $\hat{d}(v)$ of its distance to set S and v's parent $\hat{p}(v)$ on the forest spanned by this exploration such that $d_G^{(\eta)}(v, S) \leq \hat{d}(v) \leq (1+\zeta) \cdot d_G^{(\eta)}(v, S)$ in η passes through the dynamic stream using*

$$O_c(\eta/\zeta \cdot n \cdot \log^2 n \cdot \log \Lambda(\log n + \log \Lambda)) \text{ space in every pass.}$$

Note also that the space used by the algorithm on different passes can be reused, i.e., the total space used by the algorithm is $O_c(\eta/\zeta \cdot n \cdot \log^2 n \cdot \log \Lambda(\log n + \log \Lambda))$.