Parallel Exact Shortest Paths in Almost Linear Work and Square Root Depth

Nairen Cao* Jeremy T. Fineman†

Abstract

This paper presents a randomized parallel single-source shortest paths (SSSP) algorithm for directed graphs with non-negative integer edge weights that solves the problem exactly in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span, with high probability. All previous exact SSSP algorithms with nearly linear work have linear span, even for undirected unweighted graphs. Our main technical contribution is to show a reduction from the exact SSSP to directed hopsets [6] using the iterative gradual rounding technique [9]. An (h, ϵ)-hopset is a set of weighted edges (sometimes called shortcuts) that when added to the graph admit h-hop paths with weights no more than $(1+\epsilon)$ times the true shortest path distances.

Furthermore, we show how to combine this algorithm with Forster and Nanongkai's framework [15] to improve the distributed exact SSSP algorithm. Specifically, we obtain an $\tilde{O}(\sqrt{n} + D + n^{2/5+o(1)}D^{2/5})$-round algorithm in the CONGEST model for exact SSSP in directed graphs with non-negative integer edge weights, where D is the unweighted diameter of the underlying undirected graph.

1 Introduction

The shortest-path problem is one of the most fundamental problems in combinatorial optimization. Given a weighted directed graph $G = (V, E, w)$ with non-negative integer edge weights, the single-source shortest-path problem is to find minimum-weight paths from a designated $s \in S$ to all other nodes. In the sequential setting, the classic solution has running time $O(m + n \log n)$ [16], where $n = |V|$ and $m = |E|$ are the number of vertices and edges in the graph, respectively. In the parallel setting, the existing algorithms are far from attaining the efficiency we would like. 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 reasonably large p), where the $\tilde{O}$ notation suppresses logarithmic factors. In order to achieve such a bound, a parallel algorithm must have nearly linear work and strongly sublinear span; the ***work*** of a parallel algorithm is the total number of primitive operations, and its ***span*** or ***depth*** is the length of the longest chain of sequential dependencies or, equivalently, the limit of parallel time as p approaches infinity.

Although dozens of papers [3, 4, 12, 19, 22, 27, 28] have been published on the exact version of the shortest-path problem, there is no ideal parallel solution, especially when the graph is sparse. Even for the simplest case of an unweighted, undirected graph, all algorithms to date either have linear span, meaning that they are inherently sequential, or they only manage to reduce the span at the cost of significantly increasing the work. For example, when tuned to achieve the span of $\tilde{O}(\sqrt{n})$, Spencer's algorithm [27] has work $\tilde{O}(m + n^2)$. Klein and Subramanian's algorithm [20] has work $\tilde{O}(m\sqrt{n})$. Table 1 summarizes the prior art on parallel SSSP algorithms for directed graphs along with our new result.

In light of the difficulty of producing efficient parallel algorithms that produce exact solutions to the single-source shortest-paths problem, much of the progress on parallel algorithms for shortest paths has come in the form of approximate solutions. The approximate SSSP problem is a relaxed version of the problem, where instead of outputting the exact distances $dist(s, v)$ from s to every vertex v, the algorithm instead may output an approximate distance d_v satisfying $dist(s, v) \le d_v \le (1+\epsilon)dist(s, v)$. For undirected graphs, the approximate SSSP problem has been widely studied [1, 10, 13, 14, 21, 23]. Recent breakthroughs indicate that it is possible to simultaneously achieve $\tilde{O}(m)$ work and $O(\text{poly}(\log n))$ span [1, 21, 26] for approximate SSSP on undirected graphs. Nevertheless, the exact version of the problem remains difficult even for the undirected case.

*Boston College
†Georgetown University

Paper	***work***	***span***	***edge weight***	***approximation***
Parallel Dijkstra's[16]	$\tilde{O}(m)$	$O(n)$	non-negative real	exact
Klein and Subramanian[20]	$\tilde{O}(m\sqrt{n})$	$O(\sqrt{n})$	non-negative integer	exact
Spencer's algorithm [27]	$\tilde{O}(m+n\rho^2)$	$O(\frac{n}{\rho})$	non-negative real	exact
Cao, Fineman and Russell [6, 7]	$\tilde{O}(m\rho^2+n\rho^4)$	$\frac{n^{1/2+o(1)}}{\rho}$	non-negative real	approximate
Our result	$\tilde{O}(m)$	$n^{1/2+o(1)}$	non-negative integer	exact

Table 1: SSSP algorithms for directed graphs in the work/span parallel model. To state the bounds more concisely, it is assumed here that the edge weights are bounded by a polynomial in n. The variable ρ is used to denote a parameter. Spencer's algorithm requires $\rho \in [1, n]$, and for Cao, Fineman, and Russell's algorithm, $\rho \in [1, n^{1/2+o(1)}]$. Omitted from the table, Forster and Nanongkai [15] extended Klein and Subramanian's algorithm; their extension also achieves a work/span tradeoff.

For directed graphs, Cao, Fineman, and Russell [6] give the first algorithm with both nearly linear work and sublinear span for approximate SSSP problem on directed graphs, which can also be generalized to achieve a work-span tradeoff [7].

For the exact SSSP in the CONGEST model [24], Ghaffari and Li [18] presented two randomized algorithms for directed graphs with polynomially bounded integer edge weights that run in $\tilde{O}(D^{1/4}n^{3/4})$ rounds and $\tilde{O}(n^{3/4+o(1)} + min\{n^{3/4}D^{1/6}, n^{6/7}\} + D)$ rounds. At the same time, Forster and Nanongkai [15] provided two randomized algorithms for graphs with polynomially bounded integer edge weights that run in $\tilde{O}(\sqrt{nD})$ rounds and $\tilde{O}(n^{1/2}D^{1/4} + n^{3/5} + D)$ rounds. Chechik and Mukhtar [9] recently showed a randomized algorithm that achieves $\tilde{O}(\sqrt{n}D^{1/4} + D)$ rounds.

Hopsets. One of the main tools for approximate shortest paths is a combinatorial structure called a hopset; this paper leverages hopsets to produce exact distances. An (h, 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 h hops, where h is called the ***hopbound***. Formally, given a graph $G = (V, E, w)$, H is an (h, ϵ)-hopset if and only if (1) for all edges $(u, v) \in H$, the weight $w(u, v)$ of the edge is not less 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 h hops such that $w(p) \leq (1 + \epsilon)dist_G(u, v)$, where $G' = (V, E \cup H, w)$ is the graph with the hopset edges included. Although hopsets were first formalized by Cohen [10], many earlier parallel algorithms for shortest paths used hopsets or analogous combinatorial structures implicitly.

We say that a graph is an (h, ϵ)-***hopset graph*** if for any pair of nodes u, v, $dist(u, v) \leq dist^{(h)}(u, v) \leq (1 + \epsilon)dist(u, v)$, where $dist(u, v)$ is the shortest path distance and $dist^{(h)}(u, v)$ is the shortest path distance containing at most h edges. A hopset graph may be obtained by taking the union of the input graph and a hopset for that graph.

There is a straightforward sequential algorithm that constructs an $(\sqrt{n}, 0)$-hopset with hopbound $\beta = \tilde{O}(\sqrt{n})$ and size $O(n)$. The algorithm randomly samples $\tilde{\Theta}(\sqrt{n})$ vertices, and the hopset contains all edges between each pair of sampled vertices, where the edge weights are the shortest-path distances. This algorithm actually produces an exact hopset (i.e., $\epsilon = 0$), but the algorithm has high running time as it entails running $\tilde{\Theta}(\sqrt{n})$ SSSP computations. Ullman and Yannakakis [28] parallelize this algorithm for the unweighted case; Klein and Subramanian [20] extend that algorithm to the integer-weighted case, but they also increase ϵ in the process and thus no longer obtain an exact hopset. In addition, Klein and Subramanian [20] show that their hopset can also be used to solve the exact SSSP algorithm by running $\tilde{O}(\sqrt{n})$ rounds of the Bellman-Ford algorithm and applying a scaling technique. More recently, Cao, Fineman, and Russell [6] give a method for constructing a $(n^{1/2+o(1)}, \epsilon)$-hopset in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span. Unsurprisingly, the problem of constructing hopsets has

a significantly lower complexity when the graph is undirected; there are low-work algorithms [10, 13, 14, 23] that construct hopsets with small or even constant hopbound for undirected graphs. The algorithms for producing hopsets are generally randomized, and the quality of the hopset that is returned is probabilistic.

1.1 Our results. Our results leverage recent breakthroughs on directed hopsets [6], an integer scaling technique [20], and a simple but powerful rounding technique called iterative gradual rounding [9].

Klein and Subramanian [20] provide a parallel algorithm with $\tilde{O}(m\sqrt{n})$ work that produces a "distance estimate," and they show how to use a distance estimate satisfying certain properties to solve the exact SSSP problem. In particular, their distance estimate is an integer $\tilde{d}(v)$ for each node $v \in V$ such that: (a) for each $v \in V$, $dist(v)/2 \leq \tilde{d}(v) \leq dist(v)$, (b) for each $(u, v) \in E$, $\tilde{d}(u) + w(u, v) \geq \tilde{d}(v)$. With such an estimate in hand, each edge (u, v) can be reweighted by setting $\tilde{w}(u, v) = \tilde{d}(u) - \tilde{d}(v) + w(u, v)$. The first property of the estimate provides a progress guarantee as the maximum shortest-path distance decreases by at least half after reweighting; the second property, called the triangle inequality, ensures that edge weights remain nonnegative. They show that computing $O(\log W)$ distance estimates is sufficient to solve the exact SSSP on graphs with nonnegative integer weights, where W denotes the maximum weight.

In more recent work, Chechik and Mukhtar [9] show how to compute a distance estimate in the CONGEST model, with a relaxed version of the triangle inequality, and in turn how to solve exact SSSP. Building off their iterative gradual rounding technique, we obtain a parallel algorithm for computing a distance estimate in a hopset graph. Our main result is as follows.

THEOREM 1.1. *Given an* $(h, \epsilon = \frac{1}{4\log(2nK)})$*-hopset graph* G *with non-negative integer edge weights from* $\{0, 1, 2, ...K\}$*, there is a deterministic parallel algorithm with* $\tilde{O}(m \log K)$ *work and* $\tilde{O}(h \log^2 K)$ *span that computes a distance estimate of the graph.*

This result forms the basis for our improved parallel SSSP algorithms. Cao, Fineman, and Russell [6] give an efficient construction for $(h = n^{1/2+o(1)}, \epsilon)$-hopset. Using that result, we are able to solve the parallel exact SSSP algorithm efficiently. In particular, adding a hopset to the input graph produces a hopset graph on which we can find a distance estimate. Our distance estimate has a slightly different form from Klein and Subramanian's[20], but we can tranform our estimate to match theirs through a simple transformation (see Section 3) and thus use their scaling technique. We thus obtain the following theorem.

THEOREM 1.2. *There is a randomized parallel algorithm that, given an n-node m-edge directed graph with non-negative integer edge weights from* $\{0, 1, 2, ..., W\}$*, solves the exact single-source shortest path problem with* $\tilde{O}(m \log W)$ *work and* $n^{1/2+o(1)} \log W$ *span, with high probability.*

We can also transform the algorithm from Theorem 1.1 to the distributed model. Our main result for distributed SSSP algorithms is given by Theorem 1.3. Notice that there is a known lower bound of $\Omega(\sqrt{n}+D)$ [24], and our algorithm matches the lower bound when $D = o(n^{1/4})$.

THEOREM 1.3. *There is a distributed randomized algorithm that, given an m-edge n-node directed graph with non-negative integer edge weights from* $\{0, 1, 2, ..., W\}$*, solves the exact single source shortest path problem with* $O((n^{2/5+o(1)}D^{2/5} + \sqrt{n} + D) \log W)$ *rounds of communication in the CONGEST model with high probability, where* D *is the hop diameter of the undirected communication network.*

Concurrent results [25]. At around the same time as we completed this work, Rozhoň et al. [25] concurrently obtained similar and more general results. Their work [26] provides several more refined notions of shortest-path approximations, and they show various reductions between the different approximations and exact solutions. In particular, they show how to use an algorithm for approximate SSSP on directed graphs with integer edge weights to solve exact SSSP. Notably, their results also imply a solution to exact shortest paths in the PRAM model in $\tilde{O}(m \log W)$ work and $n^{1/2+o(1)} \log W$ span. Their reduction is more general than ours in the sense that their reduction applies to any approximate SSSP solution, whereas ours relies specifically on hopsets. (All directed-graph approximations for SSSP currently use hopsets, but this may change in the future; indeed, for undirected graphs, some of the newest algorithms do not build hopsets [1, 21, 26].)

1.2 Overview of the technique. Here we provide an overview of our approach and why it works. Throughout this section, we refer to the input directed graph $G = (V, E, w)$ and a source node s.

As a minor deviation from Klein and Subramanian [20], we use overestimates instead of underestimates. That is, our goal is to produce a distance estimate $\hat{d}(v)$ for each node $v \in V$, such that (a) for each $v \in V$, $dist(v) \leq \hat{d}(v) \leq 2dist(v)$, (b) for each $(u, v) \in E$, $\hat{d}(u) + 2w(u, v) \geq \hat{d}(v)$. The overestimate can be used to construct their form of underestimate, and the overestimate is more familiar in the context of the approximate shortest paths. In light of prior work on approximate single-source shortest paths, it is not hard to satisfy the first inequality; in fact, any solution to approximate SSSP satisfies inequality (a). The main challenge is to produce a distance estimate that also satisfies the relaxed triangle inequality (b).

Main technique. To convey the main idea of building a distance estimate with the relaxed triangle inequality, consider a graph $G = (V, E, w)$ containing only positive integer edge weights. The algorithm maintains a distance estimate $d'(u) \geq dist(u)$ for each vertex. Initially, $d'(s) = 0$ and $d'(u) = +\infty$ for all other vertices. The algorithm then iteratively improves the estimates. To do so, we find it convenient to reason about weights of paths rather than single edge, and thus our update rule can be viewed as a generalization of "edge relaxations." Specifically, the algorithm iteratively produces a new estimate $\hat{d}$ using a generic rule of the form $\hat{d}(v) = min_{u \in V, u \rightsquigarrow v}(d'(u) + w(u \rightsquigarrow v))$ for each node v, where $u \rightsquigarrow v$ is an arbitrary path. Note that this formula generalizes edge relaxations, and hence, when a fixed point has been reached and no further updates are possible, $\hat{d}(v)$ satisfies both properties of a distance estimate.

It should not yet be clear, however, how to apply this update rule directly. Indeed, finding the minimum over all paths is equivalent to finding the shortest path to v. Instead, we relax the problem by rounding-up the edge weights in the graph. The algorithm proceeds in rounds numbered from $\log(nK)$ down to 1, where K is the maximum integer edge weight. In each round i, we round-up the edge weights $w(e)$ to $w_i(e)$ and apply the update rule $\hat{d}(v) = min_{u \in V, \text{paths } u \rightsquigarrow v}(d'(u) + w_i(u \rightsquigarrow v))$. We shall address what weights to use next, but before doing so let us consider a sufficient ending condition. As long as in the last round, $d'(u) \geq dist(u)$ for each node $u \in V$ and $w_1(e) \in [w(e), 2w(e)]$, our final $\hat{d}(u)$ will satisfy both properties of the distance estimate.

By carefully designing how to round edges, we can make it so that updating the current round estimate $\hat{d}$ given the previous round estimate d' can be done efficiently in parallel. Our rounding method basically follows the iterative gradual rounding method [9]. Here, we give a more natural formula to show the high-level idea. Our weights are parameterized by a value $h = \Omega(1)$, which shall be set according to the hopbound of the hopset graph. This value remains unchanged across the execution of the algorithm. Let $N = nK$ be the maximum path weight. Specifically, in round i (where i goes from $\log N$ down to 1), we use roughly the following weights for each $e \in E$:

$$w_i(e) = w(e) + \Theta\left(\frac{2^i}{h \log N}\right) + \left(\frac{\log N - i}{\log N}\right) w(e)$$

The last round is $i = 1$. During round 1, for any $h = \Omega(1)$, we have $w(e) \leq w_i(e) \leq 2w(e)$, and hence the distance estimate can be used to compute the SSSP.

The first term $\Theta(\frac{2^i}{h \log N})$ gives us a good lower bound for each edge. Given this lower bound, one can perform a natural extension of parallel breadth-first search to compute the distance $w_i(u \rightsquigarrow v)$, and this algorithm is efficient as long as the number of hops on the path is small. That is, we need only argue that shortest-path updates with respect to d' and w_i have at most $O(2^i)$ hops. Specifically, the main claim we prove is that when $w_i(u \rightsquigarrow v) > \Omega(2^i)$, the estimate achieved in round $(i + 1)$ is already better than any potential update performed in the next round i. (Recall that we are numbering rounds in decreasing order.) Thus, only short paths need be considered, and these paths cannot have too many hops given the lower bound on edge weights.

Completing the proof that only short paths need be considered leverages the $(\log N - i)w(e)/\log N$ term in the $w_i(e)$ expression coupled with the fact that we are operating on a hopset graph. In particular, this term increases in each subsequent round as i is decreasing. When a particular path is long, the weight of the path contributed by this last term thus increases significantly when moving from one round to the next. In contrast, the $\Theta(2^i/(h \log N))$ term charges the path proportionally to the number of hops. Because the graph is a hopset graph, and hence almost-shortest paths with few hops exist, we are able to argue that for long paths the increase to the $(\log N - i)w(e)$ term dominates any possible decrease in the $\Theta(2^i/(h \log N))$ term.

Why doesn't the same technique produce better bounds for undirected graphs? There are more efficient constructions of hopsets for undirected graphs, so it is natural to question if the same approach yields

better results for undirected graphs. Unfortunately, it does not. While we could indeed apply an undirected hopset to produce the first distance estimate, the issue is that the algorithm performs multiple iterations, reweighting the graph further on each iteration. The reweighting process eliminates the symmetry on edge weights, and thus undirected graphs effectively become directed. Therefore, after the first iteration, each subsequent distance estimate must be produced using a directed hopset instead of an undirected one.

Differences from Chechik and Mukhtar[9]. Our algorithm follows the general blueprint of Chechik and Mukhtar[9], but we highlight the following differences:

- *Limited distance search vs. unlimited search.* Although in the implementation, both our algorithm and Chechik and Mukhtar's algorithm update distances with distance-limited searches in the graph, Chechik and Mukhtar consider limited distance as a starting point. While distance-limited searches seem necessary to limit the number of rounds in the algorithm (i.e., the span), starting from distance-limited searches makes it difficult to realize the triangle inequality because the limited distance search might destroy the inequality for an edge. Chechik and Mukhtar show a different form of the triangle inequality than we do; theirs is distance limited, which makes it more complicated. We instead analyze the algorithm with respect to an unbounded search, and then we prove that the distance-limited searches produce the same result. This leads to a significantly shorter and simpler proof.
- *Using hopsets as a black-box.* Our algorithm and analysis explicitly leverage hopsets, and we use the hopset construction as a black box. This not only reduces the complexity of the algorithm and the proof, but also means that our algorithm immediately improves when a better hopset algorithm becomes available. In contrast, Chechik and Mukhtar's algorithm implicitly constructs a specific hopset, namely roughly the Klein and Subramanian's hopset [19], and their analysis only holds for the specific construction. Moreover, that hopset construction has high work, and it is thus not suitable for a low-work parallel algorithm.

2 Definitions and Preliminaries

A directed weighted graph G is a triple (V, E, w) where $w : E \to \mathbb{Z}$ is a weight function. We use $n = |V(G)|$ and $m = |E(G)|$ to denote the number of vertices and edges, respectively. For each $e \in E$, we denote the weight by $w(e)$. If $e \notin E$, then $w(e) = +\infty$. This paper addresses the shortest-path problem when edge weights are nonnegative integers, i.e., all edge weights are from $\{0, 1, 2, \ldots, W\}$, where W denotes the maximum edge weight.

A ***path*** is a sequence of vertices joined by edges; sometimes we refer to the path by the sequence of vertices and sometimes by the edges, depending on what is more convenient. We sometimes use $u \rightsquigarrow v$ to indicate a path from u to v. Given a graph $G = (V, E, w)$ and path $p = v_0 \to v_1 \to \cdots \to v_k$, we define the ***weight*** of the path as $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$, i.e., the sum of the weights of all edges on the path. We use $|p| = k$ to denote the number of edges, also called hops, along the path.

For a pair of nodes $u, v \in V$, the ***shortest-path distance*** from u to v is the minimum weight over all paths from u to v. We use $dist_w(u, v)$ to denote the shortest-path distance from u to v with respect to the weight function w. When the weight function w is clear in the context, we simply write $dist(u, v)$. If there is no u-to-v path, then we define $dist(u, v) = \infty$. Furthermore, $dist^{(h)}(u, v)$ denotes the minimum weight over all u-to-v paths that contain at most h edges. When referring to distances from a designated source vertex s, we use $dist(v)$ and $dist^{(h)}(v)$ as shorthands for $dist(s, v)$ and $dist^{(h)}(s, v)$, respectively.

For a subset $V' \subset V$, we use $G(V')$ to denote the vertex-induced subgraph, and we use $E(V')$ to denote the edges in the induced subgraph. We sometimes refer to lists of edges that may or may not exist in these subgraphs. For a candidate path p, we say $p \in E$ if all edges in p are indeed in E, and $p \notin E$ otherwise.

We say that a node u is an ***ancestor*** of v, and conversely, v is a ***descendant*** of u, if there is a directed path from u to v in G. Every node is an ancestor and the descendant of itself. We use $Anc(G, v)$ and $Des(G, v)$ to denote the set of all nodes that are ancestors or descendants, respectively, of v. When G is clear, we use $Anc(v)$ and $Des(v)$ as abbreviations.

We say that a function $g(n) = \tilde{O}(f(n))$ if there exists a constant k such that $g(n) = O(f(n) \log^k f(n))$. Similarly, we use $g(n) = \tilde{\Omega}(f(n))$ to mean that there exists a constant k such that $g(n) = \Omega(f(n) \log^k f(n))$. We also use $g(n) = \tilde{\Theta}(f(n))$ to mean $g(n) = \tilde{O}(f(n))$ and $g(n) = \tilde{\Omega}(f(n))$.

When we say that an algorithm achieves some performance $O(f(n))$ with high probability, we mean the following: for any particular choice of constant $c > 0$, the algorithm achieves performance $O(f(n))$ with probability

at least $1 - 1/n^c$. The constant inside the big-O may depend on the constant c.

3 The High-Level Algorithm

Our exact SSSP algorithm is an efficient realization of a general algorithm outlined in this section. This section presents the algorithm at a high level and does not consider any of the implementation details. Instead, we focus here on key properties provided by each step of the algorithm. We first confirm, by simple transformation to [20], that producing distance overestimates with the triangle inequality $O(\log W)$ times is sufficient to solve the exact SSSP problem. We next give a high-level algorithm for finding a distance estimate. In more detail, the algorithm uses iterative gradual rounding [9]; in each round, the algorithm improves distance estimate d_{i+1} computed in round $(i+1)$ to distance estimate d_i, where $d_i(v) \leq d_{i+1}(v)$. Finally (Section 3.3), we show that each iteration of this process has a key feature: if the graph is a hopset graph, d_i can be built from d_{i+1} without considering paths that are very long, and in particular the paths do not comprise many hops. We thus reduce the problem of producing a valid distance estimate to (1) finding a hopset, and (2) solving a particular hop-limited shortest-path problem. Section 4 explains how to solve this latter problem by using a generalized breadth-first search.

Given the reduction outlined above, solving exact SSSP for digraphs with edge weights from $\{0, 1, ..., W\}$ entails $O(\log W)$ applications of finding a hopset [5, 8] and modified breadth-first search (Section 4). The complexity of the algorithm thus depends on the efficiency of the realizations of modified breadth-first search in parallel and distributed models, which is discussed in Section 5.

3.1 Reducing exact SSSP to computing a distance estimate.

Given a directed graph $G = (V, E, w)$ with non-negative integer weights from $\{0, 1, 2, ..., K\}$ and a source node s, a ***distance estimate*** $\hat{d} : V \to \mathbb{R}$ is a function from vertices to real numbers (in our case, rational numbers) such that

(1) for all $v \in V$, $dist(v) \leq \hat{d}(v) \leq 2dist(v)$,

(2) for all $(u, v) \in E$, $\hat{d}(v) \leq \hat{d}(u) + 2w(u, v)$

Given a slightly different type of distance estimate, Klein and Subramanian [20] show how to solve exact SSSP by computing distance estimates and reweighting the graph $O(\log W)$ times, where W is the maximum weight in the original graph. To obtain a distance estimate $\tilde{d}$ that matches their form, we simply set $\tilde{d}(v) = \left\lceil \hat{d}(v)/2 \right\rceil$. It is not hard to verify that when all weights are integers: (1) for all $v \in V$, we have $dist(v)/2 \leq \tilde{d}(v) \leq dist(v)$, and (2) for all edges $(u, v) \in E$, we have $\tilde{d}(v) \leq \tilde{d}(u) + w(u, v)$. This is the form of distance estimate they use, so we can apply the same reduction.

We note that in the original graph, the maximum edge weight is W, which may not be polynomial in the size of the graph. However, when computing each distance estimate, it suffices to consider weights that are at most $K = O(n)$ (see Lemma 4.1 in Klein and Subramanian [20] or Theorem 2.1 in Forster and Nanongkai [15]). This means that the only way the maximum weight W affects the complexity of the algorithm is with respect to the number of iterations of scaling, i.e., the number of times to find a distance estimate.

3.2 Producing a distance estimate via iterative rounding.

This section gives our high-level algorithm for producing a distance estimate, shown as pseudocode in Algorithm 1. The algorithm takes as input a weighted hopset graph $G = (V, E, w)$, where all weights are from $\{0, 1, \ldots, K\}$, and a source $s \in V$. We assume throughout that G is an (h, ϵ)-hopset graph. That is, for any pair of nodes u, v, $dist(u, v) \leq dist^{(h)}(u, v) \leq (1 + \epsilon)dist(u, v)$, where ϵ is a sufficiently small value to be determined later.

Algorithm 1 proceeds in $\beta + 1 = \lfloor \lg(nK) \rfloor + 2$ rounds, numbered in decreasing order, where the last round (lines 8-11) is designed for 0-weight edges. Initially, the distance estimate for all nodes except the source node s is set to nK, which is larger than the maximum possible shortest-path distance (which is at most $(n-1)K$). In each round i, we round-up edge weights according to the formula shown in Line 5, which we shall discuss more in the next. We then update $d_i(v)$ from $d_{i+1}(u)$. In particular, for the node u that minimizes the following expression, the update is $d_i(v) \leftarrow d_{i+1}(u) + w_i(u \rightsquigarrow v)$. That is, $d_i(v)$ is set to the the distance to u in the previous round, plus the weight of the shortest path from u to v with respect to round-i weights w_i. How to perform these updates efficiently is the topic of Sections 4 and 5. The final round (lines 8-11), which we number as round 0, considers only the zero-weight edges in the graph. Here, distances are simply propagated

Algorithm 1 Producing a distance estimate with the triangle inequality
Input: An (h,ϵ)-hopset graph $G=(V,E)$, with edge weights $w_G : E \to \{0,1,...,K\}$ and a source node $s \in V$
Output: A distance estimate $\hat{d}$

1: $\delta \leftarrow \ln(\frac{3}{2}), \beta \leftarrow \lfloor \lg(nK) \rfloor + 1$
2: **for** $v \in V$ $d_{\beta+1}(v) \leftarrow nK$
3: $d_{\beta+1}(s) \leftarrow 0$
4: **for** $i \leftarrow \beta$ downto 1
5: **for** $e \in E$ $w_i(e) \leftarrow \frac{2^i}{16 \cdot h \cdot \beta} + (1+\delta/\beta)^{\beta-i} \cdot w(e)$
6: **for** $v \in V$
7: $d_i(v) \leftarrow min_{u \in V, u \rightsquigarrow v \in E}(d_{i+1}(u) + w_i(u \rightsquigarrow v))$
8: **for** $e \in E$
9: **if** $w(e) = 0$ **then** $w_0(e) \leftarrow 0$
10: **else** $w_0(e) \leftarrow +\infty$
11: **for** $v \in V$ $\hat{d}(v) \leftarrow min_{u \in V, u \rightsquigarrow v \in E}(d_1(u) + w_0(u \rightsquigarrow v))$

across zero-weight edges. That is to say, the final distance estimate $\hat{d} = d_0$ is computed in this round by setting $\hat{d}(v) \geq \min\{d_1(u) | u \rightsquigarrow v \text{ by 0-weight edges}\}$.

The specific weights used for w_i are $w_i(e) = \frac{2^i}{16h\beta} + \alpha(i) \cdot w(e)$, where we define $\alpha(i) = (1+\ln(3/2)/\beta)^{\beta-i}$. Notice that since round numbers decrease, the first term $2^i/(16h\beta)$ decreases as the algorithm progresses. In contrast, the second term is roughly $1 + (\beta - i)\Theta(1/\beta)$, which increases as the algorithm progresses.

Before proving that this algorithm yields a distance estimate, we first clarify some properties of the algorithm.

LEMMA 3.1. *The following properties hold:*

(3.1a) For any node $v \in V$ and $i \in [0,\beta]$, $d_{i+1}(v) \geq d_i(v)$;

(3.1b) For any path $u \rightsquigarrow v$ and $i \in [0,\beta]$, $d_i(v) \leq d_i(u) + w_i(u \rightsquigarrow v)$;

(3.1c) For any $i \in [0,\beta]$, $\alpha(i+1) < (1+\delta/\beta) \cdot \alpha(i+1) = \alpha(i)$ and $\alpha(0) \leq \frac{3}{2}$.

Proof. The first part of (3.1c) is trivial. The second (that $\alpha(0) \leq 3/2$) follows because $(1+\ln(3/2)/x)^x \leq 3/2$ for all $x > 0$. (3.1a) follows from the fact that the update rule results in $d_i(v) \leq d_{i+1}(v) + w_i(v \rightsquigarrow v) = d_{i+1}(v)$. For (3.1b), consider a node x such that $d_i(u) = d_{i+1}(x) + w_i(x \rightsquigarrow u)$. Then we have $d_i(v) \leq d_{i+1}(x) + w_i(x \rightsquigarrow u \rightsquigarrow v) \leq d_i(u) + w_i(u \rightsquigarrow v)$. ∎

Now, we show that the computed distance estimate satisfies the first constraint of distance estimates.

LEMMA 3.2. *Suppose $\epsilon \leq \delta/\beta$. Then for every $v \in V$, we have $dist(v) \leq \hat{d}(v) \leq 2dist(v)$*

Proof. First, we show $\hat{d}(v) \geq dist(v)$. Because all weights w_i are at least the original weight, for every path p we have $w_i(p) \geq w(p)$. It is thus easy to show by induction that $d_i(v)$ is at least the weight of the shortest path from s to v with respect to w.

Next, we want to show $\hat{d}(v) \leq 2dist(v)$. There are two cases. First, consider the case that $dist(v) = 0$. Then there is a path $s \rightsquigarrow v$ that contains only zero-weight edges. Considering lines 8–11, every zero-weight path $s \rightsquigarrow v$ is preserved with respect to w_0 and hence $\hat{d}(v) = d_1(s) + w_0(s \rightsquigarrow v) = 0$.

The second case is that $dist(v) \geq 1$. Note G is an (h,ϵ) hopset graph, so there exists a path p from s to v such that p contains at most h hops and $w(p) \leq (1+\epsilon)dist(v)$. From (3.1b), we have $\hat{d}(v) \leq d_1(v) \leq d_1(s) + w_1(p) \leq$

$w_1(p)$. We only need to bound $w_1(p)$:

$$\begin{aligned}
w_1(p) &= \sum_{e\in p}(2/(16h\beta)) + \alpha(1)w(e)) \\
&= \sum_{e\in p}(2/(16h\beta)) + \alpha(1)\sum_{e\in p} w(e) \\
&< 1/8 + \alpha(1)\cdot w(p) && \text{(by } |p| \le h \text{ and } \beta \ge 1\text{)} \\
&\le (1/8)dist(v) + \alpha(1)\cdot(1+\epsilon)\cdot dist(v) && \text{(by assumptions } dist(v) \ge 1 \text{ and } p \text{ is } 1+\epsilon \text{ approximate)} \\
&\le (1/8)dist(v) + \alpha(0)\cdot dist(v) && \text{(if } 1+\epsilon \le 1+\delta/\beta\text{)} \\
&\le 2dist(v) && \text{(}\alpha(0) \le 3/2 \text{ by (3.1a))}
\end{aligned}$$

□

Next, we show that the second constraint of being a distance estimate is also satisfied.

LEMMA 3.3. *Suppose $\epsilon \le \delta/\beta$. Then for every $(u,v) \in E$, $\hat{d}(v) \le \hat{d}(u) + 2\cdot w(u,v)$.*

Proof. If $w(u,v) = 0$, then from (3.1b), we have $\hat{d}(v) = d_0(v) \le d_0(u) + w_0(u,v) = \hat{d}(u) + w(u,v)$.

Otherwise $w(u,v) \ge 1$, consider the node y such that $\hat{d}(u) = d_1(y)$. (It may be the case that $y = u$). Let p_u be a $y \rightsquigarrow u$ path containing only zero-weight edges, and let p_v be the same path but with (u,v) appended. Then $w(p_v) = w(u,v)$. Because the graph is a hopset graph, there must be a path p from y to v containing at most h hops with $w(p) \le (1+\epsilon)\cdot w(p_v) = (1+\epsilon)w(u,v)$. Applying the same algebraic steps as Lemma 3.2, we conclude that $w_1(p) \le 2w(u,v)$. It follows from (3.1b) that $d_1(v) \le d_1(y) + w_1(p) \le d_1(y) + 2w(u,v) = \hat{d}(u) + 2w(u,v)$. Finally, $d_0(v) \le d_1(v)$ from (3.1a), so we have $\hat{d}(v) \le d_1(v) \le \hat{d}(u) + 2w(u,v)$ as desired. □

THEOREM 3.1. *Algorithm 1 computes distance estimate satisfying (1) and (2).*

Proof. The theorem is implied by Lemma 3.2 and Lemma 3.3. □

3.3 Key properties of distance updates. Thus far, we have not discussed how to efficiently update d_i from d_{i+1}. Here we focus on showing that Algorithm 1 has two key properties.

First, we show that all distance updates can be performed by only considering paths that are not too long. That is to say, there is a relatively short path $p : u \rightsquigarrow v$ for which $d_i(v) = d_{i+1}(u) + w_i(p)$. By short here, we mean both that it does not have too many hops, and that the total weight is not too high. Because w_i provides a lower bound on all edge weights, this property suggests that it may be possible to realize the distance updates by modifying breadth-first search (BFS). As such, parallel and distributed algorithms should be possible. Though it turns out that this property alone is not sufficient.

Second, we show that the distance estimates do not decrease too quickly going from one iteration to the next. That is, if $d_i(v) = d_{i+1}(u) + w_i(u \rightsquigarrow v)$, then it must be the case that $d_{i+1}(u)$ is not too much smaller than $d_{i+1}(v)$. This property is also necessary to obtain an efficient parallel or distributed realization, but the reason is a bit more subtle. Roughly speaking, our modified BFS algorithm (Section 4) needs to operate at multiple offsets, i.e., from vertices at a wide range of distances, simultaneously. We cannot afford to include the entire graph in every BFS. But because distances do not decrease by too much, each vertex need only be added to a constant number of concurrent BFS executions.

Property 1: update paths are short. Consider a round i with $i > 0$. Based on the formula for rounding up edge weights, we know that each edge weight is lower bounded by $\Omega(2^i/(h\beta))$. The following lemma states that if $u \rightsquigarrow v$ can be used to improve the distance to vertex v, then $w_i(u \rightsquigarrow v)$ is upper bounded by $O(2^i)$. Coupled with the lower bound on edge weights, this lemma directly implies that the number of hops on the path $u \rightsquigarrow v$ is at most $O(h\beta)$.

LEMMA 3.4. *Suppose that $\epsilon \le 1/(4\beta)$. For any node $u,v \in V$, path $p = u \rightsquigarrow v$ and $i \in [1,\beta]$, if $d_i(v) = d_{i+1}(u) + w_i(p)$, then $w(p) < 2^i$ and $w_i(p) < 2^{i+1}$.*

Proof. First, we show that $w(p) < 2^i$. For $i = \beta$, we trivially have $w(p) \leq nK = 2^\beta$, as K is the maximum edge weight. When $i \in [1, \beta)$, according to the hopset assumption, there is a path p' from u to v that contains at most h hops and $w(p') \leq (1+\epsilon)dist(u,v) \leq (1+\epsilon) \cdot w(p)$. Based on (3.1a) and (3.1b), we have $d_i(v) \leq d_{i+1}(v) \leq d_{i+1}(u) + w_{i+1}(p')$. Combining with the assumption $d_i(v) = d_{i+1}(u) + w_i(p)$, we have $w_i(p) \leq w_{i+1}(p')$. Now we give a lower bound for $w_i(p)$ and an upper bound for $w_{i+1}(p')$. We have $w_i(p) \geq \alpha(i)w(p)$ for the lower bound. For the upper bound,

$$w_{i+1}(p') \leq \frac{2^{i+1}}{16 \cdot h \cdot \beta} h + \alpha(i+1)w(p') \leq \frac{2^i}{8\beta} + \alpha(i+1)(1+\epsilon)w(p)$$

So combining the upper and lower bounds, we have

$$\begin{aligned}
w_i(p) \leq w_{i+1}(p') &\Rightarrow \alpha(i)w(p) \leq \frac{2^i}{8\beta} + \alpha(i+1)(1+\epsilon)w(p) \\
&\Rightarrow \alpha(i+1)w(p)\left[(1+\delta/\beta) - (1+\epsilon)\right] \leq \frac{2^i}{8\beta} \\
&\Rightarrow \alpha(i+1)w(p)(\delta/\beta - \epsilon) \leq \frac{2^i}{8\beta} \\
&\Rightarrow 8\alpha(i+1)w(p)(\delta - \beta\epsilon) \leq 2^i \\
&\Rightarrow w(p) < 2^i \qquad \text{if } 8\alpha(i+1)(\delta - \beta\epsilon) > 1
\end{aligned}$$

Using $\alpha(i+1) \geq 1$, one can inspect that the condition holds as long as $\beta\epsilon < \ln(3/2) - 1/8$, which is true if $\epsilon < 1/(4\beta)$.

We now have the bound on $w(p)$. Since the $d_i(v) = d_{i+1}(u) + w_i(p)$, we also have

$$w_i(p) \leq w_i(p') \leq \frac{2^i}{16 \cdot h \cdot \beta} \cdot h + \alpha(i) \cdot (1+\epsilon) \cdot w(p) \leq 2^i/16 + (3/2)w(p) < 2^{i+1} \ ,$$

where the second-to-last step applies $\alpha(i)(1+\epsilon) \leq \alpha(0) \leq 3/2$ from (3.1c) and the last step uses the upper bound on $w(p)$. ∎

COROLLARY 3.1. *Suppose that $\epsilon \leq 1/(4\beta)$. For any nodes $u, v \in V$, path $p = u \rightsquigarrow v$ and i, and $i \in [1, \beta]$, if $d_i(v) = d_{i+1}(u) + w_i(p)$, then $|p| = O(h\beta)$.*

Proof. For every edge, $w_i(e) \geq 2^i/(16h\beta)$. We have $w_i(p) < 2^{i+1}$ from Lemma 3.4. Thus, $|p| \leq 2^{i+1}/(2^i/(16h\beta)) = 32h\beta$. ∎

Property 2: distances do not decrease too rapidly. Here we focus on showing that $d_i(v)$ is not very different from $d_{i+1}(v)$. Specifically, the distance to v can only decrease by $O(2^i)$ when going from round $i+1$ to round i. Thus, our modified BFS (Section 4) can group vertices by distance d_{i+1} in order to compute d_i.

LEMMA 3.5. *Suppose $\epsilon \leq 1/(4\beta)$ and consider any nodes $u, v \in V$, $i \in [1, \beta)$, and path $p = u \rightsquigarrow v$. If $d_i(v) = d_{i+1}(u) + w_i(p)$, then $d_{i+1}(v) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1})$. Furthermore, for every node x on this path p, we also have $d_{i+1}(x) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1})$.*

Proof. Based on (3.1a), we have $d_{i+1}(v) \geq d_i(v) \geq d_{i+1}(u)$, where the rightmost inequality comes from the assumption on $d_i(v)$ in the lemma statement. This gives us the claimed lower bound on $d_{i+1}(v)$; we next turn to the upper bound. By Lemma 3.4, we know $w(p) < 2^i$. Since the graph is an (h, ϵ)-hopset graph, there is a path p' from u to v that contains at most h hops and with $w(p') \leq (1+\epsilon) \cdot w(p) < (1+\epsilon) \cdot 2^i$. Based on (3.1b), we know that $d_{i+1}(v) \leq d_{i+1}(u) + w_{i+1}(p')$. We only need to bound $w_{i+1}(p')$:

$$w_{i+1}(p') \leq \frac{2^{i+1}}{16 \cdot h \cdot \beta} \cdot h + \alpha(i+1) \cdot (1+\epsilon) \cdot 2^i < (1/8)2^i + (3/2)2^i < 2^{i+1}$$

Algorithm 2 Distance estimate updates on an induced subgraphs with nonzero weights
Input: Graph $G(A_j) = (A_j, E(A_j), w_i)$, for any $v \in A_j$, $d_{i+1}(v) \in [(j-1)\cdot 2^{i+1}, (j+1)\cdot 2^{i+1})$.
Output: For all $v \in A_j$, a distance $d_i(v) = min_{u\in A_j, u\rightsquigarrow v\in E(A_j)} d_{i+1}(u) + w_i(u \rightsquigarrow v)$

function UPDATEINDUCEDGRAPHS($G(A_j) = (V, E, w_i), d_{i+1}$)
  $\Delta = \frac{2^i}{16\cdot h\cdot \beta}$
  for $k \in [0, \lfloor 2^{i+2}/\Delta \rfloor]$ $F(k) \leftarrow \emptyset$
  for $v \in V$ $d_i(v) \leftarrow d_{i+1}(v), F(\text{INDEX}(d_i(v))) \leftarrow F(\text{INDEX}(d_i(v))) \cup \{v\}$
  for $k \in [0, \lfloor 2^{i+2}/\Delta \rfloor]$ and $F(k) \neq \emptyset$
    for $v \in F(k)$ that v is not finalized
      mark v as finalized
      for $(v, x) \in E$ and x that are not finalized
        $d_i(x) = min(d_i(x), d_i(v) + w_i(v, x))$
        $F(\text{INDEX}(d_i(x))) \leftarrow F(\text{INDEX}(d_i(x))) \cup \{x\}$
function INDEX($d_i(v)$)
  $\Delta = \frac{2^i}{16\cdot h\cdot \beta}$, $b = (j-1)\cdot 2^{i+1}$
  return $\lfloor (d_i(v) - b)/\Delta \rfloor$

as long as $\epsilon \leq \delta/\beta$. Combining the above inequalities, we have $d_{i+1}(v) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1})$.

Now consider any node x on the path $p = u \rightsquigarrow v$. Let $p_u = u \rightsquigarrow x$ and $p_v = x \rightsquigarrow v$ be the subpaths of p split at node x. We claim that $d_i(x) = d_{i+1}(u) + w_i(p_u)$ (which we shall prove last). As such, it follows from the what was just proved that $d_{i+1}(x) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1})$.

Finally, we prove that $d_i(x) = d_{i+1}(u) + w_i(p_u)$. Suppose for the sake of contradiction that $d_i(x) \neq d_{i+1}(u) + w_i(p_u)$. Then by construction $d_i(x) < d_{i+1}(u) + w_i(p_u)$. From (3.1b), $d_i(v) \leq d_i(x) + w_i(p_v) < d_{i+1}(u) + w_i(p)$. This contradicts the fact that $d_i(v) = d_{i+1}(u) + w_i(p)$ by assumption in the lemma statement. It thus must be the case the $d_i(x) = d_{i+1}(u) + w_i(p_u)$, which completes the proof. ∎

4 Updating the Distance Estimate d_i

We have already shown the correctness of Algorithm 1. However, we have not yet discussed how to update $d_i(v)$ efficiently. In particular, the straightforward implementation would be to run Dijkstra's algorithm to compute $d_i(v)$ in each iteration of Algorithm 1. Given that we set out to solve the exact SSSP problem in the first place, using Dijkstra's algorithm as a black box is a nonstarter.

This section provides an efficient algorithm for computing d_i from d_{i+1}. This algorithm can be viewed either as a modified breadth-first search (BFS) or a restricted version of Dijkstra's algorithm. But parallel BFS has a span that is proportional to the distance, which could be linear in the graph size. The key feature that allows us to obtain efficient parallel and distributed algorithms is that, due to Lemmas 3.4 and 3.5, our algorithm only performs BFS out to $O(h\beta)$ hops. This section presents the algorithm at a high level—we defer the parallel or distributed details, including the work and span analysis, to Section 5.

We divide the description of the algorithm into two cases. First, we discuss how to update d_i in rounds β down to 1 of Algorithm 1. During these iterations, all weights w_i in the graph are rounded up to some nonzero values. Second, we discuss how to compute $d_0 = \hat{d}$ in round 0; this round is special because the distance update considers only the zero-weight edges. Chechik and Mukhtar [9] use a similar method in the broadcast CONGEST model.

4.1 Updating d_i in rounds $i \geq 1$.

Here we consider how to construct d_i from d_{i+1} given that all edge weights $w_i(e) > 0$.

First, we consider specific subgraphs based on to the previous distances d_{i+1}. Specifically, for positive integers $j \leq \lceil nK/2^i \rceil$, we define $A_j = \{y | (j-1)2^{i+1} \leq d_{i+1}(y) < (j+1)2^{i+1}\}$ to be the set of vertices with previous-round distances d_{i+1} falling in the interval $j2^{i+1} \pm 2^{i+1}$. Notice that these intervals overlap, and a vertex that belongs to A_j may also belong to A_{j-1} or A_{j+1}, and each vertex may belong to A_j and either A_{j+1} or A_{j-1}.

Next, we compute d_i with respect to each induced subgraph $G(A_j)$ using weights w_i, and the computation

on each subgraph is independent. Algorithm 2 provides pseudocode for the update on each induced subgraph. At a high level, this algorithm simulates a modified breadth-first search (BFS) or restricted Dijkstra's algorithm. The minimum edge weight $\Delta = 2^i/(16h\beta)$ allowed by weight function w_i corresponds to the unit edge weight in the BFS. But in our case, some of the weights may be larger, and hence the modification. Moreover, there is no designated source vertex, but instead the d_{i+1} distances act as starting points. Nevertheless, the general structure is similar.

In more detail, Algorithm 2 proceeds in rounds. We classify the nodes in the graph based on their current distance d_i, which is initialized to d_{i+1}. Specifically, we say that a node with $d_i(v) \in [b, b+(k+1)\Delta)$ has ***index*** k, where $b = (j-1)2^{i+1}$ is the minimum d_{i+1} distance of a vertex in A_j. Like BFS, during round k, all vertices with index k (i.e., distance in $[b, b+(k+1)\Delta)$ are finalized. Like Dijkstra's algorithm, we perform edge relaxations, and a vertex's distance may change multiple times if it has multiple incoming edges. To facilitate a simpler parallel implementation, we maintain a set $F(k)$ to contain all vertices with index k. Thus, during round k, the algorithm finalizes the distances to vertices in the frontier $F(k)$ and relaxes their outgoing edges, thus possibly moving some other vertices to earlier index sets than they previously belonged to. To avoid the nuisance of implementing deletions in the parallel sets, we allow vertices to belong to multiple sets $F(k)$, and we instead mark vertices as finalized the first time they are processed in a frontier. In this way, vertices may be considered once for each of their incoming edges (once per relaxation), but their outgoing edges are only processed when the vertex is first finalized.

Finally, after computing d_i with respect to all of the induced subgraphs, we combine the results. Because each vertex v belongs to at most two induced subgraphs, we simply choose the minimum $d_i(v)$ computed for that vertex in either of the two subproblems.

We next turn to correctness. First we show that Algorithm 2 correctly computes d_i with respect to the induced subgraph A_j. Second, we show that d_i is correct overall, i.e., that working with respect to induced subgraphs does indeed give the correct answer. The key to the argument is leveraging Lemma 3.5. In particular, for any path $p = u \rightsquigarrow v$ with $d_i(v) = d_{i+1}(u) + w_i(p)$, Lemma 3.5 implies that there exists a value j such that p is fully contained in $G(A_j)$.

LEMMA 4.1. *Consider a call to* UPDATEINDUCEDGRAPHS*$(G(A_j) = (A_j, E(A_j), w_i))$. When the algorithm completes, for each node $v \in A_j$, we have $d_i(v) = min_{u\in A_j, u\rightsquigarrow v\in E(A_j)}(d_{i+1}(u) + w_i(u \rightsquigarrow v))$.*

Proof. Proof by induction on round number k. We want to show that at the k-th round, for any node v with $d_i(v) < b + (k+1)\Delta$, $d_i(v) = min_{u\in A_j}(d_{i+1}(u) + w_i(u \rightsquigarrow v))$. Note that because the algorithm uses edge relaxations, at any time of updating $d_i(v)$ we always have $d_i(v) \geq min_{u\in A_j}(d_{i+1}(u) + w_i(u \rightsquigarrow v))$; thus, we only need to show $d_i(v) \leq min_{u\in A_j}(d_{i+1}(u) + w_i(u \rightsquigarrow v))$ when v is finalized.

The key observation is for each $e \in E$, $w_i(e) \geq \Delta$. In the first round, when $k = 0$, if $d_i(v) < b + \Delta$, then $d_i(v) = d_{i+1}(v)$ is correct. Otherwise, there would be some node u such that $d_{i+1}(u) = d_i(v) - w_i(u \rightsquigarrow v) \leq d_i(v) - \Delta < b$, which contradicts the fact that d_{i+1} is at least b for all nodes in A_j.

For the inductive step, we assume that by the end of the $(k-1)$-th round, all nodes with $d_i(v) < b + k\Delta$ are correctly updated. Now in the k-th round, for node v with $d_i(v) \in [b + k\Delta, b + (k+1)\Delta)$. If $d_i(v) = d_{i+1}(v)$, we have already put it in $F(k)$ and will finalize v; otherwise, we have $d_i(v) = d_{i+1}(u) + w_i(u \rightsquigarrow v)$. Consider the penultimate node y on the path $u \rightsquigarrow v$ and let $u \rightsquigarrow y$ be the sub-path of $u \rightsquigarrow v$ from u to y. We have $d_i(y) \leq d_{i+1}(u) + w_i(u \rightsquigarrow y) \leq d_i(v) - w_i(y, v) < b + (k+1)\Delta - \Delta$. Based on induction, y is correctly finalized, and $d_i(y) \leq d_{i+1}(u) + w_i(u \rightsquigarrow y)$. When we finalize y, we set $d_i(v) = d_i(y) + w_i(y, v) \leq d_{i+1}(u) + w_i(u \rightsquigarrow v)$ and put v in $F(k)$. Now in the k-th round, we extract v from $F(k)$ and finalize $d_i(v) \leq d_{i+1}(u) + w_i(u \rightsquigarrow v)$. ∎

THEOREM 4.1. *If we call* UPDATEINDUCEDGRAPHS *on subgraphs $G(A_1), G(A_2), ..., G(A_{\lceil nK/2^i \rceil})$ and set $d_i(v)$ to the minimum value of d_i returned by different subgraphs, then $d_i(v) = min_{u\in V, u\rightsquigarrow v\in E} d_{i+1}(u) + w_i(u \rightsquigarrow v)$.*

Proof. Note that we always have $d_i(v) \geq min_{u\in V, u\rightsquigarrow v\in E} d_{i+1}(u) + w_i(u \rightsquigarrow v)$. We only need to show that $d_i(v) \leq min_{u\in V, u\rightsquigarrow v\in E} d_{i+1}(u) + w_i(u \rightsquigarrow v)$.

Assume that we use p to update v in Algorithm 1 and $d_{i+1}(u) \in [k \cdot 2^{i+1}, (k+1) \cdot 2^{i+1})$. By lemma 3.5, for any node x on the path p, we have $d_{i+1}(x) \in [k \cdot 2^{i+1}, (k+2) \cdot 2^{i+1})$. In other words, the path p is included in the graph $G(A_{k+1})$.

Based on Lemma 4.1, we have $d_i(v) = min_{t\in[1,\lceil nW/2^i \rceil], u'\in A_t, u'\rightsquigarrow v\in E(A_t)} d_{i+1}(u') + w_i(u' \rightsquigarrow v) \leq min_{u'\in A_{k+1}, u'\rightsquigarrow v\in E(A_{k+1})}(d_{i+1}(u') + w_i(u' \rightsquigarrow v)) \leq d_{i+1}(u) + w_i(p) = min_{u\in V, u\rightsquigarrow v\in E} d_{i+1}(u) + w_i(u \rightsquigarrow v)$. ∎

Algorithm 3 Propagating distances over zero-weight edges
Input: Graph $G = (V, E, w)$ such that for any $e \in E$, we have $w(e) = 0$; D is the set containing possible d_1 values
Output: For all v, a distance estimation $\hat{d}(v) = min_{u \in V, u \rightsquigarrow v \in E} d_1(u) + w_0(u \rightsquigarrow v)$

```
1: function UpdateZeroWeightGraphs(G = (V, E, w_0), D)
2:     d_mid = the median of D
3:     if |D| = 1 then
4:         for v ∈ V  d̂(v) ← d_mid
5:     else
6:         S ← {v | v ∈ V, d_1(v) ≤ d_mid}
7:         V_l ← Des(S), V_u = V\V_l
8:         D_l ← {d | d ∈ D, d ≤ d_mid}, D_u = D\D_l
9:         UpdateZeroWeightGraphs(G(V_l), D_l)
10:        UpdateZeroWeightGraphs(G(V_u), D_u)
```

For our eventual performance results (Section 5), it is also important that Algorithm 2 does not perform too many iterations. The intuition outlined in Section 3.3 was that Corollary 3.1 limits the number of hops along any of the update paths. But this intuition is, in fact, subsumed by the fact that the subgraph computations are sufficient—by construction, Algorithm 2 only needs $O(h\beta)$ iterations before reaching the largest possible distance in that subgraph.

4.2 Updating $\hat{d}$ in the final round. We now consider how to produce $\hat{d} = d_0$ from d_1. Here, the algorithm starts from a graph with all nonzero-weight edges removed. The goal is then to calculate $\hat{d}(v)$ equal to the minimum d_1 value of its ancestor nodes. Our method follows Chechik and Mukhtar's algorithm [9].

We use a divide-and-conquer method to solve the problem. In the algorithm 3, we will maintain a set D such that D contains all possible d_1 values for $v \in V$. At the beginning, for each node v, we call UpdateZeroWeightGraphs$(G, \{d_1(v) \mid v \in V\})$, where D contains all possible d_1 value. If there is only one value in D, then all nodes' $\hat{d}$ are this value. Otherwise, we first sort D. Let d_{mid} be the median of D and S be the set that contains all nodes whose d_1 is at most d_{mid}. The descendants of S, which we denote by V_l, are the nodes v that should have $\hat{d}(v) \le d_{mid}$. Conversely, $V_u = V \backslash Des(S)$ is the set of nodes with higher distance. The Algorithm 3 recurs on induced graphs with D_l and D_u set.

The only complication with respect to producing the parallel or distributed version of this algorithm is finding the set V_l. We discuss this in Section 5, but suffice it to say that this can be solved by the single-source reachability problem, which is a special case of approximate SSSP. As such, this problem can be solved using the same tools we are already applying.

The following lemma plays a key role in showing the correctness and the efficient implementation.

LEMMA 4.2. *Fix two nodes $u, v \in V$ for graph $G = (V, E, w_0)$. If $\hat{d}(v) = d_1(u)$, then there exists a zero-weight path p from u to v in $E(G)$ with $|p| \le h$. Furthermore, consider every call* UpdateZeroWeightGraphs(G', D')*: if $v \in V(G')$, then $d_1(u) \in D'$ and the path $p \in E(G')$.*

Proof. By assumption that $\hat{d}(v) = d_1(u)$, there exists a path $u \rightsquigarrow v$ with weight 0. Because the graph is a hopset graph, there is also a path p from u to v containing at most h hops with $w(p) \le (1+\epsilon)w(u \rightsquigarrow v) = 0$. Since p contains only zero-weight edges, it is also include in the graph of zero-weight edges.

To show the second part, we will use induction over level of recursion, from top to bottom. At the beginning, we know D contains all possible d_1 value and all nodes are in the graph, and hence the claim holds trivial. For the inductive step, consider a call to UpdateZeroWeightGraphs$(\bar{G}, \bar{D})$ with $v \in V(\bar{G})$. Assume $p \in E(\bar{G})$, $d_1(u) \in \bar{D}$. The goal is to show that all nodes on p go to the same subproblem, and moreover that the corresponding D set contains the value $d_1(u)$. (The only distance value we need to argue is passed to the appropriate subproblem is $d_1(u)$ because for every node x on p, $\hat{d}(x) = d_1(u)$.)

Let $\bar{S}$ be the set of vertices with distance less than the median of $\bar{D}$. Let x be an arbitrary node on path p We have two cases:
Case 1. Suppose $u \in Des(\bar{S})$, i.e., $d_1(u) \le d_{mid}$. Because p is a 0-weight path, we also have $x \in Des(u) \subseteq Des(\bar{S})$.

Moreover, $u \in Des(\bar{S})$ implies $d_1(u) \in \bar{D}_l$.
Case 2. Suppose $u \notin Des(\bar{S})$, and hence $u \in V_u$. By the inductive assumption, $d_1(u) \in \bar{D}$, and so $d_1(u) > d_{mid}$. Because $\hat{d}(x) = d_1(u)$, we have $d_1(x) \geq d_1(u)$. It thus follows that $x \in V_u$ as well.
In both cases, all nodes x on p go to the same subproblem as u, as does the distance $d_1(u)$. ∎

Using the lemma 4.2, we next show that Algorithm 3 is correct.

LEMMA 4.3. *Let $D = \{d_1(v)|v \in V\}$. When the top-level call to* UPDATEZEROWEIGHTGRAPHS$(()V, E, w_0), D)$ *completes, we have $\hat{d}(v) = min_{u \in V, u \rightsquigarrow v \in E} d_1(u) + w_0(u \rightsquigarrow v)$. Moreover, the recursion depth of this algorithm is $O(\log n)$.*

Proof. To bound the recursion depth, consider the D array. The size of D reduces by half in each level of recursion. So we have $O(\log |D|) = O(\log n)$ levels of recursion.

Now consider any node u such that $\hat{d}(v) = d_1(y) + w_0(u \rightsquigarrow v)$. Then there is a zero-weight path from u to v. Thus, we can apply Lemma 4.2 to conclude that when v arrives at a base case, $d_1(u)$ is the only distance in its subproblem. And thus $\hat{d}(v) \leftarrow d_1(u)$. ∎

5 Translating to the Parallel and Distributed Models

In this section, we describe the details of our exact SSSP algorithm in the parallel and distributed model. In the parallel model, we only need to repeat constructing the hopset graph and computing the distance estimate some times. In the distributed model, we have to use the framework of Forster and Nanongkai and compute exact SSSP on the skeleton graph instead of the original graph.

5.1 Implementation in the parallel model.

We consider the work-span model[11], where the work is defined as the total number of instructions executed across all processors, and the span is the length of the critical path (i.e., the length of the longest chain of sequential dependencies).

Updating d_i without zero-weight edges. To construct the induced graph, we first put all nodes in an array and sort the array based on $d_{i+1}(v)$. For each A_j, we can simply mark the first and last nodes and compute the size of A_j. we can call a new array to hold all nodes and each node can decide its location by subtracting its index with the first node index in the sorted array. Let m_j and n_j be the number of edges and nodes in the induced graph $G(A_j)$, each node only appears in at most two induced graphs, so $\sum m_j = O(m)$ and $\sum n_j = O(n)$. In total, it takes $O(\sum m_j) = O(m)$ work and $\tilde{O}(1)$ span to construct all induced graphs.

In each induced graph, Algorithm 2 is naturally parallelized. In each induced graph, we need to search with depth $O(2^i/\Delta) = O(h\beta)$, and each edge will be accessed once. We only need to focus on the following point; We cannot afford to transverse all the elements in F. If $F(k)$ is empty, we should skip it. Otherwise, the work is at least $\Omega(h\beta)$ in each induced graph. We aim to take $\tilde{O}(m_j)$ work for each induced graph. To implement $F(k)$ efficiently, we can use the parallel ordered set as a tool. Specifically, there exist implementations of parallel ordered sets [2] that support the following operations:

- Initialization, given a vector of size n, we can construct a new ordered set in $\tilde{O}(n)$ work and $\tilde{O}(1)$ span and get an identifier of the ordered set.
- We can perform merge in $O(m \log(n/m + 1))$ work and $O(\log m \log n)$ span for sets of size m and n where $n \geq m$.
- We can enumerate all elements in a size n set in $O(n)$ work and $\tilde{O}(1)$ span.
- We can extract the minimum value with $O(1)$ work and $\tilde{O}(1)$ span.
- Given a key, we can return two ordered sets T_L and T_R such that all elements in T_L (T_R) are less (larger) than the key with $\tilde{O}(1)$ work and $\tilde{O}(1)$ span.

Now we can use the parallel ordered set to implement $F(k)$. In lines 3-4 of Algorithm 2, we simply compute INDEX$(d_i(v))$ and initialize the parallel ordered set with element $(index(d_i(v)), v)$. In lines 5 - 10, to transverse

the next non-empty $F(k)$, we can first extract the minimum value (k, v) from the parallel ordered set, then we will split the parallel ordered set by using the key $(k + 0.5, *)$ and get all the items whose index once was k. Then we can copy $F(k)$ into a vector by enumerating the returned parallel ordered set concatenating the minimum index node. There might be some finalized nodes, so we need a vector to distinguish all finalized nodes. We will sort all nodes from $F(k)$ according to whether it is finalized or not. Now, all vertices needed to update are stored in a vector, and each vertex might have several outgoing edges. We can use the prefix sum to compute the location of each outgoing edge and then update the outgoing nodes. Again, there might be some finalized nodes and we do not need to insert them back into the parallel set. We can filter all finalized nodes by sorting and initializing another parallel set and merging it with the original one. Throughout the process, there may be duplicates *index* for each node, but the work is still bounded, since we only transverse each edge once, which in total, $\tilde{O}(m_j)$ work. The span in each iteration is $\tilde{O}(1)$ and in total is $\tilde{O}(h\beta)$. In summary, updating all induced graphs takes $\sum \tilde{O}(m_j) = \tilde{O}(m)$ work and $\tilde{O}(h\beta)$ span. This gives us the following lemma:

LEMMA 5.1. *Algorithm 2 can be implemented in $\tilde{O}(m)$ work and $\tilde{O}(h \log K)$ span in the work-span model.*

Updating $\hat{d}$ on zero-weight graph. All codes in Algorithm 3 are straightforward, except that we need to compute $V_l \leftarrow Des(S)$. Lemma 4.2 gives us a way to compute V_l. In each UPDATEZEROWEIGHTGRAPHS(G', D'), if $v \in V_l$, there must be a path p from $u \in S$ to v that contains at most h edges. We can add a virtual source node s' and add edges from s' to all nodes in S, then a h-hop BFS [20] from s' can solve the reachability problem for V_l. The h-hop BFS takes $\tilde{O}(m_j + h)$ work and $O(h)$ span, where m_j is the number of edges in the induced graph. Note that to avoid $O(h)$ work on each induced graph, when $m_j < h$, we do $O(m_j)$-hop BFS and the work is $\tilde{O}(m_j)$. To recurse on the induced graph, we again sort the nodes and construct the subgraphs with $\tilde{O}(m_j + n_j)$ work and $\tilde{O}(1)$ span. Each level of recursion is a partition of the graph and we have $O(\log n)$ level of recursion. This gives us the following lemma:

LEMMA 5.2. *Algorithm 3 can be implemented in $\tilde{O}(m)$ work and $\tilde{O}(h)$ span in the work-span model.*

Combining together. Note that when we update the distance estimation, we have to run $O(\log K)$ rounds of Algorithm 2 and $O(1)$ rounds of the Algorithm 3.

LEMMA 5.3. *Given a $(h, \epsilon = \frac{1}{4\log(2nK)})$-hospset graph G with non-negative integer weights from $\{0, 1, 2, \cdots, K\}$, there is a deterministic algorithm computing the distance estimate for the exact SSSP with $\tilde{O}(m \log K)$ work and $\tilde{O}(h \log^2 K)$ span.*

Proof. Combining theorem 3.1, Lemma 5.1 and Lemma 5.2, the lemma holds when $\epsilon \leq \frac{1}{4\beta}$. ∎

To make a general graph a (h, ϵ)-hopset graph, we can use the parallel hopset results from Cao, Fineman, and Russell [6].

THEOREM 5.1. *[6] For any directed graph $G = (V, E)$ with nonnegative integer edge weights from $\{0, 1, 2, ...K\}$, there exists a randomized parallel algorithm that computes a $(n^{1/2+o(1)}, \epsilon)$-hopset of size $\tilde{O}(n \log(nK)/\epsilon^2)$, with high probability. The algorithm takes $\tilde{O}(m \log(nK)/\epsilon^2 + n \log^2(nK)/\epsilon^4)$ work and $n^{1/2+o(1)}/\epsilon$ span, also with high probability.*

To solve the exact SSSP for digraphs with edge weight from $\{0, 1, ..., W\}$, we have to transfer a general graph to a hopset graph, compute the distance estimate, and repeat the whole process $\log W$ times. Fortunately, when we construct the hopset and compute the distance estimate, we can assume $K = O(n)$ and set $\epsilon = \frac{1}{4\log(2nK)}$; this gives us the following theorem.

THEOREM 5.2. *There is a parallel algorithm that given an n-node m-edge directed graph with non-negative integer edge weights from $\{0, 1, 2, \cdots, W\}$ solves the exact single source shortest path problem with $\tilde{O}(m \log W)$ work and $n^{1/2+o(1)} \log W$ span with high probability.*

Proof. The theorem is implied by Lemma 5.3 and Theorem 5.1. ∎

5.2 Implementation in the distributed model. We consider the SSSP problem in the **CONGEST** model[24], which is one of the most studied message-passing models in the field of distributed computing. The CONGEST model is characterized by synchronized communication in a network via non-faulty bounded-bandwidth links. In the CONGEST model, a network is modeled as an undirected n-node graph $N = (V, L)$, where each node of V is modeled as a processor, and each edge $(u, v) \in L$ implies a bidirectional communication link between u and v. Each node of V represents a processor with a unique ID of size $O(\log n)$ and has infinite computational power that initially only knows its adjacent edges in L and their weights. Nodes communicate in synchronous rounds, where in each round, every node may send to each of its neighbors a different message of size $B = \Theta(\log n)$ and subsequently receive the messages sent by its neighbors. In each round, every node can perform unlimited internal computations based on all messages it has received so far. In a round, the messages sent by a node may be different to its various neighbors. The time complexity of an algorithm is measured by the number of rounds. The time complexity is usually expressed in terms of n and D, where $n = |V|$ and D is the diameter of N when the edge weight is omitted.

For SSSP problems in the CONGEST model, the network N is the same as the graph G except that in G the edges are directed, while in N the edges are undirected. To start, each node knows its set of incoming and outgoing edges and their weights, as well as whether it is the source node. Since every node can learn n in $O(D)$ rounds, we assume that all nodes already know n. At the end, each node v must learn its distance $dist(s, v)$, but these distances should not be communicated back to s.

We heavily use the concept of skeleton graph. When we say skeleton graph, we mean a virtual graph $G' = (V', E', w')$ over a subset of nodes $V' \subset V$. There may be no directed link for an edge $e \in E'$, but we require each node to know its incoming and outgoing edges and whether it is in the skeleton graph or not. The following lemmas are standard results for distributed computation in the CONGEST model.

LEMMA 5.4. *[24] Suppose each $v \in V$ holds $k_v \geq 0$ messages of $O(\log n)$ bits each, for a total of $K = \sum_{v \in V} k_v$. Then, all nodes in the network can receive these K messages within $O(K + D)$ rounds.*

LEMMA 5.5. *[17] Consider k distributed algorithms $A_1, ..., A_k$. Let **dilation** be such that each algorithm A_i finishes in dilation rounds if it runs individually. Let **congestion** be such that there are at most congestion messages, each of size $O(\log n)$, sent through each edge (counted over all rounds), when we run all algorithms together. There is a distributed algorithm that can execute $A_1, ..., A_k$ in $\tilde{O}(\textbf{dilation} + \textbf{congestion})$ rounds in the CONGEST model.*

Framework of Forster and Nanongkai [15]. Next, we present an overview of the exact SSSP algorithm, which follows the Forster and Nanongkai [15] framework. The algorithm is parameterized by α, to be set later. Steps 1, 3 and 5 are the same as Forster and Nanongkai's algorithm, and Step 2 is similar. In Step 2, Forster and Nanongkai's algorithm computes distance estimates $\tilde{d}(u, v)$ for each node $v \in S$. Step 2 performs the same computation and additionally computes the distance $\tilde{d}(u, v)$ for each node $u \in S$. The additional distance estimates are used in the computation of Step 4. The main difference is in Step 4. Both algorithms have to solve the exact SSSP on the skeleton graph. Forster and Nanongkai [15] provide two methods, simulating Dijkstra algorithm and recursing the exact SSSP on the skeleton graphs. Chechik and Mukhtar [9] improve the exact SSSP result by using iterative gradual rounding in a different way. We combine the method of distributed hopsets and iterative gradual rounding, which leads to a faster algorithm for exact SSSP on the skeleton graphs.

1. Select each node $v \in V$ to be in the set of skeleton nodes S with probability $\tilde{O}(\alpha/n)$. Add the source s to S. If $|S| > \tilde{\Omega}(\alpha)$, abort the algorithm.

2. Let $g = \tilde{O}(n/\alpha)$. For each skeleton node $u \in S$, compute 1/2-approximate g-hop distances from u, i.e., distance estimation $\tilde{d}(u, v)$ such that $d^{(g)}_G(u, v)/2 \leq \tilde{d}(u, v) \leq d^{(g)}_G(u, v)$. For each $u \in S, v \in S$, both u and v learn $\tilde{d}(u, v)$.

3. Construct the skeleton graph $G_S = (S, E_S, w_s)$, where $E_S = S \times S$, and $w_s(u, v) = \left\lceil \tilde{d}(u, v) \right\rceil$. For nodes $u, v \in S$, both u and v know $w_s(u, v)$.

4. Solve the exact SSSP on the skeleton graph G_S with s as the source, i.e., for each $v \in S$, compute $d_{G_S}(s, v)$.

5. Run the Bellman-Ford algorithm g rounds in the graph G with the starting node set S. Notice that for node $v \in S$, $d_{G_s}(s,v) = d_{G_s \cup G}(s,v)$, and we already compute $d_{G_S}(s,v)$. The key observation is for any node $u \in V$, $d^{(g)}_{G_s \cup G}(s,u) = d_{G_s \cup G}(s,u)$, so g rounds the Bellman-Ford algorithm enough to compute the exact SSSP in the graph $G_s \cup G$.

This framework computes a distance estimate that can be used to solve the exact SSSP. To solve the exact SSSP for graph G, we need to repeat $O(\log W)$ times the above algorithm, but each time we repeat, we can assume that the maximum edge weight in each round is $K = O(n)$. This will save us a $O(\log W)$ factor in the final running time.

In the next subsection, we will show that Step 4 can be implemented in the following rounds.

LEMMA 5.6. *In the CONGEST model, there is a randomized algorithm solving exact SSSP problems on an α node skelenton graphs with non-negative integer edge weights from $\{0,1,2,...,K\}$ in $\tilde{O}((\alpha\rho^2 + \frac{D\alpha^{1/2+o(1)}}{\rho})\log K)$ rounds, where $\rho \in [1, \alpha^{1/2+o(1)})$.*

Cost of the algorithm. Step 1 takes $\tilde{O}(\alpha + D)$ rounds to broadcast S to all nodes. In Step 2, computing the distance estimates can be done in $\tilde{O}(\alpha+g+D)$ rounds. Forster and Nanongkai's algorithm only computes forward distance estimates, but the backward distance estimates can be computed symmetrically. Step 3 computes the skeleton graph and broadcasts it to the graph, which can be done in $\tilde{O}(\alpha + D)$ rounds. Based on Lemma 5.6, it takes $\tilde{O}(\alpha\rho^2 + \frac{D\alpha^{1/2+o(1)}}{\rho})$ rounds to solve the exact SSSP on the skeleton graph. Finally, Step 5 takes $\tilde{O}(D + g)$ rounds to run the Bellman-ford algorithm.

The total number of rounds for the algorithm is $\tilde{O}((D + \frac{n}{\alpha} + \alpha\rho^2 + \frac{D\alpha^{1/2+o(1)}}{\rho})\log W)$, where $\log W$ comes from the fact that we need to repeat the entire algorithm $\log W$ times. We have three cases.

1. If $D = o(n^{1/4})$, set $\rho = \tilde{\Theta}(1)$ and $\alpha = \tilde{\Theta}(\sqrt{n})$. The whole algorithm takes $\tilde{O}(\sqrt{n}\log W)$ rounds.
2. If $D = \Omega(n^{2/3})$, set $\rho = \alpha^{1/2+o(1)}$ and $\alpha = \tilde{O}(n^{1/3})$. The whole algorithm takes $\tilde{O}(D\log W)$ rounds.
3. Otherwise, set $\rho = \tilde{O}(\frac{D^{2/5}}{n^{1/10-o(1)}})$ and $\alpha = \tilde{O}(\frac{n^{3/5+o(1)}}{D^{2/5}})$. The whole algorithm takes $\tilde{O}(D^{2/5}n^{2/5+o(1)}\log W)$ rounds.

Proof. [Proof of Theorem 1.3] Combining all three cases, the algorithm solves the exact SSSP in $\tilde{O}((\sqrt{n} + D + D^{2/5}n^{2/5+o(1)})\log W)$ rounds. ∎

5.2.1 Solving exact SSSP on the skeleton graphs. In this subsection, we show an algorithm to compute the distance estimate for an α-node skeleton graph G. At the end of the algorithm, each node v knows its own $\hat{d}(v)$. Then, based on Lemma 5.4, all nodes in the skeleton graph can learn all $\hat{d}$ in $O(\alpha + D)$ rounds and reweight its incoming and outgoing edges by itself.

Updating d_i without zero-weight edges. At the beginning, each node v knows its own $d_{i+1}(v)$ value. To construct the subgraph, we give each A_j an identifier. There are at most α nodes; we have at most 2α different induced graphs. Each node will broadcast its own $d_{i+1}(v)$ and set up a $O(\log\alpha)$ bit identifier for each induced graph. Now, on each induced graph, in the k-th round, if a node v is not finalized and $d_i(v) \in F(k)$, it will broadcast its $d_i(v)$ and mark itself as finalized. Based on Lemma 5.4, broadcasting all unfinalized nodes in $F(k)$ takes $O(D + x_k)$ rounds and $O(x_k)$ congestion, where x_k is the number of unfinalized nodes in $F(k)$. In total, Algorithm 2 takes $\tilde{O}(Dh\beta + n_j)$ rounds and $O(n_j)$ congestion, where $n_j = |A_j|$ is the number of nodes on the induced graph $G(A_j)$. Using Lemma 5.5, it takes $\tilde{O}(Dh\log K + \alpha)$ rounds to run Algorithm 2 on all induced graphs.

Updating $\hat{d}$ on zero-weight graph. Algorithm 3 is a recursive algorithm. At each level of recursion, we partition the graphs and run BFS with depth h on the induced graph. Note that the recursion depth is at most $O(\log n)$, it takes $O(\log n)$ bits to mark each different induced graph. In total, Algorithm 3 takes $\tilde{O}(Dh + \alpha)$ rounds. This gives us the following lemma.

LEMMA 5.7. *Given an α-node $(h, \epsilon = \frac{1}{4\log(2n\hat{K})})$-hospset skeleton graph G with non-negative integer weights from $\{0,1,2,...\hat{K}\}$, there is a deterministic algorithm computing the distance estimate for exact SSSP in $\tilde{O}(Dh\log^2\hat{K} + \alpha\log\hat{K})$ rounds.*

Proof. We have to run Algorithm 2 $\log \hat{K}$ times and Algorithm 3 once, which gives us $\tilde{O}(Dh \log^2 \hat{K} + \alpha \log \hat{K})$ running time. ∎

Cao, Fineman, and Russell [8] have the following hopsets results for skeleton graphs,

LEMMA 5.8. *[8] Given a n-node skeleton graph G_T over a subset of nodes $T \subset V$ with non-negative integer weight from $\{0, 1, 2, ..., \hat{K}\}$, there is a randomized algorithm that constructs $(h = \alpha^{1/2+o(1)}/\rho, \epsilon)$-hopsets of size $\tilde{O}(\alpha\rho^2 \log \hat{K}/\epsilon^2)$ and takes $\tilde{O}(D\alpha^{1/2+o(1)} \log \hat{K}/(\rho\epsilon) + \alpha\rho^2 \log \hat{K}/\epsilon^2)$ rounds w.h.p. in the CONGEST model, where $\rho \in [1, \alpha^{1/2+o(1)}]$.*

The hopset size is $\tilde{O}(\alpha\rho^2 \log \hat{K}/\epsilon^2)$, we can use Lemma 5.4 to broadcast edges in the hopset and construct a new skeleton graph with the hopset property. In the new skeleton graph, we can compute the distance estimate and the running time is given by Lemma 5.7. Combining Lemma 5.7 and Lemma 5.8 gives us Lemma 5.6.

Proof. [Proof of Lemma 5.6.] Note that the maximum edge weight is K, but when we compute the distance estimate each round, the maximum edge weight is $\hat{K} = O(n)$. The $\log K$ factor comes from the fact that we need to repeat Algorithm 1 $\log K$ times. ∎

Acknowledgement

This research was supported in part by NSF grants CCF-1918989 and CCF-2106759. We would also like to thank Qian Xie, Zhihao Gavin Tang and Hsin-Hao Su for helpful discussions.

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, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 253–264, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342100. doi: 10.1145/2935764.2935768. URL https://doi.org/10.1145/2935764.2935768.

[3] 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. ISBN 978-1-4503-4210-0. doi: 10.1145/2935764.2935765. URL http://doi.acm.org/10.1145/2935764.2935765.

[4] 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. ISSN 0743-7315. doi: 10.1006/jpdc.1998.1425. URL http://dx.doi.org/10.1006/jpdc.1998.1425.

[5] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Efficient construction of directed hopsets and parallel approximate shortest paths. *CoRR*, abs/1912.05506, 2019. URL http://arxiv.org/abs/1912.05506.

[6] Nairen Cao, Jeremy T. Fineman, and Katina Russell. *Efficient Construction of Directed Hopsets and Parallel Approximate Shortest Paths*, page 336–349. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450369794. URL https://doi.org/10.1145/3357713.3384270.

[7] Nairen Cao, Jeremy T. Fineman, and Katina Russell. *Improved Work Span Tradeoff for Single Source Reachability and Approximate Shortest Paths*, page 511–513. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450369350. URL https://doi.org/10.1145/3350755.3400222.

[8] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Brief announcement: An improved distributed approximate single source shortest paths algorithm. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 493–496, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385480. doi: 10.1145/3465084.3467945. URL https://doi.org/10.1145/3465084.3467945.

[9] Shiri Chechik and Doron Mukhtar. Single-source shortest paths in the congest model with improved bound. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 464–473, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375825. doi: 10.1145/3382734.3405729. URL https://doi.org/10.1145/3382734.3405729.

[10] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, January 2000. ISSN 0004-5411. doi: 10.1145/331605.331610. URL http://doi.acm.org/10.1145/331605.331610.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2nd edition, 2001. ISBN 0-262-03293-7, 9780262032933.

[12] 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. ISSN 0001-0782. doi: 10.1145/50087.50096. URL http://doi.acm.org/10.1145/50087.50096.

[13] 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. doi: 10.1109/FOCS.2016.22. URL https://doi.ieeecomputersociety.org/10.1109/FOCS.2016.22.

[14] 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. ISBN 978-1-4503-6184-2. doi: 10.1145/3323165.3323177. URL http://doi.acm.org/10.1145/3323165.3323177.

[15] S. Forster and D. Nanongkai. A faster distributed single-source shortest paths algorithm. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 686–697, 2018. doi: 10.1109/FOCS.2018.00071.

[16] 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. ISSN 0004-5411. doi: 10.1145/28869.28874. URL http://doi.acm.org/10.1145/28869.28874.

[17] Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, page 3–12, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336178. doi: 10.1145/2767386.2767417. URL https://doi.org/10.1145/2767386.2767417.

[18] Mohsen Ghaffari and Jason Li. Improved distributed algorithms for exact shortest paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, page 431–444, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355599. doi: 10.1145/3188745.3188948. URL https://doi.org/10.1145/3188745.3188948.

[19] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algorithms*, 25(2):205–220, November 1997. ISSN 0196-6774. doi: 10.1006/jagm.1997.0888. URL http://dx.doi.org/10.1006/jagm.1997.0888.

[20] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205 – 220, 1997. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1997.0888. URL http://www.sciencedirect.com/science/article/pii/S0196677497908889.

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

[22] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003. ISSN 0196-6774. doi: 10.1016/S0196-6774(03)00076-2. URL http://dx.doi.org/10.1016/S0196-6774(03)00076-2.

[23] 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. ISBN 978-1-4503-3588-1. doi: 10.1145/2755573.2755574. URL http://doi.acm.org/10.1145/2755573.2755574.

[24] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000. ISBN 9780898719772. URL https://books.google.co.il/books?id=T1hFWuDi1CsC.

[25] Václav Rozhoň, Bernhard Haeupler, Anders Martinsson, Christoph Grunau, and Goran Zuzic. Parallel breadth-first search and exact shortest paths and stronger notions for approximate distances. *arXiv preprint arXiv:2210.16351*, 2022.

[26] Václav Rozhoň, Christoph Grunau, Bernhard Haeupler, Goran Zuzic, and Jason Li. Undirected $(1+\varepsilon)$-shortest paths via minor-aggregates: Near-optimal deterministic parallel distributed algorithms, 2022.

[27] Thomas H. Spencer. Time-work tradeoffs for parallel algorithms. *J. ACM*, 44(5):742–778, September 1997. ISSN 0004-5411. doi: 10.1145/265910.265923. URL http://doi.acm.org/10.1145/265910.265923.

[28] Jeffrey D. Ullman and Mihalis Yannakakis. High probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, February 1991. ISSN 0097-5397. doi: 10.1137/0220006. URL http://dx.doi.org/10.1137/0220006.