Efficient Construction of Directed Hopsets and Parallel Approximate Shortest Paths

Nairen Cao
nairen@ir.cs.georgetown.edu
Georgetown University
Washington D.C., USA

Jeremy T. Fineman
jfineman@cs.georgetown.edu
Georgetown University
Washington D.C., USA

Katina Russell
katina.russell@cs.georgetown.edu
Georgetown University
Washington D.C., USA

ABSTRACT

The approximate single-source shortest-path problem is as follows: given a graph with nonnegative edge weights and a designated source vertex s, return estimates of the distances from s to each other vertex such that the estimate falls between the true distance and $(1+\epsilon)$ times the distance. This paper provides the first nearly work-efficient parallel algorithm with sublinear span (also called depth) for the approximate shortest-path problem on *directed* graphs. Specifically, for constant ϵ and polynomially-bounded edge weights, our algorithm has work $\tilde{O}(m)$ and span $n^{1/2+o(1)}$. Several algorithms were previously known for the case of *undirected* graphs, but none of the techniques seem to translate to the directed setting.

The main technical contribution is the first nearly linear-work algorithm for constructing hopsets on directed graphs. A (β, ϵ)-hopset is a set of weighted edges (sometimes called shortcuts) which, when added to the graph, admit β-hop paths with weight no more than $(1+\epsilon)$ times the true shortest-path distances. There is a simple sequential algorithm that takes as input a directed graph and produces a linear-cardinality hopset with $\beta = \tilde{O}(\sqrt{n})$, but its running time is quite high—specifically $\tilde{O}(m\sqrt{n})$. Our algorithm is the first more efficient algorithm that produces a directed hopset with similar characteristics. Specifically, our sequential algorithm runs in $\tilde{O}(m)$ time and constructs a hopset with $\tilde{O}(n)$ edges and $\beta = n^{1/2+o(1)}$. A parallel version of the algorithm has work $\tilde{O}(m)$ and span $n^{1/2+o(1)}$.

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**; **Shortest paths**.

KEYWORDS

Parallel algorithm, hopsets, shortest paths, shortcuts.

ACM Reference Format:
Nairen Cao, Jeremy T. Fineman, and Katina Russell. 2020. Efficient Construction of Directed Hopsets and Parallel Approximate Shortest Paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC '20), June 22–26, 2020, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357713.3384270

STOC '20, June 22–26, 2020, Chicago, IL, USA

ACM ISBN 978-1-4503-6979-4/20/06…$15.00
https://doi.org/10.1145/3357713.3384270

1 INTRODUCTION

The single-source shortest-path problem on graphs with nonnegative edge weights is notoriously difficult to parallelize.[1] In the sequential setting, the classic solution has running time $O(m + n \log n)$ [12], where throughout n denotes the number of vertices and m the number of edges. Given that the sequential solution has nearly linear runtime, an ideal parallel algorithm would run in $\tilde{O}(m/p)$ parallel time on p processors (for large p), where the $\tilde{O}$ notation suppresses logarithmic factors. Achieving such a bound requires a parallel algorithm with nearly linear work and strongly sublinear span; the ***work*** of a parallel algorithm is the total number of primitive operations, and its ***span*** is the length of the longest chain of sequential dependencies or equivalently the limit of the parallel time as p approaches infinity. The exact version of the shortest-path problem is well-studied (see e.g. [2, 3, 8, 15, 17, 19, 20]), but no ideal parallel solutions exist, especially when the graph is sparse.[2] Even for the simplest case of a unweighted, undirected graph, all algorithms to date either have linear span, meaning that they are inherently sequential, or they reduce the span by increasing the work. For example, when tuned to achieve span of $\tilde{O}(\sqrt{n})$, Spencer's algorithm [19] has work $\tilde{O}(m + n^2)$ and Ullman and Yannakakis's algorithm [20] has work $\tilde{O}(m\sqrt{n})$.

For undirected graphs at least, there has been more success on the approximate version of the problem. In the approximate shortest-path problem with source vertex s, the algorithm must output for all vertices v an estimate d_v on the shortest path-distance such that $dist(s,v) \le d_v \le (1+\epsilon)dist(s,v)$, where $dist(s,v)$ is the shortest-path distance from s to v. Several algorithms have been designed for this approximate problem on undirected graphs, see e.g., [1, 5, 9, 10, 16, 18]. Most of the results exhibit a tradeoff between work and span (requiring either superlinear work or polynomial span), but recent breakthroughs show that it is possible to simultaneously achieve $\tilde{O}(m)$ work and $O(\text{poly}(\log n))$ span [1, 16].

A natural question is whether it is possible to achieve nearly linear work and sublinear span for approximate shortest paths on *directed* graphs. This paper answers the question in the affirmative: we present an algorithm for directed graphs with $\tilde{O}(m)$ work and span $n^{1/2+O(1/\log\log n)}$.

Hopsets. While it is unknown how to efficiently compute shortest paths in parallel on general directed graphs, it is known how to find approximate shortest paths if the shortest paths consist of relatively

[1] Perhaps counter-intuitively, achieving at least a reasonable level of parallelism when the weights are both positive and negative is easier. This is in part because algorithms for the general case have roughly the same inherent sequential dependencies but with far more work that can be parallelized in each step.
[2] Achieving parallelism $p = O(m/n)$ is fairly straightforward.

few hops. Specifically, Klein and Subramanian's weighted breadth-first search algorithm [15] gives a $(1+\epsilon)$ approximation of β-hop distances in work $\tilde{O}(m)$ and span $\tilde{O}(\beta/\epsilon)$. Given this algorithm, a natural approach is to first preprocess the graph to produce a new graph whose β-hop distances are not too much larger than the actual unbounded distances; the preprocessing step amounts to finding a good hopset.

A (β, ϵ) ***hopset*** H is a set of weighted edges that, when added to the original graph, approximates the shortest-path distances by paths of at most β hops, where β is called the ***hopbound***. Formally, let $G = (V, E)$ be the original graph and $G' = (V, E \cup H)$ be the graph with the hopset edges included. H is a (β, ϵ) hopset if and only if (1) for all edges $(u, v) \in H$, the weight $w(u, v)$ of the edge is no lower than the shortest-path distance in G, i.e., $w(u, v) \geq dist_G(u, v)$, and (2) for every $u, v \in V$ there exists a path p from u to v in G' comprising at most β hops such that $w(p) \leq (1+\epsilon) dist_G(u, v)$. (The first constraint implies that $w(p) \geq dist_G(u, v)$.) Although hopsets were first formalized by Cohen [5], they were used implicitly in many of the prior algorithms. Most algorithms for constructing hopsets, including the one in this paper, are randomized and there is some small chance that the weight of some β-hop path will be too high.

There are several features characterizing the quality of a hopset: the size or number of edges in the hopset, the hopbound β, the approximation quality ϵ, and the complexity of an algorithm for constructing the hopset. When $\epsilon = 0$, the hopset produced is an ***exact hopset***, meaning that the β-hop distances in the augmented graph are the true shortest-path distances.

There is a simple folklore sequential algorithm for constructing an exact hopset with hopbound $\beta = \tilde{O}(\sqrt{n})$ and size $O(n)$. The algorithm is as follows. First sample each vertex with probability $O(1/\sqrt{n})$. Next, compute the single-source shortest-path distances from each sampled vertex to all other sampled vertices. For samples s_i and s_j, add to hopset H the edge (s_i, s_j) with weight $w(s_i, s_j) = dist(s_i, s_j)$. Since edges are only added between pairs of sampled vertices, the hopset trivially contains $O(n)$ edges with high probability. To analyze the hopbound, consider a shortest path from u to v. With high probability, the β hops nearest to u and β hops nearest to v each contain at least one sampled vertex, so the rest of the path can by bypassed using a hopset edge. Ullman and Yannakakis [20] and Klein and Subramanian [15] give parallel versions of this algorithm for the unweighted and integer-weighted cases, respectively.

The preceding algorithm gives an exact hopset with small size and reasonable hopbound, and it applies to directed graphs as well. The problem is that the construction time is too high: the sequential running time is $\tilde{O}(m\sqrt{n})$ to compute shortest paths from $\sqrt{n}$ sources.

For undirected graphs, when the exactness is relaxed and we are willing to accept a $(1+\epsilon)$ approximation, there exist linear-size hopsets with much smaller (subpolynomial) hopbound [10]. Moreover, there are more efficient algorithms [5, 9, 10, 18] for constructing the hopsets. The algorithms employ clustering techniques that strongly exploit the symmetry of distances in undirected graphs.

For directed graphs, a hopbound of $O(\sqrt{n})$ is still the best known for hopsets of linear size, even for approximate hopsets with large ϵ and ignoring construction cost. In fact, if $\epsilon \geq nW$ and all edge weights are at least one, then distances themselves become irrelevant—the problem reduces to the diameter-reduction or shortcutting problem: add edges to the graph, without changing the transitive closure, to reduce the unweighted directed diameter, i.e., the number of hops necessary to get from one vertex to another. It is yet unknown whether it is always possible to achieve diameter better than $O(\sqrt{n})$ when restricted to add at most n edges. In fact, there is a lower bound of $\Omega(n^{1/6})$ on the diameter [13], which implies a separation between the quality of hopsets on directed and undirected graphs. Revisiting construction cost, there was no more efficient algorithm known for any constant ϵ before the current paper.

Our results. This paper presents the first efficient algorithm for producing a hopset on directed graphs with sublinear hopbound. Specifically, our algorithm produces a $(\beta = n^{1/2+O(1/\log\log n)}, \epsilon)$ hopset with nearly linear size, which is close to matching the quality of the hopset produced by the highly inefficient folklore algorithm. For unweighted graphs (Sections 3–4), the hopset has size $\tilde{O}(n/\epsilon^2)$, and the algorithm runs in time $\tilde{O}(m/\epsilon^2)$. More generally for weighted graphs (Section 5), the hopset has size $\tilde{O}(n \log(nW)/\epsilon)$ and the algorithm runs in time $\tilde{O}(m \log(nW)/\epsilon^2)$, where W is the ratio between the maximum edge weight and the minimum strictly positive edge weight. The construction is successful with high probability, and failure is one sided—i.e., the result is always a hopset, but the question is whether it achieves the $(1+\epsilon)$ approximation.

Our parallel algorithm (Section 6) constructs a hopset with similar characteristics. The algorithm has work $\tilde{O}(m \log^2(nW)/\epsilon^4)$ and span $O(n^{1/2+O(1/\log\log n)}/\epsilon)$.

Using our parallel hopset construction then applying Klein and Subramanian's algorithm [15] to the augmented graph yields the first nearly work-efficient parallel algorithm for finding approximate single-source shortest paths on directed graphs with low span. More precisely, our algorithm has work $\tilde{O}(m \log(nW)/\epsilon^4)$ and span $O(n^{1/2+O(1/\log\log n)}/\epsilon)$.

1.1 Overview of Diameter Reduction

Our algorithm and analysis builds on recent breakthroughs on the diameter-reduction problem by Fineman [11] later improved by Jambulapati, Liu, and Sidford [14], henceforth referred to as the JLS algorithm. This section summarizes the previous algorithms and key aspects of the analyses, highlights the difficulties in extending the algorithms to hopsets, and gives an overview of our insights. The bulk of this section focuses on the sequential versions of the algorithms.

The diameter reduction problem is that of adding edges, or shortcuts, to a directed graph to reduce its unweighted diameter without changing the transitive closure. Fineman's algorithm [11] is the first nearly linear-time sequential algorithm with any nontrivial diameter reduction. Specifically, his algorithm runs in $\tilde{O}(m)$ time and creates $\tilde{O}(n)$ shortcuts that reduce the diameter of any directed graph to $\tilde{O}(n^{2/3})$, with high probability. The JLS algorithm [14] achieves a diameter of $n^{1/2+o(1)}$, also with nearly linear running time. Both algorithms also have parallel versions with span matching the diameter achieved to within logarithmic factors.

Our algorithm for hopsets most closely resembles the JLS algorithm for diameter reduction.

Previous algorithms for diameter reduction. Both Fineman's algorithm [11] and the JLS algorithm [14] operate roughly as follows. Select a random set of pivots x_i; how the pivots are selected varies across the two algorithms and is discussed more later. Next perform a graph search forwards and backwards from each pivot to identify the vertices reachable in either direction. Add shortcut edges between the pivots and all vertices reached, i.e., if a vertex u is reached in backward direction from pivot x_i, then the edge (u, x_i) is added. Next partition the vertices into groups according to the set of pivots that reach them. For example, a group could consist of those vertices reached by x_1 in the forward direction, x_3 in the backward direction, x_4 in both directions, and unreached by all other pivots. If a group is reached in both directions by the same pivot (as with the preceding example and pivot x_4), mark the group as done. Finally, recurse on the subgraph induced by each group that has not been marked as done.

The main difference between the algorithms is how pivots are selected. Fineman's algorithm [11] selects a single pivot uniformly at random. JLS [14] instead samples vertices to select a set of pivots. The algorithm is parameterized by a value k that controls the sampling probability; $k = \Theta(\text{poly}(\log n))$ is a good choice, so we shall assume as much going forward to simplify the statement of remaining bounds. Each vertex is a selected as a pivot with probability $k^{r+\Theta(1)}/n$, where r is the recursion depth. The probability of becoming a pivot thus increases by a factor of k with each level of recursion, and it is possible to select many pivots. Beyond achieving a better diameter, the JLS algorithm also has the advantage that the recursion depth is trivially limited to $\log_k n$. Increasing k impacts the total work as multiple overlapping searches are performed, which is why k should not be too large. We shall not discuss the analysis of the running time here, but suffice it to say that it is not hard to show that these sequential algorithms have $\tilde{O}(m)$ running time.

The diameter analysis starts by fixing any long s-to-t path P to analyze. The goal is to argue that with at least constant probability, the addition of shortcuts introduces a short-enough s-to-t path to the graph. The algorithm can be repeated to boost the success probability.

One of the key setup ideas is classifying vertices according to how they relate to the path P. We write $v \preceq P$ if it is possible to get from v to some vertex on P by following directed edges and $P \preceq v$ if it is possible to get from some vertex on P to v by following directed edges. A vertex v is an ***ancestor*** of P if $v \preceq P$ and $P \not\preceq v$. The vertex is a ***descendant*** of P if $v \not\preceq P$ and $P \preceq v$. It is a ***bridge*** if $v \preceq P$ and $P \preceq v$. The vertex is ***unrelated*** otherwise.

As the algorithm executes and partitions the graph, so too does it partition the path being analyzed. An execution can be modeled by a recursion tree where only the ***relevant subproblems***, i.e., those that contain subpaths of P, are included. The leaves of this relevant subproblem tree occur when at least one of the pivots is a bridge; if a bridge is selected, then edges are added between all vertices on the subpath and the bridge in both directions, meaning that the subpath has been shortened to two hops. The final path length from s to v is thus upper bounded by the number of leaves in the tree of relevant subproblems.

For the case of a single pivot as in Fineman's algorithm [11], it is not hard to see that a relevant subproblem gives rise to at most two recursive subproblems, and the two subproblems occur only if the pivot is an ancestor or descendant. For example, if the pivot is an ancestor, the path is partitioned at the first reachable vertex on the path. If an unrelated pivot is selected, there is only one relevant subproblem; informally, this case can be ignored in the single pivot case as tree nodes with a single child can be contracted. More generally, JLS show [14] that if t ancestors/descendants are selected, then the path is partitioned across at most $t + 1$ relevant subproblems.

A key component of the analysis is to show that the total number of ancestors and descendants is likely to decrease each time an ancestor or descendant pivot is selected. It thus becomes less and less likely to partition the path further and more likely to select a bridge. For concreteness, let us first consider a sketch of the intuition for the single-pivot case. Fineman [11] proves that if a random ancestor is selected as the pivot, then the total number of ancestors across both recursive subproblems reduces by a factor of $1/2$ in expectation. Similarly for descendants. We thus need roughly $(1/3)\lg n$ levels of recursion to reduce the total number of ancestors to $n^{2/3}$ and another $(1/3)\lg n$ levels to similarly reduce the number of descendants. At recursion depth $(2/3)\lg n$, there are thus at most $2^{(2/3)\lg n} = n^{2/3}$ subproblems and at most $O(n^{2/3})$ ancestors and descendants. Even if all of the remaining ancestors and descendants eventually become pivots, there can be at most $O(n^{2/3})$ leaves in the recursion tree, which yields the final path length.

If one could ensure that the algorithm always selects either zero or t related pivots, then one could easily extend Fineman's analysis to the multi-pivot case. In particular JLS prove [14] that with t random ancestor/descendant pivots, the total number of ancestors and descendants reduces by $c/(t + 1)$ in expectation, for some constant c. Consider the rth level of recursion assuming t related pivots are always selected. The number of subproblems is at most $(t + 1)^r$. The number of ancestors and descendants is upper bounded by $c^r n/(t + 1)^r$, which also upper bounds the number of leaves that could arise lower in the recursion tree. Setting $r = (1/2)\log_{t+1} n$ roughly balances these two terms and gives a path length of at most $\sqrt{n}c^{\log_{t+1} n} = n^{1/2+O(1/\log(t+1))}$.

Unfortunately, the algorithm is unaware of the path P, and it cannot ensure that t of the pivots are related to the path. Nevertheless it is still possible to obtain the same bound. The JLS analysis [14] adopts a bottom-up approach, solving a recurrence on the shortcutted path length for a given number of ancestors/descendants.

Parallel versions. The big challenge in parallelizing these algorithms is performing the graph searches used to partition the graph. To achieve low span both Fineman and JLS employ h-hop-limited searches, i.e., only identifying vertices reachable from the pivot within h hops. Fineman and JLS set h to $h = \tilde{\Theta}(n^{2/3})$ and $h = n^{1/2+o(1)}$, respectively. As noted previously, there are parallel algorithms implementing h-hop limited searches with $\tilde{O}(h)$ span [15]. Unfortunately, using hop-limited searches it is no longer immediately true that selecting t related pivots partitions the path into at most $t + 1$ subpaths, which was crucial for the analyses. To fix this issue, Fineman [11] and JLS [14] (1) only analyzes paths with length $\tilde{\Theta}(h)$, and (2) handle vertices near the boundary of the

search, called ***fringe vertices***, differently from other vertices. In doing so, they are able to achieve the $\leq t + 1$ relevant subproblems, though the details become significantly more complicated.

1.2 Overview of the Hopset Algorithm

A natural first step to extend the diameter-reduction algorithms to build hopsets is to add weights to any added shortcuts. Specifically, perform a shortest-path algorithm from each pivot and augment the shortcuts with weight equal to the shortest-path distances to each vertex. Our algorithm includes weights on shortcuts, but this change alone is not sufficient to achieve a good approximation.

The main challenge is that bridges do not necessarily make good pivots. Specifically, consider any bridge x for an s-to-t path. If x is selected as a pivot, then a 2-hop path is created from s to t, which is enough for the diameter-reduction problem. For hopsets, however, the weight of the path matters. If $dist(s, x) + dist(x, t) \gg dist(s, t)$, then the 2-hop path taking the shortcuts does not approximate the shortest-path distance. It may thus be necessary to continue recursing on subpaths in subproblems until better shortcuts are found.

In both prior analyses [11, 14], it is crucial that selecting a bridge acts as a base case to the recursion. Selecting a bridge that is too far away here, however, is not a base case. Moreover, it does not seem possible to argue that a far-away bridge yields any reasonable reduction on the number of ancestors or descendants in the resulting subproblems.

Our algorithm for hopsets. Our algorithm builds off the JLS algorithm, also parameterized by sampling parameter k, but with several key modifications. The goal is to circumvent the preceding challenge by ensuring, at least in effect, that shortcuts added to or from bridges are good enough for the approximation. We first summarize the differences in the algorithm before revisiting the analysis.

(1) **Pivots and shortcutters.** In the previous algorithms, pivots are used both to partition the graph and to add shortcuts. Here, we split the roles; we use some vertices, called ***pivots*** to establish the partition of the graph, and other vertices, called ***shortcutters***, to add edges to the hopset. Pivots are selected analogously to JLS, but we sample a larger set of shortcutters. More precisely, if a vertex becomes a pivot at recursion depth r, then it first becomes a shortcutter at recursion depth $r - f(\epsilon, n)$ for some function f. Larger f improves the approximation quality but increases the work of the algorithm.
(2) **Weighted shortcuts.** From each shortcutter s, we compute the single-source shortest paths from s to all other vertices in both the forwards (and backwards) directions. We then add the weighted edges (s, v) (and (v, s)) with weight $w(s, v) = dist(s, v)$ (and $w(v, s) = dist(v, s)$) to the hopset. Using weighted shortcuts is the obvious modification necessary for a hopset.
(3) **Decreasing distance-limited searches from pivots.** To establish the graph partition, we perform graph searches from each pivot as before, but the searches are now limited to a bounded distance. Moreover, the search distances decrease by a factor of $\lambda\sqrt{k}$ with each level of recursion, for constant λ. The initial distance is important—the algorithm only well-approximates paths if the initial search distance is similar to the shortest-path distance—so we run the algorithm at all relevant initial-distance scales.

It is worth noting that the distance-limited searches here are not analogous in purpose to the hop-limited searches used by the prior [11, 14] parallel algorithms for diameter reduction. (Our parallel version also imposes a hop limit.) Here the distance-limited searches are important even for the sequential algorithm in order to obtain a good approximation. Moreover, the distances decrease significantly with each level of recursion, whereas the hop-limited searches use roughly the same number of hops at all levels. Nevertheless, some of the technical machinery (e.g., fringe vertices) is similar.

Because our sequential algorithm for hopsets uses distance-limited searches, the details of both the algorithm and analysis are more complicated than the sequential algorithms for diameter reduction.

Key ideas of the analysis. Our analysis has two main novelties, summarized next. Note that the bounds stated here are correct in spirit but imprecise in that that they omit some lower-order terms in favor of conciseness.

For the following discussion, it is important to interpret the vertex classifications (ancestor, descendant, and bridge) to be with respect to the bounded distances, analogous to the hop-limited searches in prior work [11, 14]. For example, a vertex is only a bridge if it can reach the path in both directions by an appropriate distance-limited search.

The first technical contribution can be viewed as an alternative way of analyzing the JLS algorithm, but this version makes it easier to cope with the new features of the hopset algorithm. Specifically, we show that the number of subproblems increases by at most $O(\sqrt{k})$ on average with each level of recursion. For any constant in the big-O, it follows that there be at most $(O(\sqrt{k}))^r = (k^{1/2+O(1/\log k)})^r$ relevant subproblems at recursion depth r. Looking at the maximum recursion depth $r = \log_k n$ gives a direct bound of $n^{1/2+O(1/\log k)}$ on the number of relevant subproblems, and hence the length in hops of the shortcutted path.

Now consider what happens if we augment the JLS algorithm with decreasing distance-limited searches. Let $w(P)$ be the weight of the path P being analyzed, and assume that the initial search distance is roughly $w(P)$. The general issue when decreasing the search distance is that when searches do not reach the end of the path, the path may be partitioned into more pieces than desired.[3] We circumvent the issue by logically dividing any long paths into subpaths of length roughly $w(P)/(\lambda\sqrt{k}))^r$ (proportional to the search distance), where r is the recursion depth. In this way, the searches can now traverse the full length of the path. It is easy to see that there can be at most $O((\lambda\sqrt{k})^r)$ logical subproblems created. For large-enough λ, this term dominates the number of subproblems arising from the previous level of recursion, so we have a total of $O(\lambda^r k^{r/2})$ subproblems at recursion depth r. Again, this bound readily implies

[3] The use of "fringe vertices" suffices if the search distance is sufficiently long with respect to the path length. The new issue that arises here is that the search distance can be significantly shorter than the path length.

a hop bound for the shortcutted path of $n^{1/2+O(1/\log k)}$, albeit with a larger constant in the big-O.

The second new idea is in analyzing the approximation factor achieved by the hop set, which requires all three algorithmic modifications. Let us first consider only the shortcuts generated by "nearby" bridges. For an s-to-t subpath at recursion depth r, we say that a bridge x is nearby if $dist(s,x) + dist(x,t) = dist(s,t) + O((\epsilon/\log n)w(P)/(\lambda^r k^{r/2}))$. Since the total number of subproblems is $O(\lambda^r k^{r/2})$, shortcuts from nearby bridges contribute a total of $O(\epsilon w(P)/\log n)$ additive error to the path length. Summing across all $O(\log n)$ levels of recursion gives a total additive error of $O(\epsilon w(P))$, and hence a multiplicative error of $(1 + O(\epsilon))$.

The goal is thus to show that all bridges are effectively nearby bridges. This statement seems implausible, but we can achieve it by leveraging both the bounded search distance as well as the oversampling of shortcutters. In fact, for $\epsilon = \Omega(\log n)$, we can immediately see that all bridges are nearby—the additive error is bounded by twice the maximum search distance, i.e., $O(w(P)/(\lambda^r k^{r/2})) = O((\epsilon/\log n)w(P)/(\lambda^r k^{r/2}))$. We thus achieve a hopset with $\epsilon = O(\log n)$ even setting the shortcutters and pivots to be identical.

To achieve a better approximation, we leverage the oversampling of shortcutters. Observe that moving the shortcutters to a higher level of recursion can only improve the length in hops of the shortcutted path, as strictly more edges are added. To analyze quality of the approximation, we consider the recursion tree of relevant subproblems, but we now have a base case whenever a nearby bridge is selected as a shortcutter.

Since moving shortcutters higher in the recursion only helps, it suffices to show that the pivots selected in relevant subproblems are never bridges, i.e., that all shortcuts important to the hopbound also have small additive error. We prove the claim that pivots are never bridges by contradiction. Suppose that a pivot x is a bridge in a relevant subproblem at recursion depth r. Then it must be within a distance $O(w(P)/(\lambda^r k^{r/2}))$ of both the start and end of the path, as that is both the search distance and the path length. The additive error contributed by this bridge is thus at most $O(w(P)/(\lambda^r k^{r/2}))$. While x would not be considered a nearby bridge at level r, recall that x is first selected as a shortcutter at recursion depth $r - f(\epsilon, n)$. For appropriate choice of f, i.e., $(\lambda\sqrt{k})^{f(\epsilon,n)} = \Omega(\log n/\epsilon)$, x is a nearby bridge at depth $r - f(\epsilon, n)$, constituting a base case of the recursion. Thus the subproblem in which x is selected as a pivot is not a relevant subproblem.

2 PRELIMINARIES

A directed weighted graph is a pair (G, w) where $G = (V, E)$ is a graph and $w : E \to R$ is a weight function. In this paper, we treat w as an attribute of E. Hence, we refer G as the weighted graph and ignore w. For a weighted directed graph $G = (V, E)$, the number of vertices and edges are $|V| = n$ and $|E| = m$, respectively. For $e \in E$, we denote the weight as $w_E(e)$ and we write $w_E(e)$ as $w(e)$ for simplicity. If $e \notin E$, then $w(e) = +\infty$. If the graph is unweighted, then $w(e) = 1$ for all $e \in E$. For a subset $V' \subset V$, we denote the induced graph on V' as $G[V']$. For any vertices $u, v \in V$, define $dist_G^{(\beta)}(u, v)$ to be the minimum weight of a path from u to v containing at most β edges. If there is no path containing at most β edges from u to v, then $dist_G^{(\beta)}(u, v) = +\infty$. We also refer to $dist_G(u, v)$ as the shortest path distance from u to v. For a set of edges E and a constant c, we define $c \cdot E$ to be E where the weight of each edge in E is multiplied by c. For two sets of edges E and E', the union of E and E' is denoted as $E \cup E' = \{e | e \in E \text{ or } e \in E'\}$ and the weight of $e \in E \cup E'$ is the minimum weight of $w_E(e)$ and $w_{E'}(e)$, i.e, $w_{E\cup E'}(e) = min(w_E(e), w_{E'}(e))$. We assume the lightest non-zero edge weight is 1, and the heaviest edge weight is W. If the lightest non-zero edge weight $w(e)$ is less than 1, then all edges are scaled by $1/w(e)$.

Paths. A **path** $P = \langle v_0, v_1, \ldots v_\ell \rangle$ is a sequence of constituent vertices such that (v_i, v_{i+1}) is an edge in the graph, for all $i \in [0, \ell - 1]$. We denote the length of path P as $|P|$ and $|P| = \ell$ is the number of edges on P. We also call $|P|$ the number of hops of P. The first and the last vertex of the path are $head(P) = v_0$ and $tail(P) = v_\ell$. For a vertex v, we say $v \in P$ if $v = v_i$ for some $i \in [0, \ell]$. We consider the weight of path P to be the sum of the weights of the edges that make up the path, $w(P) = \sum_{i=1}^{\ell} w(v_{i-1}, v_i)$. A path P' is a $(1 + \epsilon)$-approximation path for another path P, if $head(P) = head(P')$, $tail(P) = tail(P')$, and $w(P) \le w(P') \le (1 + \epsilon)w(P)$.

Hopsets. A (β, ϵ)–**hopset** for directed graph $G = (V, E)$ is a set of weighted edges H, such that for any vertices u and v in V, $dist_G(u, v) \le dist_{G'}^{(\beta)}(u, v) \le (1 + \epsilon)dist_G(u, v)$, where $G' = (V, E \cup H)$. β is considered the **hopbound** of the hopset, and $|H|$ is the size of the hopset.

Related nodes. For nodes u, v define the relation $u \preceq_d v$ if and only if $dist_G(u, v) \le d$. We say u can reach v within d-distance or v can be reached by u within d-distance if $u \preceq_d v$. If $u \preceq_d v$ or $v \preceq_d u$, then u and v are d-**related**. For a directed graph $G = (V, E)$ and vertices $u, v \in V$, denote $R_d^+(G, v) = \{u | v \preceq_d u\}$ and $R_d^-(G, v) = \{u | u \preceq_d v\}$ to be the set of nodes which can be reached by v, and which can reach v within d-distance. We denote the set $R_d(G, v) = R_d^+(G, v) \cup R_d^-(G, v)$ be v's related nodes within d-distance. If $d = n$, we will ignore d. Similarly, we can define $R_d^+(G, P) = \{u | v_i \preceq_d u, v_i \in P\}$, $R_d^-(G, v) = \{u | u \preceq_d v_i \in P\}$ and $R_d(G, P) = R_d^+(G, P) \cup R_d^-(G, P)$. If $v \in R_d(G, P)$, then v and P are d-**related**.

Path related nodes. For a vertex x and a path P, vertex x is a **d-descendant** of P if and only if $x \in R_d^+(G, P) \backslash R_d^-(G, P)$. Vertex x a **d-ancestor** of P if and only if $x \in R_d^-(G, P) \backslash R_d^+(G, P)$. x a **d-bridge** of P if and only if $x \in R_d^-(G, P) \cap R_d^+(G, P)$. Notice that these sets are all disjoint by definition.

Binomial distribution. In the paper, denote binomial variables with n independent experiments and probability p as $B(n, p)$. For a random variable X, if $X \sim B(n, p)$, the following holds by a Chernoff bound,

$$\mathbf{Pr}[\, X \ge (1 + \delta)np \,] \le exp(-\frac{\delta^2}{2 + \delta} np).$$

If $X \sim B(n, p)$, then

$$E[\frac{1}{X + 1}] \le \frac{1}{E[X]}.$$

3 ALGORITHM

In this section, we describe the hopset algorithm Hopset(G). The algorithm takes as input graph $G = (V, E)$, and has parameters k, λ and L. The goal of the algorithm is to output a set of edges E' that is a $((n^{1/2+o(1)}), \epsilon)$-hopset of G.

At a high level, the algorithm chooses vertices, called **pivots**, to search forwards and backwards adding labels to each reached vertex. The labels are used to partition the graph into subgraphs for recursion. There is another set of vertices, called **shortcutters** that search forwards and backwards adding edges to the hopset for each reached vertex. The edges that are added to the hopset are weighted by the distance between the shortcutter and the reached vertex. The search is limited in distance, so vertices on the boundary of the search, called **fringe vertices**, are replicated and put into multiple subproblems. With each level of recursion, the number of pivots increases, while the search distance decreases. The union of the edges added in each level of recursion is returned as the hopset. Next, we will describe some components of the algorithm and then describe the details of the algorithm.

Parameters. The algorithms Hopset and HSRecurse have parameters k, λ and L. The parameter k controls the probability that a vertex is chosen as a pivot in each level of recursion. The parameter L controls the number of shortcutters in each level of recursion. A higher value for L gives a better approximation but also increases the runtime. Finally, the parameter λ, which is a constant and controls the probability the algorithm succeeds. The algorithm requires that $k \geq 2$, and $\lambda \geq 8$.

Pivots and shortcutters. Each vertex v is assigned a level, $\ell(v)$ that is used to determine at what level of recursion it becomes a pivot or a shortcutter. A vertex v is a **pivot** at recursive level r if $\ell(v) = r$. A vertex u is a **shortcutter** at recursive level r if $\ell(u) \leq r + L$. Since each vertex v is assigned $\ell(v)$ at the onset of the algorithm and not changed, we can note that if v becomes a pivot at level r, then it was a shortcutter at level $max(0, r - L)$. Pivots search the graph and add labels to reached vertices that used to partition the graph in subgraphs for recursion. Shortcutters searc the graph and add hopset edges but do not add labels, and therefore do not affect the partitioning of the graph at that level.

Search distances. Each level of recursion has a range for search distances. The ranges are disjoint and decreasing with each level of recursion. For a level of recursion r and vertex v, the search distance is $\rho_v D_r$ where $D_r = D/(\lambda^r k^{r/2})$ is the basic search distance and ρ_v is the scalar. The range of search distances is $(\rho_{min} D_r, \rho_{max} D_r)$, where $\rho_{min} = 16\lambda^2 k^2 \log^2 n - 1$ and $\rho_{max} = 32\lambda^2 k^2 \log^2 n$. The search distance range is divided into $4\lambda^2 k \log^2 n$ disjoint subintervals, each with length $4k$. A subinterval is chosen uniformly at random, which is represented by σ_v in the algorithm. Finally, the scalar ρ_v is chosen from within the subinterval to minimize the number of fringe vertices when using search distance $\rho_v D_r$. We use these search distances to guarantee that there are not too many fringe vertices.

Explanation of Algorithm 1 and Algorithm 2. Hopset(G), shown in Algorithm 1, repeats $\log n$ times to make the probability of success high. It assigns $\ell(v) = i$ for each vertex v with probability $(\lambda k^{i+1} \log n)/n$. The $\ell(v)$ is the level of recursion that v becomes a pivot. The probability increases by k with each level of recursion. The recursive subroutine HSRecurse(G, D, r) is called for D set to $2^j/k^c$ for $j \in [\log n/2, \log n]$. This ensures that a path of any length in $n^{1/2}$ to n is shortcutted. For each vertex v, after assigning $\ell(v)$, if $\ell(v) \leq L$, search forwards and backwards for 2^{j+1} and add an edge to the hopset for each reached vertex with weight equal to the distance from the shortcutter to the reached vertex. Call the recursive subroutine HSRecurse($G, D = 2^j k^{-c}, r = 0$) on the whole graph G with D set to $2^j k^{-c}$ for $j \in [\log n/2, \log n]$. Return the set of edges added to the hopset in all recursive executions.

HSRecurse(G, D, r) is the recursive subroutine shown in Algorithm 2. It takes graph G, distance D, and level of recursion r as input. For each pivot at level r, i.e. each vertex v where $\ell(v) = r$, choose a σ_v uniformly at random from $[1, 4\lambda k \log^2 n]$. Next, search from v to distance $16\lambda^2 k^2 \log n + 4k\sigma_v$ and find the distance ρ_v that has the minimal number of vertices exactly ρ_v distance away, where ρ_v is restricted to $[16\lambda^2 k^2 \log^2 n + 4k(\sigma_v - 1), 16\lambda^2 k^2 \log^2 n + 4k\sigma_v)$. Search forwards and backwards from v to distance $\rho_v D_r$ and add labels v^{Des} and v^{Anc} to the vertices reached in the forwards and backwards directions, respectively. Add the label X on any vertex that is reached in both directions. Next define the fringe vertices V_v^{fringe} for vertex v as $R_{(\rho_v+1)D_r}(G, v) \backslash R_{(\rho_v-1)D_r}(G, v)$, and recurse on the induced subgraph $G[V_v^{fringe}]$.

Next for each shortcutter, i.e. each vertex v where $\ell(v) \leq r + L$, search forwards and backwards from v for distance $32\lambda^2 k^2 D_r \log^2 n$ and for each reached vertex u, add edge (u, v) for ancestors (or (v, u) for descendants) with weight $dist(u, v)$ (or $dist(v, u)$ to the hopset. Next, remove any vertices that received a label X from the pivots. Finally, partition the vertices into groups as described in the next section, and recurse on the subgraph induced on each group of vertices.

Partition based on labels. Line 15 from Algorithm 2 is as follows. Partition the graph such that two vertices u and v are in the same group V_i, if and only if u and v receive the same labels from all pivots. There could be a group of vertices that receives no labels from any pivots. Notice that any vertices that received a X label from a pivot are removed in the step before. Therefore, none of the subgraphs contain vertices that received a X label. Finally, the pivots themselves are removed from the graph, as each pivot receives the X label from itself.

4 ANALYSIS

The goal of this section is to prove the following theorem.

Theorem 4.1. *There exists a randomized sequential algorithm that takes a directed graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, computes a $(n^{1/2+o(1)}, \epsilon)$-hopset of size $\tilde{O}(n/\epsilon^2)$ with high probability, and runs in $\tilde{O}(m/\epsilon^2)$ time.*

We start by proving the runtime and the size of the hopset in Section 4.1. Then we show the hopbound in Section 4.2, and finally, the approximation in Section 4.3.

Algorithm 1 Hopset algorithm for unweighted directed graphs. k, λ and L are parameters.

function HOPSET($G = (V, E)$)
  $H \leftarrow \emptyset$
  repeat $\lambda \log n$ times
    for each $j \in [\log n/2, \log n]$
      for each $v \in V$
        for each $i \in [0, \log_k n]$
          With probability $(\lambda k^{i+1} \log n)/n$, set $\ell(v)$ to i, **break** if setting successful.
          if $\ell(v) \leq L$ **then**
            for each $u \in R^{+}_{2^{j+1}}(G, v)$ add edge (v, u) to H with weight $dist_G(v, u)$
            for each $u \in R^{-}_{2^{j+1}}(G, v)$ add edge (u, v) to H with weight $dist_G(u, v)$
      $H \leftarrow H \cup$ HSRECURSE($G, D = 2^j k^{-c}, r = 0$)
  return H

Algorithm 2 Recursive subroutine for HOPSET Algorithm. k, λ and L are parameters.

function HSRECURSE(G, D, r)
  $D_r \leftarrow D/(\lambda^r k^{r/2}), H \leftarrow \emptyset$
  for each $v \in V$ with $\ell(v) = r$
    Choose σ_v uniformly at random from $[1, 4\lambda^2 k \log^2 n]$
    Minimize $|R_{(\rho_v+1)D_r}(G, v) \backslash R_{(\rho_v-1)D_r}(G, v)|$ such that $\rho_v \in [16\lambda^2 k^2 \log^2 n + 4k(\sigma_v - 1), 16\lambda^2 k^2 \log^2 n + 4k\sigma_v)$
    for each $u \in R^{+}_{\rho_v D_r}(G, v)$ add label v^{Des} to vertex u
    for each $u \in R^{-}_{\rho_v D_r}(G, v)$ add label v^{Anc} to vertex u
    for each $u \in R^{+}_{\rho_v D_r}(G, v) \cap R^{-}_{\rho_v D_r}(G, v)$ add label X to vertex u
    $V_v^{\mathbf{fringe}} \leftarrow R_{(\rho_v+1)D_r}(G, v) \backslash R_{(\rho_v-1)D_r}(G, v)$
    $H \leftarrow H \cup$ HSRECURSE($G[V_v^{\mathbf{fringe}}], D, r + 1$)
  for each $v \in V$ with $\ell(v) = r + L$
    for each $u \in R^{+}_{32\lambda^2 k^2 D_r \log^2 n}(G, v)$ add edge (v, u) to H with weight $dist_G(v, u)$
    for each $u \in R^{-}_{32\lambda^2 k^2 D_r \log^2 n}(G, v)$ add edge (u, v) to H with weight $dist_G(u, v)$
  for each $u \in V$ that has a X label, remove u
  $V_1, V_2, ..., V_t \leftarrow$ partition based on labels
  for each $i \in [1, t]$
    $H \leftarrow H \cup$ HSRECURSE($G[V_i], D, r + 1$)
  return H

4.1 Running Time and Hopset Size

In this section we bound the runtime of the algorithm and the size of the hopset the algorithm returns.

THEOREM 4.2. *One execution of* HOPSET($G = (V, E)$) *with parameter k, where $n = |V|$, $m = |E|$, runs in $\tilde{O}(mk^{L+1})$ time and produces a hopset of size $\tilde{O}(nk^{L+1})$.*

The proof of Theorem 4.2 follows the same structure as the runtime proof from JLS [14]. First, we bound the related vertices in each recursive subproblem in Lemma 4.3. Then we show the number of times a vertex is added to the fringe problem is small in Lemma 4.4. Since only fringe vertices are duplicated, we can bound the total number of vertices and edges in all recursive subproblems in Lemma 4.5. This allows us to prove the number of edges added to the hopset and the cost of all recursive executions. The runtime differs from JLS [14] because of the extra searches from shortcutters. For the same reason, the size of the hopset is larger than the number of shortcutters added in JLS [14].

We start by bounding the number of related vertices in recursive subproblems. In each level of recursion, the probability of being a pivot increases. With more pivots, the graph is partitioned into more subproblems, and the number of related vertices in each subproblem decreases. The proof of vertices in core problems is the same as JLS [14]. Our algorithm differs from JLS [14] for the fringe problem because we increase r as we recurse on fringe problems. Since the search distance is chosen to minimize the number of vertices on the fringe, the number of vertices in the fringe problem is small, and therefore each vertex does not have too many related nodes. The upper bound for the vertices in the fringe problem is needed for the hopbound in Section 4.2. The proof of the following lemma is in the full version of the paper [4].

Lemma 4.3. *Consider an execution of HSRecurse$(G', d, 0)$ on n-node m-edge graph G. With probability at least $1 - n^{-0.7\lambda+3}$ in each recursive call of HSRecurse(G', D, r) the following holds for all $v \in G'$,*

$$|R^{+}_{\rho_{max} D_r}(G', v)| \leq nk^{-r}, |R^{-}_{\rho_{max} D_r}(G', v)| \leq nk^{-r}.$$

Next, we consider the expected number of nodes added to fringe problems. JLS [14] has a similar lemma, where they consider the expected number of times a vertex is added to a fringe problem. Since we choose a search distance to minimize fringe vertices, we cannot get the same expectation. Instead, we count the number of vertices each pivot adds to its fringe problem, and get the same result.

The basic search distance D_r for a pivot v is scaled by a factor ρ_v that is chosen from an interval to minimize the number of vertices in the fringe problem. The interval that ρ_v is chosen from is selected uniformly at random from a larger interval. By using a scaling factor that minimizes the number of vertices on the fringe, and chosen from a random interval, we can guarantee that the number of vertices added to each fringe problem is small. See the full version of the paper for the proof of Lemma 4.4 [4].

Lemma 4.4. *Consider a call to HSRecurse(G', D, r) and any vertex $v \in G'$. The expected number of nodes added to v's fringe problem i.e. $|R_{(\rho+1)D_r}(G', v) \backslash R_{(\rho-1)D_r}(G', v)|$ is $1/(4\lambda k \log n)$.*

Now that the number of vertices in fringe problems is bounded, we can bound the total number of vertices in all recursive subproblems. Lemma 4.5 is based on Lemma 5.3 and Corollary 5.5 from JLS [14]. The vertices in the core problem form a partition of the vertices in the level before. The vertices in the fringe problem are copies of vertices in the core problem, which means the total number of vertices increases with each level. However, since we just showed the number of vertices in the fringe problem is small, the total number of vertices in all recursive subproblems can still be bounded.

Lemma 4.5. *Consider one execution of Hopset$(G = (V, E))$ where $n = |V|$ and $m = |E|$. The expected number of vertices in all recursive executions of HSRecurse(G', D, r) is $2n \log n$. The expected number of edges in all recursive executions of HSRecurse(G', D, r) is $2m \log n$.*

Proof. In one execution of HSRecurse$(G' = (V', E'), D, r)$, the number of vertices called in recursive subproblems is the number called in the fringe problem, HSRecurse$(G[V_u^{fringe}], D, r + 1)$, and the number of vertices called in HSRecurse$(G[V_i], D, r + 1)$ for $i \in [1, t]$. By Lemma 4.4, the expected number of nodes added to one vertex's fringe problem is $1/(4\lambda k \log n)$. The vertices in V_i are a partition of the vertices in G'. Therefore the total expected number of vertices in the following subproblem is $|V'|(1 + 1/(4\lambda k \log n))$. The total number of levels of recursion is at most $\log_k n$. Therefore over all levels of recursion, the expected number of vertices in all subproblems is

$$\sum_{r=0}^{1+\log_k n} n(1 + \frac{1}{4\lambda k \log n})^r \leq 2n \log n$$

for $k \geq \log n$. The edge case can be proved in the same way. □

Next, we bound the number of related pivots each vertex has. This will set up for the proof of the runtime and size of the hopset.

Lemma 4.6. *Consider a call to Hopset(G) and all recursive calls of HSRecurse(G', D, r). For all $v \in V$, the number of pivots u, such that $v \in R(G', (\rho_u + 1)D_r, u)$ is $6\lambda k \log n$ with probability at least $1 - n^{-0.7\lambda+4}$.*

Proof. To bound the number of pivots u, where $v \in R(G', (\rho_u + 1)D_r, u)$, we will slightly overcount the pivots, by extending $\rho_u + 1$ to ρ_{max}. This will only increase the pivots we are counting. Observe that all pivots u such that $v \in R(G', \rho_{max} D_r, u)$ are in $R(G', \rho_{max} D_r, v)$. By Lemma 4.3, $|R(G', \rho_{max} D_r, v)| \leq 2nk^{-r}$ with probability $1 - n^{-0.7\lambda+3}$. The number of pivots is a binomial distribution of $B(|R(G', \rho_{max} D_r, v)|, \frac{\lambda k^{r+1} \log n}{n})$ and therefore,

$$\mathbf{Pr}[B(\left|R(G', \rho_{max} D_r, v)\right|, \frac{\lambda k^{r+1} \log n}{n}) > 6\lambda k \log n] \leq e^{-2\lambda k \log n} \leq n^{-2\lambda}.$$

By taking a union bound over all $v \in V$ and all r, the claim holds with probability at least $1 - n^{-0.7\lambda+4}$. □

Lemma 4.7. *Consider a call to Hopset$(G = (V, E)$. For all nodes $v \in V$, the number of shortcutters u, such that $v \in R(G, 2^{j+1}, u)$ is $6\lambda k^{L+1} \log n$ with probability at least $1 - n^{-0.7\lambda+4}$. Consider all recursive calls of HSRecurse(G', D, r). For all $v \in V$, the number of shortcutters u, such that $v \in R(G', \rho_{max} D_r, u)$ is $6\lambda k^{L+1} \log n$ with probability at least $1 - n^{-0.7\lambda+4}$.*

Proof. We will prove each of the two cases separately, starting with the second case of the shortcutters in HSRecurse(G', D, r). This case is almost the same as Lemma 4.6 except for the probability of being a shortcutter at level r is $\lambda k^{L+r+1} \log n/n$. Therefore, the expected number of shortcutters u such that $u \in R(G', \rho_{max} D_r, v)$ is $2\lambda k^{L+1} \log n$, and with probability $1 - n^{-0.7\lambda+4}$, the number of shortcutters u is at most $6\lambda k^{L+1} \log n$.

For the first case, only vertices v where $\ell(v) \leq L$ are shortcutters, and there are at most n vertices. Hence, one vertex is a shortcutter with probability at most $\sum_{i=0}^{L} \lambda k^{i+1} \log n/n \leq 2\lambda k^{L+1} \log n/n$, for $k \geq 2$. The number of shortcutters in Hopset(G) is a binomial distribution $B(n, \frac{2\lambda k^{L+1}}{n})$ and by a Chernoff bound,

$$\mathbf{Pr}[B(n, \frac{2\lambda k^{L+1} \log n}{n}) > 6\lambda k^{L+1} \log n] \leq e^{-2\lambda k \log n} \leq n^{-2\lambda}.$$

□

Now we can prove Theorem 4.2, the runtime of the algorithm, and the size of the hopset. The runtime is different from the JLS algorithm because of the additional shortcutters that perform searches.

Proof of Theorem 4.2. Assigning probabilities to vertices can be done in linear time. The searches from pivots and shortcutters can be implemented using breadth-first search. The cost of the searches by pivots is the number of edges explored in the breadth-first searches times the number of edges in all recursive subproblems. This is $O(mk \log^2 n)$ by Lemma 4.5 and Lemma 4.6. Similarly, the cost of searches for shortcutters is the edges explored in the breadth-first searches, which is $6\lambda k^{L+1} \log n$ by Lemma 4.7, times the number of edges in all recursive subproblems, which is $2m \log n$

by Lemma 4.5. Finally, the partition step can be implemented to run in $O(n \log nk)$ by sorting different labels. In total the runtime is $O(mk^{L+1} \log^4 n)$. The number of hopset edges added is, at most, the number of vertices explored in the searches. The total number of vertices searched is the expected number of vertices in all recursive subproblems times the number of times each vertex is searched over all levels of recursion. By Lemma 4.7 and Lemma 4.5, this is $O(nk^{L+1} \log^4 n)$ total hopset edges. □

4.2 Hopbound

Our goal in this section is to show the hopbound of the hopset produced by the Hopset(G) algorithm is $n^{\frac{1}{2}+O(1/\log k)}k^{c+\frac{1-L}{2}} \log^2 n$. The main idea comes from Fineman [11] and JLS [14]. We consider the shortest path P from u to v through the full execution of the algorithm. If a bridge is selected as a pivot, then the path is shortcutted to two hops. If no bridges are selected as pivots, then the pivots are ancestors, descendants, or unrelated the path. When an ancestor or a descendant is a pivot, it splits that path into subpaths that are contained in different recursive subproblems. Define a **path-relevant subproblem** (G, P, r) as a call to HSRecurse(G, D, r) that contains a nonempty subpath of P. Splitting the path makes it more challenging to shortcut because a bridge is needed for each subpath in its path-relevant subproblem. However, we are still making progress because the number of nodes in path-relevant subproblems is reduced. Hence, we would like to track the collection of path-relevant subproblems throughout the execution of the algorithm.

The path-relevant subproblems form a **path-relevant subproblem tree** defined as follows. The root of the tree, called level 0, is the whole path P. If a bridge is selected as a pivot in a path-relevant subproblem, then the node is a leaf and has no children. If no bridges are selected in a path-relevant subproblem (G', P', r), then the path-relevant subproblems containing subpaths of P' are the children. At the end of the execution of the algorithm, the leaves of path-relevant subproblems tree represent the entire path P. The path consists of at most two hops for each leaf node in the tree and the edges that go between subproblems. Our goal is to bound the number of nodes in the path-relevant tree to provide an upper bound of the hopbound. The idea of the path-relevant subproblems tree comes from Fineman [11]. However, ours becomes more complicated because we use multiple pivots, and fringe and core problems.

In Lemma 4.8, we will construct the path-relevant subproblem tree. The proof relies on a helper lemma to show that choosing ancestor and descendant pivots will decrease the number of path-related nodes. We will show this claim after Lemma 4.8 in Lemma 4.9. The construction of the path-relevant subproblems tree becomes more complicated for two reasons. First, the basic search distance D_r decreases with each level of recursion, which means that a pivot may not reach the end of the path in its search. This splits the path into an additional subpath. Second, the algorithm calls core and fringe problems from each pivot. It creates many subproblems, so we must choose which of these subproblems to consider in the analysis.

To resolve the first difficulty, we will logically split certain path-relevant subproblems to create **logical path-relevant subproblems**. The path is split logically for the sake of analysis. However, the algorithm is unaware of these splits. This means that some logical subproblems are in the same call of HSRecurse(G, D, r), but this will not change our analysis. Notice that between two consecutive levels, the basic search distance will decrease by a $O(\sqrt{k})$ factor. The pieces of the subpath are split such that the length of each piece is less than the next level's search distance. This guarantees that the search distance in the next level is long enough to reach the end of the subpath in the logical path-relevant subproblem. The ancestors and descendants of each piece of the subpath are copied and added to each relevant subproblem. By splitting subproblems, we introduce an additional $O(\sqrt{k})$ subproblems, as well as multiple copies of many nodes. Fortunately, since we have already shown that path related nodes in one subproblem are bounded, this increase in vertices is tolerable.

More specifically, each call to HSRecurse(G, D, r) is associated with path $\hat{P}$ where $|\hat{P}| = \ell \in (k^c D/2, k^c D]$. If a path-relevant subproblem (G', P, r) at level r contains a subpath $P = \langle v_i, v_{i+1}, ..., v_j \rangle$ with $j - i > D_r = D/(\lambda^r k^{r/2})$, then we will split P into $q = \lceil j - i/D_r \rceil$ disjoint subpaths $P = P_1, P_2, ...P_q$, such that every subpath except the last one has length D_r. This partition splits the path into at most $\lambda^r k^{r/2}$ subpaths where each subpath has length at most D_r, which is less than the length of the basic search distance at level r. Each related vertex to a path vertex v_i in G' is copied to v_i's new logical path-relevant subproblem. From Lemma 4.3, each subpath P_i at level r contains at most $2nk^{-r}$ related vertices. We have at most $\lambda^r k^{c+r/2}$ new logical nodes since we have at most $\lambda^r k^{c+r/2}$ subpaths of length $D/(\lambda^r k^{c+r/2})$. Hence, we only duplicate $2\lambda^r nk^{c-r/2}$ additional vertices in this procedure. Next, we will construct the path-relevant subproblem tree based on the logical path-relevant subproblems in the following lemma, and show how to create the next level of subproblems from the logical subproblems layer. Let ρ_v be the scalar of the searching distance for pivot v. The proof of Lemma 4.8 is in the full version [4].

Lemma 4.8. *Consider a logical path-relevant subproblem $(G', P = \langle v_0, v_1, ..., v_\ell \rangle, r)$ corresponding to a call to HSRecurse(G', D, r). Let $p_r = (\lambda k^{r+1} \log n)/n$ be the probability a vertex is a pivot at level r. Let $S = \{v \mid \ell(v) = r, v \in R_{\rho_v D_r}(G', P)\}$ be the set of pivots at level r related to P within distance $\rho_v D_r$. There exists subpaths $P_0, P_1, P_2, ..., P_{2|S|}$ such that,*

1. *If a vertex $v \in S$ is a $\rho_v D_r$-bridge, there are no path-relevant subproblems.*
2. *If no vertex $v \in S$ is a $\rho_v D_r$-bridge, then the vertex union of all P_i for $0 \le i \le 2|S|$ is P.*
3. *$P_0, P_1, P_2, ..., P_{|S|+1}$ are in core problems and each P_i is contained in some V_{a_i}.*
4. *$P_{|S|+1}$, ..., $P_{2|S|}$ are called in fringe problems and each P_i is contained in some V_u^{Fringe}, where $u \in S$.*

Additionally, with probability $1 - n^{-0.7\lambda+4}$, we have that

$$\sum_{i=0}^{|S|} E[|R_{\rho_{min} D_r}(G'[V_{a_i}], P_i)|] \le \frac{3}{p_r}$$

and

$$\sum_{i=|S|+1}^{2|S|} E[|R_{\rho_{min} D_r}(G'[V_{u \in S}^{Fringe}], P_i)|] \le \frac{1}{p_r}.$$

Now we will show the helper lemma for the case the pivots are ancestors and descendants. The proof is in the full version of the paper [4]. Fineman [11] shows a similar result when there is just one pivot, and JLS [14] extends this to t pivots. In our case, we have the additional difficulty that each pivot searches for a different distance, but we are able to get the same result.

Lemma 4.9. *Consider the path $P = \langle v_0, v_1, .., v_\ell \rangle$, where $\ell \le D_r$, and its $\rho_{min} D_r$-distance ancestor set $R^-_{\rho_{min} D_r}(G, P)$ in the r^{th} level of recursion. Let I be the set containing all possible values of interval scalar. Choose t ancestor pivots uniformly at random from $R^-_{\rho_{min} D_r}(G, P)$. Let P_i be the path defined in Lemma 4.8. If the chosen interval $|I| \ge 4t$, then*

$$\sum_{i=0}^{|S|} E[|R^-_{\rho_{min} D_r}(G'[V_{a_i}], P_i)|] \le \frac{1.5}{t+1} \left| R^-_{\rho_{min} D_r}(G', P) \right|.$$

Notice that each subpath P_i will be contained in a subproblem, which means all P_i are valid in subproblems even if they were split in the logical layer. There might be some path-relevant subproblems replicated multiple times, so the path-relevant subproblems are no longer independent. Each subpath is limited in length $|P_i| \le D/(\lambda^r k^{r/2})$. We construct new logical layer based on the rule we mentioned before. Next, based on the path-relevant subproblem tree, we will give a lemma about the expected number of related nodes and subproblems in each level of recursion.

Lemma 4.10. *Consider the path-relevant subproblem tree for one execution of Hopset(G). Let Z_r be the number of subproblems in the r^{th} level of recursion. For all $r \ge 0$,*

$$\bigcap_{r \le \log_k n - L} \mathbf{Pr}\left[Z_r \le 32\lambda^r k^{c+\frac{r+1}{2}} \log^2 n \right] \ge \frac{1}{2}.$$

Proof. To show the claim, we will first show the expectation of Z_r. Let Y_r be the number of path related vertices in the r^{th} level of recursion. Our target is to show the following formula holds with probability $1 - n^{-0.7\lambda+4}$, for all r,

$$E[Y_r] \le 4\lambda^r n k^{c-\frac{r}{2}}$$
$$E[Z_r] \le 15\lambda^r k^{c+\frac{r+1}{2}} \log n.$$

If the expectation of Z_r in the above formula holds, then by Markov's inequality,

$$\mathbf{Pr}\left[Z_r \ge 30\lambda^r k^{c+\frac{r+1}{2}} \log^2 n \right] \le \frac{1}{2\log n},$$

and by a union bound, the following holds if $\lambda \ge 8$,

$$\bigcup_{r \le \log_k n - L} \mathbf{Pr}\left[Z_r \ge 32\lambda^r k^{c+\frac{r+1}{2}} \log^2 n \right] \le \frac{\log_k n - L}{2\log n} + n^{-0.7\lambda+4} \le \frac{1}{2}.$$

Next we will show the expectation of Z_r and Y_r by induction on the level of recursion. When $r = 0$, the claim is trivial since there is one subproblem and at most n path-related vertices. Assume for level r, the formulas hold. Then we will construct the logical layer. Let Y'_r be the number of path-related nodes in the logical layer at level r. Let Z'_r be the number of subproblems in the logical layer at level r. The search distance for level r is $D/\lambda^r k^{r/2}$ and subproblem is duplicated if the path length in the subproblem is greater than $\ell/(\lambda^r k^{c+r/2})$. Thus, at most $\lambda^r k^{c+r/2}$ subproblems are duplicated and $Z'_r = Z_r + \lambda^r k^{c+r/2}$. On the other side, from Lemma 4.3, the number of related nodes in each subproblem at level r is less than or equal to $2nk^{-r}$ with probability $1 - n^{-0.7\lambda+3}$. Therefore,

$$Y'_r = Y_r + \lambda^r k^{c+r/2} \cdot 2nk^{-r} = Y_r + 2\lambda^r n k^{c-r/2}$$

Next we can count Z_{r+1} and Y_{r+1} based on the logical layer. By Lemma 4.8, for each subproblem at level r, the number of related nodes at level $r+1$ can be bounded. For a logical subproblem s at level r, let Y_s be the number of path-related nodes in s's subproblem at level $r+1$. The expectation of Y_{r+1} is,

$$\begin{aligned} E[Y_{r+1}] =& E[\sum_s Y_s] = \sum_s E[Y_s] = \sum_{Z'_r} \sum_s E[Y_s \mid Z'_r]\mathbf{Pr}[Z'_r] \\ \le& \frac{4}{p_r} \sum_{Z'_r} Z'_r \mathbf{Pr}[Z'_r] = \frac{4}{\lambda k^{r+1} \log n/n} \cdot (E[Z_r] + \lambda^r k^{c+r/2}) \\ \le& 64\lambda^{r-1} n k^{c-\frac{r+1}{2}} \le 4\lambda^{r+1} n k^{c-\frac{r+1}{2}} \end{aligned}$$

for $\lambda \ge 4$. For the Z_{r+1}, if there are t pivots, there will be at most $2t+1$ subproblems. To count Z_{r+1}, split $2t+1$ subproblems to two parts, $2t$ subproblems and 1 subproblem. The $2t$ part will contribute to the total number of pivots. On the other hand, each subproblem at level r will have 1 additional subproblem, which implies another Z'_r item. Therefore, if $k \ge 2$ then,

$$\begin{aligned} E[Z_{r+1}] &= \sum E[Z_{r+1} \mid Y'_r] \cdot \mathbf{Pr}[Y'_r] = p_r \cdot \sum 2Y'_r \mathbf{Pr}[Y'_r] + E[Z'_r] \\ &= \frac{2\lambda k^{r+1} \log n}{n} \cdot E[Y'_r] + E[Z'_r] \\ &\le \frac{2\lambda k^{r+1} \log n}{n} \cdot (4\lambda^r n k^{c-r/2} + 2\lambda^r n k^{c-r/2}) \\ &\quad + 15\lambda^r k^{c+\frac{r+1}{2}} \log n + \lambda^r k^{c+r/2} \\ &\le 15\lambda^{r+1} k^{c+1+r/2} \log n. \end{aligned}$$

□

Lastly, we will show the hopbound based on the path-relevant subproblem tree.

Lemma 4.11. *Consider any graph $G = (V, E)$ and any shortest path $\hat{P}$ with $|\hat{P}| \ge n^{1/2}$ and let $u = head(\hat{P})$ and $v = tail(\hat{P})$. Consider an execution of Algorithm 1. Let E' be the hopset produced, and let $Z_0, Z_1, ..., Z_r$ be the number of corresponding path-relevant tree subproblems at level r, then there is a u-to-v path in $G' = (V, E' \cup E)$ containing at most $3\sum_{r \le \log_k n - L} Z_r$ edges.*

Proof. A path-relevant subproblem tree node will have no children if the subproblem contains a path-relevant pivot that is a bridge. If any pivots w, are bridges at or before level L, then w will be a shortcutter in Algorithm 1. Notice that w is $\rho_{max} D_r$-related to $\hat{P}$ for $r \le L$. We require that $\rho_{max} D_0 \le \ell$ since we only search for additional ℓ distance. The new path will be u to w to v.

Otherwise, there are no bridges in the first L levels. Consider a path-relevant subproblem at level $r' > L$. If there is a pivot w at level r' that is a bridge, then at level $r' - L$ w was a shortcutter in a path-relevant subproblem $(G, P', r' - L)$. In Lemma 4.8 we showed

that $P' \leq D_r$. Since shortcutters search for $\rho_{max} D_r$, w reaches $head(P')$ and the edges $tail(P')$, $(head(P'), w)$ and $(w, tail(P'))$ are added to E', creating a two hop path from u to v in G'. At level $\log_k n$, all vertices are pivots, and therefore the path must have a bridge pivot. In total there are at most $2\sum_{r \leq \log_k n - L} Z_r$ hopset edges that shortcut path-relevant subproblems, and there are at most $\sum_{r \leq \log_k n - L} Z_r$ edges between subproblems. Adding these together completes the proof. □

Lemma 4.12. *Consider any graph $G' = (V, E)$ and an execution of Hopset(G') with parameters k, λ and L. The hopset produced has hopbound $n^{1/2+O(1/\log k)} k^{c+(1-L)/2} \log^2 n$ with probability $1 - n^{-\lambda+2}$.*

Proof. Consider any shortest path $\hat{P}$ with $|\hat{P}| > n^{1/2}$ and let $u = head(\hat{P})$ and $v = tail(\hat{P})$. By Lemma 4.11, there is a path from u to v with at most $3\sum_{r \leq \log_k n - L} Z_r$ edges where Z_r is the number of path-relevant subproblems in the path-relevant subproblem tree at level r. Since the algorithm is repeated $\lambda \log n$ times, there exists a path relevant tree such that $\bigcap_{r \leq \log_k n - L} Z_r \leq 32\lambda^r k^{c+\frac{r+1}{2}} \log^2 n$ holds with probability $1 - n^{-\lambda}$, by Lemma 4.10. Therefore the hopbound is,

$$\sum_{r \leq \log_k n - L} 3Z_r = \sum_{r \leq \log_k n - L} 96\lambda^r k^{c+\frac{r+1}{2}} \log^2 n$$
$$= n^{1/2+O(1/\log k)} k^{c+(1-L)/2} \log^2 n,$$

with probability $1 - n^{-\lambda+2}$, where the probability comes from taking a union bound over all possible shortest paths. □

4.3 Approximation

In this section, we will show the approximation that the algorithm acheives. We have already showed that the path-relevant tree has $n^{\frac{1}{2}+O(1/\log k)} k^{c+\frac{1-L}{2}} \log^2 n$ nodes, which means there exist a path P' that contains at most $n^{\frac{1}{2}+O(1/\log k)} k^{c+\frac{1-L}{2}} \log^2 n$ hops. Now we want to show that P' is an good approximation of the original path $\hat{P}$. Notice that in the path-relevant tree, a path-relevant problem has no subproblems if one of the pivots at that level is a bridge. Consider the following two cases:

(1) If there is a bridge u with $\ell(u) \leq L$, then we stop the path-relevant tree at level 0. In this case, the search distance is at most $D \in [\ell k^{-c}, 2\ell k^{-c})$, so the bridge will have at most $2 \cdot 32\lambda^2 k^2 \log^2 n \cdot D \leq 128\lambda^2 k^{2-c} \log^2 n \cdot \ell$ error. The 2 comes from the forward and backward searches, the second item $32\lambda^2 k^2 \log^2 n$ comes from the scaling factor.

(2) Consider the path-relevant tree after level 0. If a path-relevant subproblem selects a shortcutter that is a bridge at level $r + L$, then the path-relevant subproblem will end at level r. The error for this subproblem is at level r is at most $2 \cdot 32\lambda^2 k^2 \log^2 n \cdot D_{r+L}$. Summing up all possible bridges, we have the error

$$\sum_{r=1}^{r=\log_k n - L} Z_r \cdot 64\lambda^2 k^2 \log^2 n \cdot D_{r+L} \leq 4096\lambda^{2-L} k^{(5-L)/2} \log^5 n \cdot \ell.$$

The accumulating error will be $4096\lambda^{2-L} k^{(5-L)/2} \log^5 n \cdot \ell$.

To make the first error equal to second error, set $k^c = \frac{\lambda^L k^{(L-1)/2}}{32 \log^3 n}$. If $k = \Omega(\log n)$ and the desired error is $\epsilon\ell$, set $L = 15 - 2\log_k \epsilon$. The hopbound β is at most $6\lambda^{\log_k n} n^{1/2} / \log n$. The running time is $O(mk^{16} \log^4 n/\epsilon^2)$ and the hopset size is $O(nk^{16} \log^4 n/\epsilon^2)$. Combining all this together, the following corollary holds.

Corollary 4.13. *For any unweighted directed graph $G = (V, E)$, Hopset(G) with above parameter returns a $(\beta = n^{1/2+O(1/\log k)}, \epsilon)$-hopset of size $O(nk^{16} \log^4 n/\epsilon^2)$ in running time $O(mk^{16} \log^4 n/\epsilon^2)$ with probability $1 - n^{-\lambda+2}$.*

Proof of Theorem 4.1. From Theorem 4.2 and Corollary 4.13, Theorem 4.1 follows directly. □

5 WEIGHTED GRAPHS

This section presents an algorithm for hopsets for weighted directed graphs. The algorithm is almost the same as the unweighted case, so most of the analysis still holds. Our goal is to show that for graph G, the algorithm returns a $(n^{1/2+o(1)}, \epsilon)$-hopset of size $O(nk^{16} \log^3 n \log(nW)/\epsilon^2)$, and runs in $O(mk^{16} \log^4 n \log(nW)/\epsilon^2)$ time. Next we will present the algorithm, and in Section 5.2 we provide the analysis.

5.1 Weighted Hopsets Algorithm

Algorithm 3 shows the hopsets algorithms for weighted directed graphs. The algorithm is the same as the unweighted algorithm with one exception. Namely, WHopset(G) searches all possible path weights from -1 to nW where W is the maximum weight of an edge in the graph, whereas Hopset$(G = (V, E))$ only searches over path weights from $n^{1/2}$ to n. This difference is Line 4. The weighted algorithm extends the searches because the maximum shortest path distance in a weighted graph is nW. In the unweighted case, the maximum shortest path was at most n. WHopset(G) searches from -1 to account for edges with weight zero.

5.2 Analysis

The goal of this section is to prove Theorem 5.1.

Theorem 5.1. *For any weighted directed graph $G = (V, E)$, there exists a randomized algorithm that computes a $(\beta = n^{1/2+o(1)}, \epsilon)$-hopset of size $O(nk^{16} \log^3 n \log(nW)/\epsilon^2)$. The randomized algorithm runs in $O(mk^{16} \log^4 n \log(nW)/\epsilon^2)$ time with probability $1 - n^{-\lambda+2}$.*

Most of the analysis from the weighted case holds for the unweighted case. First, we will show the difference in the runtime in Lemma 5.2 and then the hopbound and approximation.

Lemma 5.2. *One execution of WHopset$(G = (V, E))$ with parameters k and L, where $n = |V|$, $m = |E|$, runs in $\tilde{O}(mk^{L+1} \log(nW))$ time and returns a hopset of size $\tilde{O}(nk^{L+1} \log(nW))$ with high probability.*

Proof. The running time proof follows from the proof of Theorem 4.2. The only comes from performing the searches. Breadth-first search can no longer be used because the graph is weighted. Instead Dijkstra's algorithm for shortest paths can be used which has cost $O(m + n \log n)$ [7]. This increases the runtime from the unweighted case by a $O(\log(nW))$ factor resulting in a runtime of $O(mk^{L+1} \log^4 n \log(nW))$. For the same reason the size of hopset is $O(nk^{L+1} \log^3 n \log(nW))$. □

Algorithm 3 Hopset algorithm for weighted directed graphs. k, λ and L are parameters.

function WHopset($G = (V, E)$)
  $H \leftarrow \emptyset$
  repeat $\lambda \log n$ times
    for each $j \in [-1, \log(nW)]$
      for each $v \in V$
        for each $i \in [0, \log_k n]$
          With probability $(\lambda k^{i+1} \log n)/n$, set $\ell(v)$ to i, **break** if setting successful.
          if $\ell(v) \leq L$ **then**
            for each $u \in R^+_{2^{j+1}}(G, v)$ add edge (v, u) to H with weight $dist_G(v, u)$
            for each $u \in R^-_{2^{j+1}}(G, v)$ add edge (u, v) to H with weight $dist_G(u, v)$
      $H \leftarrow H \cup$ HSRecurse($G, D = 2^j k^{-c}, r = 0$)
  return H

Next, we consider the hopbound of the weighted case. We again consider the path-relevant subproblems and construct the logical path-relevant subproblems. The only difference comes in how the logical path-relevant subproblems are constructed. Consider a path $\hat{P}$ from u to v, where $|w(\hat{P})| \in (k^c D/2, k^c D]$. If a path-relevant subproblem (G', P, r) at level r contains a subpath $P = \langle v_i, v_{i+1}, ..., v_j \rangle$, with $w(P) > D_r = \frac{D}{\lambda^r k^{r/2}}$ then split P into q disjoint subpaths $P = P_1, P_2, ..., P_q$ such that $(head(P_i), tail(P_{i+1})) \in P$ for $i \in [1, q)$ and maximize each subpath P_i such that $w(P_i) \leq D_r$ except for the last subpath. Here the path is split based on weight rather than the number of hops. Since $w(P_i) + w(head(P_i), tail(P_{i+1})) > D_r$, there are at most $\lambda^r k^{c+r/2}$ new logical nodes. Since the rest of the number of logical nodes introduced is the same, the rest of the analysis is unaffected.

Lastly, we show the approximation of the hopsets. For paths P where $w(P) > 0$, the analysis is the same. However for a path P where $w(P) = 0$, the analysis changes. Recall that the lightest non-zero edge weight is 1. The algorithm is run with $j = -1$ for this case. When $j = -1$, we are considering the path p with $w(p) < 1/2$. However, there is only ϵ error and the approximate path weight will be less than $(1 + \epsilon)w(p) < 1$. Therefore, the approximate path weight is 0 since the graph has no non-zero edge weight less than 1. By setting appropriate c, WHopset($G = (V, E)$) will return a $(n^{1/2+o(1)}, \epsilon)$- hopset for G. For the final error to be ϵ, set $L = 15 - 2\log_k \epsilon$. Combining the above analysis, gives us Theorem 5.1.

6 PARALLEL ALGORITHM

In this section, we show how to extend the weighted hopsets algorithm to a work-efficient, low span parallel algorithm. First, we will explain the difficulties of the hopsets algorithm in the parallel setting and give the high-level idea of overcoming these difficulties. Then we describe the details of our parallel algorithm for hopsets in Section 6.1. Finally, in Section 6.2, we provide an analysis of the work and span.

There are two main difficulties in making the weighted algorithm work in a parallel setting. First, Dijkstra's algorithm is used to perform the searches, but Dijkstra's algorithm is expensive in the parallel setting. To resolve this problem, we use the rounding technique from Klein and Subramanian [15]. Consider a path from v_0 to v_ℓ, $\hat{P} = \langle v_0, v_1, ..., v_\ell \rangle$. For each edge $e \in \hat{P}$, $w(e)$ is rounded up to the nearest integer multiple of $\delta w(\hat{P})/\ell$, where δ is a small number to be set later. Since $\hat{P}$ contains ℓ edges, each edge has at most $\delta w(\hat{P})/\ell$ error. The whole path has at most $\delta w(\hat{P})/\ell \cdot \ell = \delta w(\hat{P})$ error. The error is tolerable if δ is set to be small enough. Now consider the path with the rounded weights, but treating $\delta w(\hat{P})/\ell$ as one unit. Since all rounded edge weights are integer multiples of $\delta w(\hat{P})/\ell$, the new weight of path P is at most $\tilde{w}(\hat{P}) = \frac{w(\hat{P}) + \delta w(\hat{P})}{\delta w(\hat{P})/\ell} = (1 + \delta)\ell/\delta$. Therefore, the algorithm can use breadth first search with depth at most $O(\ell/\delta)$ to compute $R^+_{\rho_v D_r}(G, v)$ and $R^-_{\rho_v D_r}(G, v)$ in a call to HSRecurse($G, D = O(\ell/\delta), r$). The cost of the depth-first search depends only on ℓ instead of $w(\hat{P})$.

The second difficulty is that searching the entire path can be too expensive, even after the rounding step because a path may contain too many hops. The key idea is to run HSRecurse(G, D, r) with limited hops D. Then add the edges produced by the HSRecurse(G, D, r) to the graph. Consider HSRecurse(G, D, r) searches for at most 2β hops, where β is the hopbound HSRecurse(G, D, r) achieves, and a path $\hat{P}$ with $|\hat{P}| = 4\beta$. After the first execution of HSRecurse(G, D, r), there will be an approximate path P' for P such that $|P'| \leq 2\beta$ and $w(P') \leq (1 + \epsilon)\tilde{w}(P) \leq (1 + \delta)(1 + \epsilon)w(P)$. By repeating these steps, we can ensure that a path of any length gets approximated, and the hopbound is limited by the previous executions of HSRecurse(G, D, r). Moreover, for a path P of any length, run HSRecurse($G, D, r = 0$) $\log(|P|/(2\beta))$ times. This gives a $(1 + \delta)^{\log(|P|/2\beta)}(1 + \epsilon)^{\log(|P|/2\beta)}$ approximation, with hopbound 2β. One more execution gives the β hopbound.

6.1 Algorithm Description

In this section, we describe the parallel algorithm, PHopset(G), shown in Algorithm 4. The parallel algorithm extends the hopsets algorithm for weighted graphs in Section 5. There are two main differences. First, the parallel algorithm will round the weights of edges. Second, the parallel algorithm will execute the recurse subroutine HSRecurse(G', D, r) and then add the edges returned from the subroutine to the graph before executing the recursive subroutine again. We will describe these two steps in more detail.

One key modification to Algorithm 4 is as follows. In Lines 16-17, if the weight of an edge is less than 1, then set the weight to 0. Also, notice that the algorithm searches from $i = -2$. These steps are both done to account for zero weighted paths.

Algorithm 4 Parallel hopset algorithm for weighted directed graphs. δ, k, λ, c, L are parameters.

1: **function** PHopset($G = (V, E)$)
2: $H \leftarrow \emptyset$
3: $\beta \leftarrow 6\lambda^{\log_k n} n^{1/2}/\log n$
4: **repeat** $\lambda \log^2 n$ times
5: **for each** $i \in [-2, \log(n^2 W)]$
6: $\hat{w} = \delta \cdot 2^{i-1}/\beta, \hat{H}' \leftarrow \emptyset$
7: Construct a new graph $\hat{G} = (\hat{V} = V, \hat{E} = E)$
8: **for each** $e \in \hat{E}$
9: $$\tilde{w}(e) = \begin{cases} +\infty & \text{if } w(e) \geq 2^{i+1} \\ \left\lceil \frac{w(e)}{\hat{w}} \right\rceil & \text{if } w(e) < 2^{i+1} \\ 1 & \text{if } w(e) = 0 \end{cases}$$
10: **for each** $v \in \hat{V}$
11: **for each** $i' \in [0, \log_k n]$
12: With probability $(\lambda k^{i'+1} \log n)/n$, set $\ell(v)$ to i', **break** if setting successfully.
13: **if** $\ell(v) \leq L$ **then**
14: **for each** $u \in R^{+}_{8(1+\delta)\beta/\delta}(G, v)$ add edge (v, u) to $\hat{H}'$ with weight $dist_{\hat{G}}(v, u)$
15: **for each** $u \in R^{-}_{8(1+\delta)\beta/\delta}(G, v)$ add edge (u, v) to $\hat{H}'$ with weight $dist_{\hat{G}}(u, v)$
16: $H \leftarrow H \cup (\hat{w} \cdot \hat{H}') \cup (\hat{w} \cdot \text{HSRecurse}(\hat{G}, D = 4(1+\delta)\beta/(\delta k^c), r = 0))$
17: $E \leftarrow E \cup H$
18: **return** H

Rounding the edge weights. The algorithm starts by rounding up the weights of edges. This is Lines 6-9 in PHopset($G = (V, E)$). Recall that the lightest non-zero edge weight is 1, and the heaviest edge weight is W. β is the hopbound of the hopset produced by the sequential algorithm Hopset(G) in Section 4.2.

Consider a path $\hat{P} = \langle v_0, v_1, ..., v_\ell \rangle$ and suppose $\ell \in (\beta, 2\beta]$ and $w(\hat{P}) \in [2^i, 2^{i+1})$ for integer i. Let δ be a small number. Define $\hat{w} = 2^{i-1}\delta/\beta$. Round the weight of each edge e to the following integers,

$$\tilde{w}(e) = \begin{cases} \hat{w} & \text{if } w(e) = 0, \\ \left\lceil \frac{w(e)}{\hat{w}} \right\rceil \cdot \hat{w} & \text{if } w(e) < 2^{i+1}, \\ +\infty & \text{if } w(e) \geq 2^{i+1}. \end{cases}$$

By construction each edge has at most $\hat{w}$ error. Therefore, the rounded weight of the path, $\tilde{w}(\hat{P})$ has at most $\ell\hat{w} \leq \frac{2^{i-1}\delta}{\beta} \cdot 2\beta \leq \delta d$ error. By treating $\hat{w}$ as one unit, $\hat{P}$ is in the range of

$$\tilde{w}(\hat{P}) \in \left[\left\lceil \frac{w(\hat{P})}{\hat{w}} \right\rceil \cdot \hat{w}, \left\lceil \frac{(1+\delta)w(\hat{P})}{\hat{w}} \right\rceil \cdot \hat{w} \right)$$
$$\subset \left[\left\lceil \frac{2\beta}{\delta} \right\rceil, \left\lceil \frac{4(1+\delta)\beta}{\delta} \right\rceil \right) \cdot \hat{w} \subset [k^c D/(2+2\delta), k^c D] \cdot \hat{w},$$

if $k^c D = 4(1+\delta)\beta/\delta$. Since $\hat{w}$ is treated as one unit, breadth-first search can be run to depth at most $4(1+\delta)\beta/\delta$ to search the whole path, which is independent of d. In the algorithm, $\hat{w}$ is ignored in the rounding step and added back when HSRecurse(G, D, r) returns the hopset.

Adding hopset edges to the graph. After a recursive call to HSRecurse(G, D, r), Line 17 in Algorithm 4 adds the edges returned by HSRecurse(G, D, r) to the original graph G. HSRecurse(G, D, r) returns a (β, ϵ)-hopset for any path with length at most 2β with probability at least $1/2$. Therefore, for any path P with $|P| > 2\beta$, there will be a path P' approximating P, with length $|P'| = max(|P|/2, 2\beta)$.

6.2 Parallel Hopbound and Hopset Size

Lemma 6.1. *Consider any graph $G' = (V, E)$ and an execution of PHopset(G'). For any P where $|P| \leq 2\beta$, after the rounding code in Lines 6-9, suppose HSRecurse($G, D, r = 0$) returns a $(1+\epsilon')$ approximate path P' containing at most β hops with probability at least $1/2$. If Lines 5-16 in PHopset(G') are repeated $j\lambda \log n$ times, then for any u-to-v path $\hat{P}$ with $|\hat{P}| = 2^j\beta$, there will be an approximate path $\hat{P}'$ in E with probability $1 - (2^j - 1)n^{-\lambda}$ such that $|\hat{P}'| \leq \beta$ and $w(\hat{P}') \leq (1+\delta)^j(1+\epsilon')^j w(P)$.*

Proof. Proof by induction on j. When $j = 1$, then for $\hat{P}$ with $|\hat{P}| \leq 2\beta$, after $\lambda \log n$ repetitions of Lines 5-17, with all possible values of D, with probability $1 - \frac{1}{2^{\lambda \log n}} = 1 - n^{-\lambda}$, HSRecurse($G, D, R$) returns a $(1+\epsilon)$-approximate path for $\hat{P}$. When the edges of $\hat{P}$ are rounded, there is at most $\delta w(\hat{P})$ error for $\hat{P}$. Therefore, the final approximation ratio is $(1+\delta)(1+\epsilon')$.

For the inductive step, we will show that for $\hat{P}$ with $|\hat{P}| \leq 2^{j+1}\beta$, the claim holds. Split $\hat{P}$ into two subpaths, $\hat{P}_1$ and $\hat{P}_2$, where each of $\hat{P}_1$ and $\hat{P}_2$ contains no more than $2^j\beta$ edges. By the inductive hypothesis, with probability $1 - (2^{j+1} - 2)n^{-\lambda}$, there exists $\hat{P}'_1$ and $\hat{P}'_2$ such that $\hat{P}'_1$ and $\hat{P}'_2$ are $(1+\delta)^j(1+\epsilon')^j$-approximations for $\hat{P}_1$ and $\hat{P}_2$, respectively. Furthermore, $|\hat{P}'_1| \leq \beta$ and $|\hat{P}'_2| \leq \beta$. Hence, after $\log n$ repetitions, with probability $1 - n^{-\lambda}$, there will be a $(1+\delta)(1+\epsilon')$ approximate path $\hat{P}'$ with at most H edges for $\langle \hat{P}'_1, \hat{P}'_2 \rangle$, which

implies the approximate path for $\hat{P}$. By taking a union bound over the existence of $\hat{P}'_1$, $\hat{P}'_2$ and $\hat{P}'$, the probability is $1-(2^{j+1}-1)n^{-\lambda}$. □

For a path P with $|P| \le 2\beta$, HSRecurse(G, D, r) with corresponding $\hat{w}$ returns a (β, ϵ')-hopset for P. By setting $k^c = \frac{\lambda^L k^{(L-1)/2}}{32 \log^3 n}$ and $L = 15 - 2\log_k \epsilon'$, the hopbound is $\beta = 6\lambda^{\log_k n} n^{1/2}/\log n$. By repeating Lines 5-16 $\lambda \log^2 n$ times, Lemma 6.1 can be applied to all possible paths. The maximum path weight will increase each round, but it will be no greater than $(1+\epsilon)^{\log n} nW \le n^2 W$. Thus a maximum path weight of $n^2 W$ covers all possible paths. Finally, to get a (β, ϵ)-hopset, set $\delta = \epsilon/(8 \log n)$ and $\epsilon' = \epsilon/(8 \log n)$. If $k = \Omega(\log n)$, then $L = 17 - \log_k \epsilon$ is sufficient. The constant $1/8$ in ϵ' will cancel out with the λ^{-L} in the error formula. Recall that HSRecurse$(G, D, r = 0)$ will returns a hopset of size $O(nk^{L+1} \log^2 n)$. Summing up all items, the final hopset size is $O(nk^{18} \log^4 n \log(nW)/\epsilon^2)$.

Corollary 6.2. *For any weighted directed graph $G = (V, E)$, PHopset(G) with above parameter returns a $(\beta = n^{1/2+o(1/\log k)}, \epsilon)$-hopset of size $O(nk^{18} \log^4 n \log(nW)/\epsilon^2)$ with probability $1 - n^{-\lambda+3}$.*

6.3 Work and Span

Here we consider PHopset(G) in the work-span model [7]. Recall that the work is the total number of operations that the algorithm performs while the span is the longest chain of sequential dependent operations.

Work. The work of the algorithm is dominated by the cost of the searches. Updating the graph, and adding the edges back to the graph can be done using parallel merge sort [6]. See Fineman [11] and JLS [14] for details of the parallel implementation. From the proof of Theorem 4.2, the total amount of work to compute the set of related nodes in a call of HSRecurse(G, D, r) is $O(mk^{L+1} \log^4 n)$. In the parallel algorithm, the m term increases as more edges are added to the graph. When Lines 5-14 are repeated j times, there are at most $O(jnk^{18} \log^2 n \log(nW)/\epsilon^2)$ edges in H. The total work is,

$$O\Big(\sum_{j=1}^{\lambda \log^2 n} (m + jnk^{18} \log^2 n \log(nW)/\epsilon^2) k^{18} \log^2 n \log(nW)/\epsilon^2\Big)$$
$$= O(mk^{18} \log^4 n \log(nW)/\epsilon^2 + nk^{36} \log^8 n \log^2(nW)/\epsilon^4).$$

Span. The searches dominate the span. In each call to HSRecurse$(G, D, r = 0)$, the maximum search distance is $4(1+\delta)\beta/\delta$. On each recursive call, the search distance decreases by at least 1/2. Therefore the span in one call to HSRecurse$(G, D, r = 0)$ is $O(\beta/\delta)$. Since the algorithm is run $O(\log^2 n)$ times, the span is $O(\beta \log^2 n/\delta) = n^{1/2+o(1/\log k)} \log^2 n/\epsilon$.

Summing up all these together, allows us to prove the following theorem.

Theorem 6.3. *For any weighted directed graph $G = (V, E)$, there exists a randomized parallel algorithm for weighted graphs that computes a $(n^{1/2+o(1)}, \epsilon)$-hopset of size $O(n \log^{22} n \log(nW)/\epsilon^2)$. The algorithm has $O(m \log^{22} n \log(nW)/\epsilon^2 + n \log^{44} n \log^2(nW)/\epsilon^4)$ work and $n^{1/2+o(1)}/\epsilon$ span with high probability.*

Proof. Combining above analysis and Corollary 6.2, the theorem holds with $k = \Theta(\log n)$ and appropriate λ. □

Theorem 6.4. *There exists a parallel algorithm that takes as input a graph G with non-negative edge weights and computes approximate single-source shortest paths in $\tilde{O}(m \log(nW)/\epsilon^2 + n \log^2(nW)/\epsilon^4)$ work and $n^{1/2+o(1)}/\epsilon$ span.*

Proof. By Theorem 6.3, PHopset(G) produces a $(n^{1/2+o(1)}, \epsilon)$-hopset with the desired work and span. Then running Klein and Subramanian's hop-limited parallel algorithm for shortests paths [15] completes the proof. □

REFERENCES

[1] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd ACM Symposium on Theory of Computing*, 2020.

[2] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 443–454, New York, NY, USA, 2016. ACM.

[3] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D. Zaroliagis. A parallel priority queue with constant time operations. *J. Parallel Distrib. Comput.*, 49(1):4–21, February 1998.

[4] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Efficient construction of directed hopsets and parallel approximate shortest paths. *CoRR*, abs/1912.05506, 2019.

[5] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, January 2000.

[6] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, August 1988.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[8] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, November 1988.

[9] M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 128–137, Los Alamitos, CA, USA, oct 2016. IEEE Computer Society.

[10] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in rnc. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pages 333–341, New York, NY, USA, 2019. ACM.

[11] Jeremy T. Fineman. Nearly work-efficient parallel algorithm for digraph reachability. In *Proceedings of the 50th Annual ACM SIGACT Symposium on the Theory of Computation*, pages 457–470, 2018.

[12] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[13] Shang-En Huang and Seth Pettie. Lower bounds on sparse spanners, emulators, and diameter-reducing shortcuts. *ArXiv e-prints*, February 2018.

[14] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. Parallel reachability in almost linear work and square root depth. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1664–1686. IEEE Computer Society, 2019.

[15] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, November 1997.

[16] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd ACM Symposium on Theory of Computing*, 2020.

[17] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003.

[18] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 192–201, 2015.

[19] Thomas H. Spencer. Time-work tradeoffs for parallel algorithms. *J. ACM*, 44(5):742–778, September 1997.

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