Single-source Shortest Paths for Directed Graphs

A Dissertation
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in Computer Science

By

Nairen Cao, M.S.

Washington, DC
July 8, 2022
The single-source shortest path (SSSP) problem is as follows: Given a graph with nonnegative edge weights and a designated source vertex $s$, return the true distances from $s$ to each other vertex. The approximate version of SSSP allows estimates of distance such that the estimate falls between the true distance and $(1 + \epsilon)$ times the true distance. This dissertation mainly studies the SSSP problem for directed graphs in parallel and distributed models. This dissertation presents a series of efficient randomized algorithms with theoretical guarantees.

- For directed graphs with non-negative real weight edges, there is a randomized algorithm solving the $(1 + \epsilon)$-approximate SSSP problem with $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span in the parallel work-span model.

- For directed graphs with non-negative real edge weights, there is a randomized algorithm solving the $(1 + \epsilon)$-approximate SSSP problem with $O(n^{2/5+o(1)}D^{2/5} + \sqrt{n} + D)$ rounds in the distributed broadcast CONGEST model.

- For directed graphs with non-negative integer edge weights, there is a randomized algorithm solving the exact distance-limited SSSP problem with $\tilde{O}(m)$ work and $\sqrt{Ln^{1/2+o(1)}}$ in the parallel work-span model, where $L$ is the maximum shortest path length.
• For directed graphs with non-negative integer edge weights, there is a randomized algorithm solving the exact SSSP problem with $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span in the parallel work-span model.

• For directed graphs with non-negative integer edge weights, there is a randomized algorithm solving the exact SSSP problem with $O(n^{2/5+o(1)}D^{2/5} + \sqrt{n} + D)$ rounds in the distributed broadcast CONGEST model.

**INDEX WORDS:** Graph algorithms, Parallel algorithms, Distributed algorithms
First and foremost, I would like to thank my advisor Jeremy T. Fineman. I feel extremely lucky to have been his student for the past 4 years. My previous advisor left Georgetown University at the beginning of my third year, and it is difficult to switch to another research area. Jeremy is a brilliant advisor and helped me adapt to a new area very quickly. He gave me many interesting problems to think about, taught me how to solve a problem, and provided me with many very interesting ideas. I am not good at explaining ideas to others. Thanks to his patience and very helpful advice for writing and presenting papers. He also introduced me to many parallel guys. I cannot imagine where I would be without him.

I also thank my advisor, Ophir Frieder, for making my Ph.D. experience fun. He organized the IR lab and is friends with his students. IR lab students have a very good relationship because of his style. I can always remember his story starting with "Let me tell you a story". I also thank my committee members Justin Thaler and Der-Chen Chang for giving invaluable feedback on my ideas. I would like to express my greatest appreciation to all the individuals who have helped and supported me throughout the project. In alphabetical order they are as follows: Der-Chen Chang, Shi Li, Julián Mestre, Adam O’Neill, Katina Russell, Seeun William Umboh, Eugene Yang, Hao-Ren Yao, and Mohammad Zaheri.

I would like to thank all members of the IR lab and all friends of Georgetown University. We had many memorable times together including having drinks, having seminar lunches, playing Frisbee’s games, and running.
Thank you to my girlfriend Qian Xie for being the absolute best for the last 4 years. Without her, my life will not be so pleasant. Before I knew her, I knew only about 10 restaurants in Washington, DC. She occasionally came to Washington, DC. and took me to explore so many restaurants and attractions. Sometimes I feel like she is the one who lived in Washington, DC. for 6 years. I would also express my love for Georgetown University and Washington DC. In the last year of my Ph.D., I moved to Ithaca with my girlfriend. Ithaca is the smallest town I have ever stayed in, which is even smaller than my hometown. That is the toughest time I have ever had, even more difficult than the time my previous advisor left. At the same time, I realized how lovely Washington, DC, is. I hope to spend more time on the trail, in restaurants, on the rope course, and other fun activities in Washington, DC.
# Table of Contents

## Chapter

1 Introduction .................................................. 1  
1.1 SSSP in Parallel Model ................................. 2  
1.1.1 Related Work ........................................ 2  
1.1.2 Contribution ........................................ 4  
1.2 SSSP in Distributed Model ......................... 6  
1.2.1 Related Work ........................................ 7  
1.2.2 Contribution ........................................ 8  
1.3 Organization ............................................ 8  

2 Preliminaries .................................................. 10  

3 Parallel Approximate Single-Source Shortest Paths ............. 12  
3.1 Introduction ............................................. 12  
3.1.1 Hopsets ............................................. 13  
3.1.2 Overview of Diameter Reduction ................. 15  
3.1.3 Previous Algorithms for Diameter Reduction .... 16  
3.1.4 Overview of the Hopset Algorithm ............. 19  
3.2 Algorithm ................................................. 24  
3.2.1 Parameters .......................................... 25  
3.2.2 Pivots and Shortcutters ............................ 25  
3.2.3 Search Distances .................................... 25  
3.2.4 Explanation of Algorithm 1 and Algorithm 2 ... 26  
3.2.5 Partition Based on Labels ......................... 27  
3.3 Analysis .................................................... 29  
3.3.1 Running Time and Hopset Size .................. 29  
3.3.2 Hopbound ............................................ 37  
3.3.3 Approximation ...................................... 53  
3.3.4 Tradeoff of the Hopsets ......................... 55  
3.4 Weighted Graphs .......................................... 55  
3.4.1 Weighted Hopsets Algorithm .................. 56  
3.4.2 Analysis ............................................. 56  
3.5 Parallel Algorithm ...................................... 58  
3.5.1 Algorithm Description ............................ 61  
3.5.2 Parallel Hopbound and Hopset Size ........... 62
Chapter 1

Introduction

This dissertation studies the single-source shortest paths problem (SSSP) for directed graphs. The shortest paths problem is one of the most fundamental problems in combinatorial optimization. Given a directed graph $G = (V, E, w)$ with nonnegative edge weights, the single source shortest paths problem aims to compute minimum weighted paths from a designated $s \in V$ to all other nodes. In the classical RAM model, Dijkstra's algorithm [10] and the improvements solve SSSP almost optimally. However, in other models, including parallel model and distributed model, the existence of optimal SSSP algorithms for shortest paths is a long-standing open problem and still receives much attention. The effort to obtain better parallel algorithms led to considering the relaxed problem of computing paths that are within $(1 + \epsilon)$ approximate of the shortest path, called the approximate shortest paths problem. This dissertation gives several algorithms for solving the approximate SSSP or exact SSSP problem for directed graphs in parallel and distributed models, and these algorithms outperform all previous algorithms in related models.

Some of the the algorithms discussed depend on the magnitude of the weight. Recall that given a graph $G = (V, E, w)$ with non-negative edge weights, I assume that the minimum non-zero graph edge weight is 1 and the maximum graph edge weight is $W$. I use $|V| = n$ and $|E| = m$ to represent the number of nodes and edges for graph $G$. 
1.1 SSSP in Parallel Model

There are several different models to describe parallel algorithms. In this dissertation, I use the **work-span model**, which abstracts low-level details of the machine, such as the number of processors or how parallel tasks are scheduled.

In the work-span model, algorithms are expressed through the inclusion of parallel loops, i.e. a parallel foreach. A parallel foreach indicates that each task corresponding to a loop iteration may execute in parallel and that all parallel tasks must complete before continuing to the next step after the loop. The cost of the algorithm is measured by work and span, where the **work** of a parallel algorithm is the total number of primitive operations and the **span** is the length of the longest chain of the DAG. The work and span are denoted as $W(n)$ and $S(n)$, respectively. Ideally, the work of the parallel algorithm should be similar to the best sequential running time $T^*(n)$ known for the problem. An algorithm is work efficient if $W(n) = O(T^*(n))$ and nearly work efficient if $W(n) = \tilde{O}(T^*(n)) = O(T^*(n) \cdot \text{poly}(\log n))$, where $\tilde{O}$ hides logarithmic factors.

1.1.1 Related Work

Approximate SSSP has been extensively studied on undirected graphs [8, 13, 15, 26]. Recently, Andoni et al. [1] and Li [25] independently designed undirected approximate SSSP algorithms which both achieve $\tilde{O}(m)$ work and $\tilde{O}(1)$ span and are nearly optimal.

On the other hand, there are essentially no good parallel algorithms known for directed graphs, especially when the graph is sparse. Even for the simplest case of an unweighted, undirected graph, all exact SSSP algorithms to date either have linear span, meaning that they are inherently sequential, or they only manage to reduce
Table 1.1: SSSP algorithms for directed graphs in the parallel model. $m$ and $n$ are the number of edges and nodes, $L$ is the maximum path length for distance-limited search. All algorithms can achieve tradeoff between the work and span and this table mainly compares the span for nearly linear work algorithm. My first result is approximate algorithm, and all other algorithms are exact SSSP.

<table>
<thead>
<tr>
<th>Paper</th>
<th>Work</th>
<th>Span</th>
<th>Edge weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Dijkstra’s[5]</td>
<td>$\tilde{O}(m)$</td>
<td>$O(n)$</td>
<td>non-negative real</td>
</tr>
<tr>
<td>Klein and Subramanian[24]</td>
<td>$\tilde{O}(m\sqrt{n})$</td>
<td>$O(\sqrt{n})$</td>
<td>non-negative integer</td>
</tr>
<tr>
<td>Spencer’s algorithm [29]</td>
<td>$\tilde{O}(m)$</td>
<td>$O(n\sqrt{\frac{m}{n}})$</td>
<td>non-negative real</td>
</tr>
<tr>
<td>My result</td>
<td>$\tilde{O}(m)$</td>
<td>$n^{1/2+o(1)}$</td>
<td>non-negative real</td>
</tr>
<tr>
<td>My result</td>
<td>$\tilde{O}(m)$</td>
<td>$\sqrt{L} \cdot n^{1/2+o(1)}$</td>
<td>non-negative integer</td>
</tr>
<tr>
<td>My result</td>
<td>$\tilde{O}(m)$</td>
<td>$n^{1/2+o(1)}$</td>
<td>non-negative integer</td>
</tr>
</tbody>
</table>

the span at the cost of significantly increasing the work. Klein and Subramanian [24] designed an algorithm for exact SSSP achieving $\tilde{O}(m\sqrt{n})$ work and $\tilde{O}(n)$ span. Spencer’s algorithm [29] solves the exact SSSP problem with $\tilde{O}(m + n\rho^2)$ work and $\tilde{O}(n/\rho)$ span, where $\rho$ is a parameter between 1 and $n$. Later, Forster and Nanongkai [17] designed an algorithm which extends Klein and Subramanian’s algorithm [24] and has work $\tilde{O}((\rho^3 + mn/\rho + m\rho) \log W)$ and span $\tilde{O}(n/\rho)$, where $\rho$ is a parameter between 1 and $n$. Note that for all approximate SSSP algorithms in parallel model, when the work is nearly linear, namely $\tilde{O}(m)$, the span is $\tilde{O}(n)$. All parallel SSSP algorithms for directed graphs are summarized in Table 1.1.
1.1.2 Contribution

This dissertation devises better approximate and exact SSSP algorithms for directed graphs. More specifically, Chapter 3 shows a nearly efficient approximate SSSP algorithm.

**Theorem 1.1.1.** There exists a parallel algorithm that takes as input a directed graph $G$ with non-negative real edge weights and computes $(1 + \epsilon)$-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.

This is the first parallel approximate SSSP algorithm with nearly linear work and sublinear span for directed graphs with nonnegative real edge weights. The main technical contribution is the first nearly linear work algorithm for constructing hopsets on directed graphs. A $(\beta, \epsilon)$-hopset is a set of weighted edges (sometimes called shortcuts) that, when added to the graph, admit $\beta$-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 $\tilde{O}(m\sqrt{n})$. This algorithm is the first more efficient algorithm that produces a directed hopset with similar characteristics. Specifically, I show that the hopsets can be constructed with following running time,

**Theorem 1.1.2.** For any weighted directed graph $G = (V, E)$, there exists a randomized parallel algorithm for weighted graphs that computes a $(n^{1/2+o(1)}/\rho, \epsilon)$-hopset of size $O(n\rho^2 \log^{22} n \log(nW)/\epsilon^2)$. The algorithm has $O(m\rho^2 \log^{22} n \log(nW)/\epsilon^2 + n\rho^4 \log^{44} n \log^2(nW)/\epsilon^4)$ work and $n^{1/2+o(1)}/(\epsilon\rho)$ span with high probability, where $\rho \in [1, \sqrt{n}]$.

Although hopsets only guarantee that some bounded-hop path is an approximate shortest path, it turns out that the hopset is very useful for the exact SSSP problem.
For the exact SSSP problem, even in undirected graph, it remains open whether one can give a parallel algorithm with nearly linear work and sublinear span.

First, I consider a natural relaxation of the exact SSSP, which is distance-limited SSSP. The goal is to return the correct shortest path distance to all vertices having the shortest path distance \( \leq L \) from the source, where \( L \) is part of the input to the problem. In addition, the algorithm should also identify which vertices have the shortest path distance strictly more than \( L \) from the source. The only interesting distance-limited variant is for unweighted graphs, where the problem can be solved by parallel breadth-first search (BFS). In particular, it is straightforward to parallelize breadth-first search to achieve \( \tilde{O}(m) \) work and \( O(L \log n) \) span and to correctly output the exact distances to all vertices with distance at most \( L \) from the source. For this variant of the problem, the edge weights are all nonnegative integers, but the problem is still nontrivial even if all weights are from \{0, 1\}. In Chapter 5, I show that

**Theorem 1.1.3.** There exists a parallel algorithm solving nonnegative \( L \) distance-limited SSSP with work \( \tilde{O}(m) \) and span \( n^{1/2+o(1)}L^{1/2} \) span, with high probability. That is, consider a directed graph \( G = (V, E) \) with nonnegative integer weights and source \( s \in V \). The algorithm outputs for each node \( v \in V \), \( d(v) = \text{dist}(s, v) \) if \( \text{dist}(s, v) \leq L \), and infinity otherwise.

Recently, I also start the work of exact SSSP problem. In Chapter 6, I show that the parallel exact SSSP problem can be efficiently solved,

**Theorem 1.1.4.** There exists a parallel algorithm that takes as input a graph \( G \) with nonnegative integer edge weights and computes exact 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.
1.2 SSSP in Distributed Model

I consider the SSSP problem in CONGEST model [27], 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 \( O(\log n) \) bits ID and has infinite computational power that initially only knows its adjacent edges in \( L \) and their weights. The nodes communicate in synchronous rounds, wherein each round, every node may send to each of its neighbors a 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. For SSSP in CONGEST, each node \( v \) must learn its distance estimate \( \tilde{d}(s, v) \), but these distances need not be communicated back to \( s \). 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, I assume that all nodes already know \( n \). This thesis also considers a stronger distributed model, broadcast CONGEST model. The only difference is that in each round, each node can send only
the same message to all its neighbors, while in the CONGEST model, the messages can be different.

1.2.1 Related Work

Peleg and Rubinovich [28] showed that $\tilde{\Omega}(\sqrt{n} + D)$ is required for the approximate SSSP in the CONGEST model, even for undirected graphs. The Bellman-Ford algorithm for SSSP [11] can be used in CONGEST model, and runs in $O(n)$ rounds. This result was the fastest known algorithm for a long time until Elkin [14] provided a randomized algorithm that runs in $\tilde{O}(n^{5/6})$ rounds for $D = \tilde{O}(\sqrt{n})$ and $\tilde{O}(D^{1/3}n^{2/3})$ rounds for larger $D$.

For undirected graphs, the approximate version of the problem has been well studied. The state of the art is a deterministic algorithm from Becker et al. [2] which computes $(1 + \epsilon)$-approximate shortest paths in $\tilde{O}(\sqrt{n} + D)$ rounds.

For directed graphs in the CONGEST model, recent progress has been made on the exact version of the problem. Ghaffari and Li [19] presented two randomized algorithms for 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 Nonongkai [17] 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 [6] recently showed a randomized algorithm that achieves $\tilde{O}(\sqrt{nD^{1/4}\log^2 W + D})$ rounds. For the approximate shortest paths, Forster and Nonongkai [17] give a randomized algorithm that runs in $\tilde{O}((n^{1/2}D^{1/4} + D)\log W/\epsilon)$ rounds.
1.2.2 Contribution

In this thesis, the main result is faster algorithms for approximate SSSP and exact SSSP in the CONGEST and broad CONGEST models.

In Chapter 4, I show how to apply the result of parallel hopsets to the distributed model and solve the approximate SSSP problem,

**Theorem 1.2.1.** In the CONGEST and the broadcast CONGEST model, there exists a randomized algorithm that solves \((1 + \epsilon)-approximate single-source shortest-paths\) problem for directed \(n\)-node graph \(G\) with nonnegative real edge weights, in \(\tilde{O}((\sqrt{n} + D + D^{2/5}n^{2/5+o(1)}) \log W/\epsilon^2)\) rounds, with high probability, where \(D\) is the undirected diameter of \(G\).

Then in Chapter 6, I show how to construct an algorithm for the exact SSSP,

**Theorem 1.2.2.** There is a distributed randomized algorithm that, given an \(m\)-edge \(n\)-node directed graph with nonnegative 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 and broadcast CONGEST model with high probability, where \(D\) is the hop diameter of the undirected communication network.

1.3 Organization

The remainder of the thesis is organized as follows. Chapter 2 introduces some basic notation and formulas. The next four chapters present my (approximate) SSSP algorithms in work-span and (broadcast) CONGEST models. Specifically, Chapter 3 shows how to construct hopsets efficiently and use it solve the approximate SSSP problems in the work-span model. Chapter 4 gives an algorithm for the approximate SSSP problem in the CONGEST and broadcast CONGEST model. Next, I first
give a parallel exact SSSP algorithm for distance limited shortest paths problem in
Chapter 5 and show how to solve the exact SSSP problem in Chapter 6.
For any vertices $u, v \in V$, define $\text{dist}^{(\beta)}_{G}(u, v)$ to be the minimum weight of a path from $u$ to $v$ containing at most $\beta$ edges. If there is no path containing at most $\beta$ edges from $u$ to $v$, then $\text{dist}^{(\beta)}_{G}(u, v) = +\infty$. I also refer to $\text{dist}_{G}(u, v)$ as the shortest path distance from $u$ to $v$. For a set of edges $E$ and a constant $c$, I 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 $w_{E'}(e)$, that is, $w_{E \cup E'}(e) = \min(w_{E}(e), w_{E'}(e))$.

I assume that the lightest nonzero 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 $i \in [0, \ell - 1]$. I denote the length of the path $P$ as $|P|$ and $|P| = \ell$ is the number of edges on $P$. I also call $|P|$ the number of hops of $P$. The first and last vertex of the path are $\text{head}(P) = v_{0}$ and $\text{tail}(P) = v_{\ell}$.

For a vertex $v$, I say $v \in P$ if $v = v_{i}$ for some $i \in [0, \ell]$. I consider the weight of the 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 $\text{head}(P) = \text{head}(P')$, $\text{tail}(P) = \text{tail}(P')$, and $w(P) \leq w(P') \leq (1 + \epsilon)w(P)$. 

Paths.
Hopsets. A $(\beta, \epsilon)$-hopset for directed graph $G = (V, E)$ is a set of weighted edges $H$, such that for any vertices $u$ and $v$ in $V$, $\text{dist}_G(u, v) \leq \text{dist}^{(\beta)}_G(u, v) \leq (1 + \epsilon)\text{dist}_G(u, v)$, where $G' = (V, E \cup H)$. $\beta$ 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 $\text{dist}_G(u, v) \leq d$. I 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 by $R^+_d(G, v) = \{u|u \preceq_d v\}$ 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. Denote by the set $R_d(G, v) = R^+_d(G, v) \cup R^-_d(G, v)$ be $v$'s related nodes within $d$-distance. If $d = n$, I will ignore $d$. Similarly, I 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 $x \in R^+_d(G, P) \setminus R^-_d(G, P)$. Vertex $x$ a $d$-ancestor of $P$ if and only if $x \in R^-_d(G, P) \setminus R^+_d(G, P)$. $x$ 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 thesis, 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,
\[
\Pr[X \geq (1 + \delta)np] \leq \exp\left(-\frac{\delta^2}{2 + \delta}np\right).
\]
If $X \sim B(n, p)$, then
\[
E\left[\frac{1}{X + 1}\right] \leq \frac{1}{E[X]}.
\]
3.1 Introduction

This section presents the first efficient algorithm for producing a hopset on directed graphs with sublinear hopbound. Specifically, my 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 (Section 3.2–Section 3.3), 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 3.4), 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.

My parallel algorithm (Section 3.5) 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 my parallel hopset construction then applying Klein and Subramanian’s algorithm [23] 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, my algorithm has work \(\tilde{O}(m \log(nW)/\epsilon^4)\) and span \(O(n^{1/2 + O(1/\log \log n)}/\epsilon)\).
3.1.1 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 few hops. Specifically, Klein and Subramanian’s weighted breadth-first search algorithm [23] gives a \((1 + \epsilon)\) approximation of \(\beta\)-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 \(\beta\)-hop distances are not too much larger than the actual unbounded distances; the preprocessing step amounts to finding a good hopset.

A \((\beta, \epsilon)\) 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 \(\beta\) hops, where \(\beta\) 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 \((\beta, \epsilon)\) 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., \(w(u, v) \geq \text{dist}_G(u, v)\), and (2) for every \(u, v \in V\) there exists a path \(p\) from \(u\) to \(v\) in \(G'\) comprising at most \(\beta\) hops such that \(w(p) \leq (1 + \epsilon)\text{dist}_G(u, v)\). (The first constraint implies that \(w(p) \geq \text{dist}_G(u, v)\).)

Although hopsets were first formalized by Cohen [8], 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 \(\beta\)-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 \(\beta\), the approximation quality \(\epsilon\), and the complexity of an algorithm for constructing the hopset. When \(\epsilon = 0\), the hopset produced
is an *exact hopset*, meaning that the $\beta$-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 $w(s_i, s_j) = \text{dist}(s_i, s_j)$. 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 $\beta$ hops nearest to $u$ and $\beta$ hops nearest to $v$ each contain at least one sampled vertex, so the rest of the path can be bypassed using a hopset edge. Ullman and Yannakakis [30] and Klein and Subramanian [23] 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 hop-bound, 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 I am willing to accept a $(1 + \epsilon)$ approximation, there exist linear-size hopsets with much smaller (subpolynomial) hopbound [15]. Moreover, there are more efficient algorithms [8, 13, 15, 26] 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 $\epsilon$ 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 [20], 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 $\epsilon$ before the current paper.

3.1.2 Overview of Diameter Reduction

The algorithm and analysis builds on recent breakthroughs on the diameter-reduction problem by Fineman [16] later improved by Jambulapati, Liu, and Sidford [21], 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 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 [16] 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 [21] 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.

My algorithm for hopsets most closely resembles the JLS algorithm for diameter reduction.
3.1.3 Previous Algorithms for Diameter Reduction

Both Fineman’s algorithm [16] and the JLS algorithm [21] 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 [16] selects a single pivot uniformly at random. JLS [21] 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 I shall assume as much going forward to simplify the statement of remaining bounds. Each vertex is 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. I 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 $P$. I write $v \leq P$ if it is possible to get from $v$ to some vertex on $P$ by following directed edges and $P \leq v$ if it is possible to get from some vertex on $P$ to $v$ by following directed edges. A vertex $v$ is an \textit{ancestor} of $P$ if $v \leq P$ and $P \not\leq v$. The vertex is a \textit{descendant} of $P$ if $v \not\leq P$ and $P \leq v$. It is a \textit{bridge} if $v \leq P$ and $P \not\leq v$. The vertex is \textit{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 \textit{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 [16], 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 [21] 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 [16] 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. I 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 [21] 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 $r$th 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 [21] adopts a bottom-up approach, solving a recurrence on
the shortcutted path length for a given number of ancestors/descendants.

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 \( h = \tilde{\Theta}(n^{2/3}) \) and \( h = n^{1/2+o(1)} \), respectively. As noted
previously, there are parallel algorithms implementing \( h \)-hop limited searches with \( O(h) \) span [23]. 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 [16] and JLS [21] (1)
only analyzes paths with length \( \tilde{\Theta}(h) \), and (2) handle vertices near the boundary of
the search, called \textit{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.

3.1.4 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. My 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. Specifi-
cally, 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 \( \text{dist}(s, x) + \text{dist}(x, t) \gg \text{dist}(s, t) \),
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 [16, 21], 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.

**Algorithm for hopsets**

My 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. I 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, I split the roles; I 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 I 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 $s$, I compute the single-source shortest paths from $s$ to all other vertices in both the forwards (and backwards) directions. I then add the weighted edges $(s, v)$ (and $(v, s)$) with weight $w(s, v) =$
\( \text{dist}(s, v) \) (and \( w(v, s) = \text{dist}(v, s) \)) 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, I 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 \( \lambda \). The initial distance is important—the algorithm only well-approximates paths if the initial search distance is similar to the shortest-path distance—so I 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 [16, 21] parallel algorithms for diameter reduction. (My 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 my 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**

The 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 [16, 21]. 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, I 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 $r = \log_{\sqrt{k}} n$ 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 I 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. I 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 $\lambda$, this term dominates the number of subproblems arising from the previous level of recursion, so I have a total of $O(\lambda^r k^{r/2})$ subproblems at recursion depth $r$. Again, this bound

---

1 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.
readily implies 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 $r$, I say that a bridge $x$ is nearby if $\text{dist}(s, x) + \text{dist}(x, t) = \text{dist}(s, t) + O((\epsilon/\log n)w(P)/(\lambda^r k^r/2))$. Since the total number of subproblems is $O(\lambda^r k^r)$, 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 I can achieve it by leveraging both the bounded search distance as well as the oversampling of shortcutters. In fact, for $\epsilon = \Omega(\log n)$, I can immediately see that all bridges are nearby—the additive error is bounded by twice the maximum search distance, i.e., $O(w(P)/(\lambda^r k^r/2)) = O((\epsilon/\log n)w(P)/(\lambda^r k^r/2))$.

I thus achieve a hopset with $\epsilon = O(\log n)$ even setting the shortcutters and pivots to be identical.

To achieve a better approximation, I 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, I consider the recursion tree of relevant subproblems, but I 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. I 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.

### 3.2 Algorithm

In this section, I describe the hopset algorithm \( \text{HOPSET}(G) \). The algorithm takes as input graph \( G = (V, E) \), and has parameters \( k, \lambda \) 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, I will describe some components of the algorithm and then describe the details of the algorithm.
3.2.1 Parameters

The algorithms Hopset and HSRecurse have parameters $k$, $\lambda$ 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 $\lambda$, which is a constant and controls the probability the algorithm succeeds. The algorithm requires that $k \geq 2$, and $\lambda \geq 8$.

3.2.2 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, 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.

3.2.3 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 $\rho_v$ 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 $4\lambda^2k\log^2 n$ disjoint subintervals, each with length $4k$. A subinterval is chosen uniformly at random, which is represented by $\sigma_v$ in the algorithm. Finally, the scalar $\rho_v$ is chosen from within the subinterval to minimize the number of fringe vertices when using search distance $\rho_v D_r$. I use these search distances to guarantee that there are not too many fringe vertices.

3.2.4 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 $2^j C_k^{-c}$ 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$(G, D = 2^j C_k^{-c}, r = 0)$ on the whole graph $G$ with $D$ set to $2^j C_k^{-c}$ 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 $\sigma_v$ uniformly at random from $[1, 4\lambda k \log^2 n]$. Next, search from $v$ to distance $16\lambda^2k^2 \log n + 4k\sigma_v$ and find the distance $\rho_v$ that has the minimal number of vertices exactly $\rho_v$ distance away, where $\rho_v$ is restricted to $[16\lambda^2k^2 \log^2 n + 4k(\sigma_v - 1), 16\lambda^2k^2 \log^2 n + 4k\sigma_v]$. 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 $V_v^{\text{fringe}}$ for vertex $v$ as $R_{(\rho_v+1)D_r}(G, v) \setminus R_{(\rho_v-1)D_r}(G, v)$, and recurse on the induced subgraph $G[V_v^{\text{fringe}}]$.

Next for each shortcutter, i.e. each vertex $v$ where $\ell(v) \leq r + L$, search forwards and backwards from $v$ for distance $32\lambda^2k^2D_r \log^2 n$ 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.

### 3.2.5 Partition Based on Labels

Line 20 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. I I

**Algorithm 1** Hopset algorithm for unweighted directed graphs. $k, \lambda$ and $L$ are parameters.

1: function Hopset($G = (V, E)$)
2: $H \leftarrow \emptyset$
3: repeat $\lambda \log n$ times
4: for each $j \in [\log n/2, \log n]$
5: for each $v \in V$
6: for each $i \in [0, \log_k n]$
7: With probability $(\lambda k^{i+1} \log n)/n$, set $\ell(v)$ to $i$,
8: if setting successful.
9: if $\ell(v) \leq L$ then
10: for each $u \in R_{2j+1}^+(G, v)$ add edge $(v, u)$ to $H$ with weight $dist_G(v, u)$
11: for each $u \in R_{2j+1}^-(G, v)$ add edge $(u, v)$ to $H$ with weight $dist_G(u, v)$
12: $H \leftarrow H \cup \text{HSRecurse}(G, D = 2^j k^{-c}, r = 0)$
13: return $H$
Algorithm 2 Recursive subroutine for Hopset Algorithm. $k$, $\lambda$ and $L$ are parameters.

1: function HSRecurse($G, D, r$)
2: $D_r \leftarrow D/(\lambda^r k^{r/2})$, $H \leftarrow \emptyset$
3: for each $v \in V$ with $\ell(v) = r$
4:    choose $\sigma_v$ uniformly at random from $[1, 4\lambda^2 k \log^2 n]$
5:    minimize $|R(\rho_{v+1}D_r(G, v) \setminus R(\rho_{v-1}D_r(G, v))|$ such that $\rho_v \in [16\lambda^2 k^2 \log^2 n + 4k(\sigma_v - 1), 16\lambda^2 k^2 \log^2 n + 4k\sigma_v]$
6:    for each $u \in R_{\rho_v D_r}^+(G, v)$ add label $v^{Des}$ to vertex $u$
7:    for each $u \in R_{\rho_v D_r}^-(G, v)$ add label $v^{Anc}$ to vertex $u$
8:    for each $u \in R_{\rho_v D_r}^+(G, v) \cap R_{\rho_v D_r}^-(G, v)$ add label $X$ to vertex $u$
9: $V^\text{fringe}_v \leftarrow R(\rho_{v+1}D_r(G, v) \setminus R(\rho_{v-1}D_r(G, v))$
10: $H \leftarrow H \cup \text{HSRecurse}(G[V^\text{fringe}_v], D, r + 1)$
11: for each $v \in V$ with $\ell(v) = r + L$
12:    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 $\text{dist}_G(v, u)$
13:    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 $\text{dist}_G(u, v)$
14: for each $u \in V$ that has a $X$ label, remove $u$
15: $V_1, V_2, \ldots, V_t \leftarrow$ partition based on labels
16: for each $i \in [1, t]$
17:    $H \leftarrow H \cup \text{HSRecurse}(G[V_i], D, r + 1)$
18: return $H$
3.3 Analysis

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

**Theorem 3.3.1.** 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.

I start by proving the runtime and the size of the hopset in Section 3.3.1. Then I show the hopbound in Section 3.3.2, and finally the approximation in Section 3.3.3.

### 3.3.1 Running Time and Hopset Size

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

**Theorem 3.3.2.** One execution of \( \text{Hopset}(G = (V,E)) \) 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 3.3.2 follows the same structure as the runtime proof from JLS [21]. First, I bound the related vertices in each recursive subproblem in Lemma 4.4.3. Then I show the number of times a vertex is added to the fringe problem is small in Lemma 3.3.4. Since only fringe vertices are duplicated, I can bound the total number of vertices and edges in all recursive subproblems in Lemma 3.3.5. 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 [21] 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 [21].

I 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 [21]. My algorithm differs from JLS [21] for the fringe problem because I increase $r$ as I 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 3.3.2.

**Lemma 3.3.3.** Consider an execution of $\text{HSRecurse}(G', d, 0)$ on $n$-node $m$-edge graph $G$. With probability at least $1 - n^{-0.7\lambda+3}$ in each recursive call of $\text{HSRecurse}(G', D, r)$ the following holds for all $v \in G'$,

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

**Proof.** Proof by induction on $r$. I will show $R^-_{\rho_{\max}D_r}(G', v) \leq nk^{-r}$, and $R^+_{\rho_{\max}D_r}(G', v) \leq nk^{-r}$ follows by a symmetric argument. For $r = 0$, it is clear that the number of related ancestors is at most $n$.

For $r > 0$, there are two types of recursive subproblems. Specifically, there are core problems called by $\text{HSRecurse}(G[V_i], D, r+1)$ and fringe problems called by $\text{HSRecurse}(G[V_{vfringe}], D, r+1)$. I will start with the related ancestors in the fringe problems, and then show the core problems case.

Each fringe problem is called by $\text{HSRecurse}(G[V_{vfringe}], D, r+1)$. Let vertex $u$ be the pivot that calls this fringe problem. By the inductive hypothesis, $R^-_{\rho_{\max}D_r}(G', u) \leq 2nk^{-r}$. The pivot $u$ chooses $\rho$ in $[16\lambda^2k^2 \log^2 n + 4k(\sigma_u - 1), 16\lambda^2k^2 \log^2 n + 4k\sigma_u]$, such that the set of vertices $|R^+_{\rho+1D_r}(G, v)\backslash R^-_{\rho-1D_r}(G, v)|$ is minimized. The size of the range is $4k$, and the search distance for that fringe problem is at most $\rho_{\max}$. By taking the minimum in the range of $4k$, I can guarantee that the total size of the subproblem is reduced by $2/(4k)$, where the $2$ comes from $(\rho - 1)D_r$ to $(\rho + 1)D_r$. 
Therefore, the size of the problem is at most $nk^{-r-1}$, and each vertex has at most $nk^{-r-1}$ related ancestor or descendant nodes.

Now I will prove the claim for the core problems. Let $A = R_{\rho_{\text{min}} D_r}(G', v)$ be the set of ancestors of $v$ directly after a call of $\text{HSRecurse}(G[V_i], D, r+1)$. If $|A| < nk^{-r-1}$, then I am done because $\rho_{\text{max}} D_{r+1} < \rho_{\text{min}} D_r$. Otherwise assume $|A| \geq nk^{-r-1}$. If there are no cycles in $G'$, then the proof is straightforward. Order $A$ such that for any $u, y \in A$, if $u \preceq y$ then $u$ precedes $y$. Call the vertices of $A$ in this order $\langle w_1, w_2, \ldots, w_{|A|} \rangle$. If $w_i$ is a pivot, then any vertex $w_j$, with $j \leq i$ is not in the same subproblem as $v$. This is because any $w_j$ gets either $w_i^{\text{Anc}}$ or no label from $w_i$, whereas $v$ gets $w_i^{\text{Des}}$ label. Therefore, if some $w_i$ is a pivot where $i \geq |A| - nk^{-r-1}$, then the number of ancestors of $v$ is no greater than $nk^{-r-1}$. In level $r$, the probability of being a pivot is $\lambda k^{r+1} \log n/n$. The probability that no $w_i$ with $i \geq |A| - nk^{-r-1}$ is a pivot is $(1 - \frac{\lambda k^{r+1} \log n}{n})^{nk^{-r-1}} \leq e^{-\lambda \log n} \leq n^{-1.4\lambda}$.

If $G'$ contains cycles, then the proof becomes more complicated. Consider a topologically sorted order of the strongly connected components on $A = \langle A_1, A_2, \ldots, A_i, \ldots \rangle$. The difficulty arises within $A_i$ because there is no order between strongly connected vertices. It is possible for any order of $A_i = \langle u_1, u_2, \ldots \rangle$, there exists $j$ such that if $u_j$ is a pivot at level $r$, $u_j$ gets $u_j^{\text{Des}}$ label and $j' < j$. In this case, the argument from the acyclic case no longer holds. I would like to identify how many nodes in $A_i$ could be in $R_{\rho_{\text{min}} D_r}(G', v)$.

I will show later that for any strongly connected components $A_i$, there will be at most $n^{-k-1}/2$ nodes in $R_{\rho_{\text{max}} D_r}(G', v)$ with probability $n^{-0.7\lambda+1}$. Now consider the index $j$, such that $|\cup_{i \geq j} A_i| \leq nk^{-r-1}/2$ and $|\cup_{i \geq j} A_i| > nk^{-r-1}/2$. If $|\cup_{i \geq j} A_j| > nk^{-r-1}$, then $R_{\rho_{\text{min}} D_r}(G', v) \leq nk^{-r-1}$ with probability $1 - n^{-0.7\lambda}$. This is the same as acyclic argument except using $n^{-k-1}/2$ instead of $n^{-k-1}$. Otherwise $|\cup_{i \geq j} A_j| >
nk^{-r-1}, in which case |A_j| \geq n^{-k-1}/2. Based on the strongly connected components argument, A_j will have at most n^{-k-1}/2 nodes in R_{\rho_{\text{min}}D_r}(G', v) and all A_{j'}, where j' < j, will be not in R_{\rho_{\text{min}}D_r}(G', v). Therefore, R_{\rho_{\text{min}}D_r}(G', v) \leq nk^{-r-1} with probability 1 - 2n^{-0.7\lambda+1}. By taking a union bound over all v \in V and all r, the lemma holds for the core problems with probability 1 - n^{-0.7\lambda+3}.

The last part of the proof is to show that for any strongly connected component A_i, there will be at most n^{-k-1}/2 nodes in R_{\rho_{\text{max}}D_r}(G', v) with probability n^{-0.7\lambda+1}. For any strongly connected component A_i and two nodes u_j, u_{j'} \in A_i, define the relation \mathcal{R} as follows. If dist_{G'}(u_j, u_{j'}) \geq dist_{G'}(u_{j'}, u_j), then \mathcal{R}(u_j, u_{j'}) = 1, and if dist_{G'}(u_j, u_{j'}) < dist_{G'}(u_{j'}, u_j), then \mathcal{R}(u_j, u_{j'}) = 0. Note that \mathcal{R}(u_j, u_{j'}) = 0 implies \mathcal{R}(u_{j'}, u_j) = 1. If \mathcal{R}(u_{j'}, u_j) = 1 and u_{j'} is chosen as a pivot, then u_j gets u_{j'}^{\text{Des}} label only if u_j gets u_{j'}^{\text{Anc}} label. I know that if u_{j'} is chosen as a pivot, v gets only u_{j'}^{\text{Des}} label. Therefore, if \mathcal{R}(u_{j'}, u_j) = 1 and u_{j'} is chosen as a pivot, then u_j is not in v’s subproblem. Consider the set for node u_j \in A_i, T(u_j) = \{u_{j'} \in A_i \mid \mathcal{R}(u_{j'}, u_j) = 1\}. In other words, T(u_j) is the set of nodes where if u_{j'} \in T(u_j) is a pivot means then u_j is not in the same subproblem as v. If |T(u_j)| > nk^{-r-1}/2, then u_j will be in R_{\rho_{\text{min}}D_r}(G', v) with probability,

\[(1 - \frac{\lambda k^{r+1} \log n}{n})^{nk^{-r-1}/2} \leq n^{-0.7\lambda}.
\]

Define a set S = \{u_j \mid |T(u_j)| \leq nk^{-r-1}/2\}. Based on the above analysis, for a node u \in A_i, if u \notin S, then with probability 1 - n^{-0.7\lambda}, u \notin R_{\rho_{\text{min}}D_r}(G', v). If u_j \in S is a pivot, then there will be at most nk^{-r-1}/2 nodes u_{j'} such that \mathcal{R}(u_{j'}, u_j) = 1, which implies there are at most nk^{-r-1}/2 nodes where \mathcal{R}(u_j, u_{j'}) = 0. Therefore, if u_j is a pivot, there will be at most nk^{-r-1}/2 nodes getting u_j^{\text{Des}} label and all other nodes in A_i will get X or no label from u_j.
Now I divide the size of $S$ into two cases. If $|S| \leq nk^{-r-1}/2$, then at most $nk^{-r-1}/2$ vertices in $A_i$ will be in $R_{\rho_{min},D_r}(G', v)$. On the other hand, if $|S| \geq nk^{-r-1}/2$, a node in $S$ will be a pivot with probability

$$1 - (1 - \frac{\lambda k r + 1 \log n}{n})^{nk^{-r-1}/2} \geq 1 - n^{-0.7\lambda}.$$ 

Once there is a pivot in $S$, there will be at most $nk^{-r-1}/2$ nodes in $R_{\rho_{min},D_r}(G', v)$. In both cases, there will be at most $nk^{-r-1}/2$ nodes of $A_i$ in $R_{\rho_{min},D_r}(G', v)$ with probability $1 - n^{-0.7\lambda+1}$. \hfill $\square$

Next, I consider the expected number of nodes added to fringe problems. JLS [21] has a similar lemma, where they consider the expected number of times a vertex is added to a fringe problem. Since I choose a search distance to minimize fringe vertices, I cannot get the same expectation. Instead, I 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 $\rho_v$ that is chosen from an interval to minimize the number of vertices in the fringe problem. The interval that $\rho_v$ 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, I can guarantee that the number of vertices added to each fringe problem is small.

**Lemma 3.3.4.** 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. $|R_{(\rho+1)D_r}(G', v) \setminus R_{(\rho-1)D_r}(G', v)|$ is $1/(4\lambda k \log n)$.

**Proof.** If $v$ is not a pivot, I define $v$’s fringe problem size as 0. By Lemma 4.4.3, the number of related nodes to $v$ is $2nk^{-r}$. The scaling factor for the search distance, $\rho$, is chosen from an interval of size $8k$ to select the distance with a minimal number of
nodes that are between \((\rho - 1)D_r\) and \((\rho + 1)D_r\) distance away. The \(4k\) size interval is chosen uniformly at random from a larger interval of size \(4\lambda^2 k \log^2 n\). For a pivot \(v\), there will be at most \(2nk^{-r} \cdot 1/(4\lambda^2 k \log^2 n) \cdot 2/(4k)\) nodes in its fringe problem in expectation. Multiplying the probability of \(v\) being a pivot at level \(r\), the expected number of vertices in \(v\)'s fringe problem is \(1/(4\lambda k \log n)\).

Now that the number of vertices in fringe problems is bounded, I can bound the total number of vertices in all recursive subproblems. Lemma 3.3.5 is based on Lemma 5.3 and Corollary 5.5 from JLS [21]. 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 I 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 3.3.5.** Consider one execution of \(\text{Hopset}(G = (V, E))\) where \(n = |V|\) and \(m = |E|\). The expected number of vertices in all recursive executions of \(\text{HSRecurse}(G', D, r)\) is \(2n \log n\). The expected number of edges in all recursive executions of \(\text{HSRecurse}(G', D, r)\) is \(2m \log n\).

**Proof.** In one execution of \(\text{HSRecurse}(G' = (V', E'), D, r)\), the number of vertices called in recursive subproblems is the number called in the fringe problem, \(\text{HSRecurse}(G[V_u^{\text{fringe}}], D, r + 1)\), and the number of vertices called in \(\text{HSRecurse}(G[V_i], D, r + 1)\) for \(i \in [1, t]\). By Lemma 3.3.4, 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, I 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 3.3.6.** Consider a call to \( \text{Hopset}(G) \) and all recursive calls of \( \text{HSRecurse}(G', D, r) \). 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 \( v \in R(G', (\rho_u + 1)D_r, u) \), I will slightly overcount the pivots, by extending \( \rho_u + 1 \) to \( \rho_{\text{max}} \). This will only increase the pivots I am counting. Observe that all pivots \( u \) such that \( v \in R(G', \rho_{\text{max}} D_r, u) \) are in \( R(G', \rho_{\text{max}} D_r, v) \). By Lemma 4.4.3, \( |R(G', \rho_{\text{max}} D_r, v)| \leq 2nk^{-r} \) with probability \( 1 - n^{-0.7\lambda+3} \). The number of pivots is a binomial distribution of \( B(|R(G', \rho_{\text{max}} D_r, v)|, \frac{\lambda k^{r+1} \log n}{n}) \) and therefore,

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

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

**Lemma 3.3.7.** Consider a call to \( \text{Hopset}(G = (V, E)) \). For all nodes \( v \in V \), the number of shorteners \( u \), 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 \( \text{HSRecurse}(G', D, r) \). For all \( v \in V \), the number of shorteners \( u \), such that \( v \in R(G', \rho_{\text{max}} D_r, u) \) is \( 6\lambda k^{L+1} \log n \) with probability at least \( 1 - n^{-0.7\lambda+4} \).
Proof. I 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 3.3.6 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 $u \in R(G', \rho_{\text{max}} D_r, v)$ 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
\[
\sum_{i=0}^{L} \lambda k^{i+1} \log n/n \leq 2\lambda k^{L+1} \log n/n, \quad \text{for } k \geq 2.
\]
The number of shortcutters in HOPSET($G$) is a binomial distribution $B(n, \frac{2\lambda k^{L+1} \log n}{n})$ and by a Chernoff bound,
\[
\Pr[B(n, \frac{2\lambda k^{L+1} \log n}{n}) > 6\lambda k^{L+1} \log n] \leq e^{-2\lambda k \log n} \leq n^{-2\lambda}.
\]

Now I can prove Theorem 3.3.1, 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 3.3.1. 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 3.3.5 and Lemma 3.3.6. 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 3.3.7, times the number of edges in all recursive subproblems, which is $2m \log n$ by Lemma 3.3.5. 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 3.3.5 and Lemma 3.3.7, this is \( O(nk^{L+1}\log^4 n) \) total hopset edges.

### 3.3.2 Hopbound

The goal in this section is to show the hopbound of the hopset produced by the \textsc{Hopset}(G) algorithm is \( n^{\frac{1}{2}+O(1/\log k)}k^{c+\frac{1-\frac{L}{2}}{2}}\log^2 n \). The main idea comes from Fineman [16] and JLS [21]. I 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 \textsc{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, I am still making progress because the number of nodes in path-relevant subproblems is reduced. Hence, I 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. The 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 [16]. However, mine becomes more complicated because I use multiple pivots, and fringe and core problems.

In Lemma 3.3.8, I 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. I will show this claim after Lemma 3.3.8 in Lemma 3.3.9. 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 I must choose which of these subproblems to consider in the analysis.

To resolve the first difficulty, I 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 \( \text{HSRECURSE}(G, D, r) \), but this will not change the 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, I introduce an additional \( O(\sqrt{k}) \) subproblems, as well as mul-
tiple copies of many nodes. Fortunately, since I 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 $|\hat{P}| = \ell \in (k^CD/2, k^CD]$. 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 I 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 4.4.3, each subpath
$P_i$ at level $r$ contains at most $2\ell k^{-r}$ related vertices. I have at most $\lambda^r k^{c+r/2}$ new
logical nodes since I have at most $\lambda^r k^{c+r/2}$ subpaths of length $D/(\lambda^r k^{c+r/2})$. Hence, I
only duplicate $2\lambda^r nk^{c-r/2}$ additional vertices in this procedure. Next, I 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 $\rho_v$ be the scalar of the searching distance for pivot $v$.

**Lemma 3.3.8.** 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
$P_0, P_1, P_2, ..., P_{|S|}$ 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 $0 \leq i \leq
   2|S|$ 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. $P_{|S|+1}, \ldots, P_{2|S|}$ are called in fringe problems and each $P_i$ is contained in some $V_u^{\text{Fringe}}$, where $u \in S$.

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

$$\sum_{i=0}^{|S|} E[|R_{\rho_{\mu_{e}}D_r}(G'[V_u], P_i)|] \leq \frac{3}{p_r}$$

and

$$\sum_{i=|S|+1}^{2|S|} E[|R_{\rho_{\mu_{e}}D_r}(G'[V_u^{\text{Fringe}}], P_i)|] \leq \frac{1}{p_r}.$$

Proof. If $u$ is a pivot and $u \preceq_{\rho_{u}D_r} P$ or $P \preceq_{\rho_{u}D_r} u$, then $u$ will put a $u^{\text{Anc}}$ or $u^{\text{Dec}}$ label on vertices on path $P$. Otherwise, $u$ does not put a label on any vertices on $P$.

If $u \preceq_{\rho_{u}D_r} P$ and $P \preceq_{\rho_{u}D_r} u$, then $u$ is a bridge and I will stop at this path-relevant subproblem.

I now consider the case where all the pivots are ancestors. Let $u$ be an ancestor pivot, and I will show $u$ divides $P$ into three subpaths, which are contained in path-relevant subproblems. The first subpath is contained in the fringe problem, the second one is contained in the core problem with label $u^{\text{Des}}$, and the third one is in the core problem without a label from $u$. To define these subproblems, consider the indices of the following two nodes on $P$. The first index is the node that is the earliest $\rho_{u}D_r$-descendant to $u$ on $P$,

$$\text{Fringe}(P, u) = \min\{i \mid u \preceq_{\rho_{u}D_r} v_i\}.$$

The second index is the node that is the earliest $(\rho_{u} - 1)D_r$- descendant to $u$ on $P$,

$$\text{Core}(P, u) = \min(\min\{j \mid u \preceq_{(\rho_{u} - 1)D_r} v_j\}, \ell + 1).$$

If no node on path $P$ is $(\rho_{u} - 1)D_r$-descendant of $u$, then set $\text{Core}(P, u) = \ell + 1$. Since any path node $v_j$ that is a $(\rho_{u} - 1)D_r$-descendant to $u$ is also a $\rho_{u}D_r$-descendant to $u$, $\text{Fringe}(P, u) \leq \text{Core}(P, u)$. Now split the path into three subpaths, $P^{\text{Unrelated}} = \langle v_0, \ldots, v_{\text{Fringe}(P, u)-1} \rangle$, $P^{\text{Fringe}} = \langle v_{\text{Fringe}(P, u)}, \ldots, v_{\text{Core}(P, u)-1} \rangle$ and
$P_{Core} = \langle v_{Core(P,u)}, ..., v_t \rangle$. If $\text{Fringe}(P,u) = 0$, then the unrelated subpath is defined as empty. If $\text{Fringe}(P,u) = \text{Core}(P,u)$, then the fringe subpath is defined as empty. If $\text{Core}(P,u) = \ell + 1$, then the core subpath is defined as empty.

Next I will show that $P^{Unrelated}$, $P^{Fringe}$ and $P^{Core}$ are contained in path-relevant subproblems. $P^{Unrelated}$ is trivial because $P^{Unrelated}$ gets no labels from $u$. For any node $v_{i'} \in P^{Fringe}$, $u \not\leq_{(\rho_u-1)D_r} v_{i'}$, because if $u \leq_{(\rho_u-1)D_r} v_{i'}$, then $i' \geq \text{Core}(P,u)$, which contradicts the fact $i' < \text{Core}(P,u)$. On the other hand, I can show $u \leq_{(\rho_u+1)D_r} v_{i'}$, which implies $v_{i'} \in R_{(\rho_u+1)D_r}(G,u) \setminus R_{(\rho_u-1)D_r}(G,u)$, and so $P^{Fringe}$ is in $u$’s fringe problem. By definition $u \leq_{\rho_uD_r} v_{\text{Fringe}(P,u)}$, and since the length of $P$ is at most $D_r$, $v_{\text{Fringe}(P,u)} \leq_{D_r} v_{i'}$. Therefore $u \leq_{(\rho_u+1)D_r} v_{i'}$. Lastly, I will show that any node $v_{j'} \in P^{Core}$, is a $\rho_uD_r$-descendant of $u$ and therefore gets $u^{Des}$ label. Since $u \leq_{(\rho_u-1)D_r} v_{Core(P,u)}$ by definition, and $v_{Core(P,u)} \leq_{D_r} v_{j'}$ by the length of the path being at most $D_r$, $u \leq_{\rho_uD_r} v_{j'}$. Thus, I have shown that each pivot $u$ will split $P$ into at most three subpaths, one that does not get a label from $u$, one that is in $u$’s fringe problem and one that gets $u^{Des}$ label.

Now consider the case that there are $t$ ancestor pivots. I will show the subpaths are contained in the unrelated, fringe, and core problems, and that the union of these subpaths is the whole path $P$. For each pivot $u$, define $P^{Fringe}_u = \langle v_{\text{Fringe}(P,u)}, ..., v_{Core(P,u)-1} \rangle$. $P^{Fringe}_u$ is in $u$’s fringe problem. Consider the $\text{Core}(P,u)$ values for all $t$ ancestor pivots in non-decreasing order, $c_1, c_2, ..., c_t$. Let $u_1, ..., u_i, ..., u_t$ be the corresponding ancestor pivots and for convenience, set $c_0 = 0$ and $c_{i+1} = \ell + 1$. Each vertex on the path $\langle v_{c_i}, ..., v_{c_{i+1}-1} \rangle$ gets a label from each of the $u_1, ..., u_i$ pivots. However, the subpaths become more complicated because of the fringe problem. In particular, if a node is in a fringe problem, I no longer need to consider it in the core problem. The $i$-th core path is $P_i = \langle v_{c_i}, ..., v_{f_i-1} \rangle$, where $f_i$ is defined as $f_i = \min(\min\{\text{Fringe}(P,u_j) \mid j > i\}, c_{i+1})$. $P_i$ gets a label from each $u_1, ..., u_i$ because
For ease of notation, let \( P_i \) be a path in the graph, where \( i \) is the index of the path node. If \( c_i \geq Core(P, u_j) \) for all \( j \leq i \), then \( P_i \) will be a core problem. On the other hand, the end index of \( P_i \) is at most \( f_i - 1 \), which means \( P_i \) does not get a label from any of \( u_{i+1}, \ldots, u_t \). Therefore, each \( P_i \) will be in a core problem. Finally, define \( P_0 = \langle v_0, \ldots, v_{c_1-1} \rangle \) and \( P_i = P_i^{Core} \) and so there are at most \( t + 1 \) core problems.

Now I will show the union of these subpaths is the whole path \( P \). I have showed any vertices in \( P_0 \) and \( P_i, i \in [1, t] \) are in the unrelated and core problems, respectively. The remaining vertices on \( P \) are vertices in \( \langle v_{f_i}, \ldots, v_{c_i+1-1} \rangle \). By definition, these are in fringe problems. Therefore, the union of \( P_0, P_i^{Fringe}, \) and \( P_i^{Core} \) for \( i \in [1, t] \) is the path \( P \).

Next, I need to consider the case where some pivots are descendants. This case becomes more complicated, but the basic idea is the same as just ancestor pivots. I will define the \( P_i^{Core} \) and \( P_i^{Fringe} \) subpaths and show that each vertex on these subpaths gets the appropriate label. Then I will show the union of these subpaths is the entire path. I first need to define \( Fringe(P, u) \) and \( Core(P, u) \) for a descendant pivot \( u \). \( Fringe(P, u) \) is the index of the latest path node that is a \( \rho_u D_r \) ancestor of \( u \),

\[
Fringe(P, u) = \max \{ i \mid v_i \preceq_{\rho_u D_r} u \} + 1.
\]

The core node is the latest node that is \( (\rho_u - 1)D_r \)-ancestor of \( u \) on \( P \),

\[
Core(P, u) = \max(\max \{ j \mid v_j \preceq_{(\rho_u - 1)D_r} u \}, -1) + 1.
\]

For convenience, I shift \( Fringe(P, u) \) and \( Core(P, u) \) by 1 index. If no node on path \( P \) is a \( (\rho_u - 1)D_r \)-ancestor of \( u \), then set \( Core(P, u) = 0 \). I use the same strategy as the ancestor case to define all subpaths \( P_i^{Fringe} \) in the fringe problem. If \( u \) is a descendant pivot, then \( P_u^{Fringe} = \langle v_{Core(P,u)}, \ldots, v_{Fringe(P,u)-1} \rangle \), and if \( u \) is an ancestor pivot, \( P_u^{Fringe} = \langle v_{Fringe(P,u)}, \ldots, v_{Core(P,u)-1} \rangle \). It is clear that \( P_u^{Fringe} \) is in \( u \)'s fringe problem based on the ancestor case. For the core problems, consider the \( |S| \) \( Core(P, u) \)
values in non-decreasing order, \(c_1, c_2, \ldots, c_{|S|}\). Let \(u_1, \ldots, u_i, \ldots, u_{|S|}\) be the corresponding pivot, and for convenience, set \(c_0 = 0\) and \(c_{|S|+1} = \ell\). Notice that some of the \(u_i\) are ancestor pivots, and some are descendant pivots. If a path node \(v_i\) is not in a fringe problem, then \(v_i\) will get a \(u_j^{Des}\) label from each ancestor pivot \(u_j\) where \(c_j \leq i\) and a \(u_j^{Anc}\) label from each descendant pivot \(u_j'\) where \(c_{j'} \geq i\). I will next show the subpaths in more detail. First define \(f_i = \min(\min\{\text{Fringe}(P, u_j) \mid j > i\}, c_{i+1})\) and \(g_i = \max(\max\{\text{Fringe}(P, u_j) \mid j \leq i\}, c_i)\). Now I can define path \(P_i = \langle v_{g_i}, \ldots, v_{f_i-1}\rangle\). It's possible that \(g_i \geq f_i\) in which case \(P_i\) is empty. Let \(A_i = \{u_j \mid j \leq i, u_j\text{ is an ancestor pivot}\}\) be the set of ancestors on the "left" of \(u_i\), and \(\bar{A}_i = \{u_j \mid j > i, u_j\text{ is an ancestor pivot}\}\) be the set of ancestors on the "right" of \(u_i\). I want to show each node on \(P_i\) gets a \(u_j^{Des}\) label from each \(u_j \in A_i\) and does not get any labels from nodes in \(\bar{A}_i\). The first claim is trivial since the core value of the nodes in \(A_i\) is less than or equal to \(c_i\), and so \(P_i\) gets a label from each node in \(A_i\). For the second claim, if any node \(u'\) on \(P_i\) gets label from ancestor pivot \(u_j \in \bar{A}_i\), then \(\text{Fringe}(P, u_j) \leq f_i-1\) in order for the search from \(u_j\) to reach a node on \(P_i\). However, this is a contradiction the definition of \(f_i\). Therefore \(P_i\) does not get any labels from \(u_j \in \bar{A}_i\). Next I will show the descendants case. Define \(D_i = \{u_j \mid j \geq i+1, u_j\text{ is a descendant pivot}\}\) and \(\bar{D}_i = \{u_j \mid j < i+1, u_j\text{ is a descendant pivot}\}\). These definitions are symmetric to the case for ancestors except they are shifted by 1 index to account for the shift by 1 in the definition in \(\text{Fringe}(P, u)\) and \(\text{Core}(P, u)\) for descendants. The proof that all nodes in \(P_i = \langle v_{g_i}, \ldots, v_{f_i-1}\rangle\) get an \(Anc\) label from each vertex in \(D_i\), and get no labels from any vertex in \(\bar{D}_i\) is symmetric to the ancestor case with the exception of the shift by 1 index. I can conclude that each \(P_i\) will be in a core problem. There are \(P_0, P_1, \ldots, P_{|S|}\) subpaths, which is at most \(|S| + 1\) subpaths.

The last thing left to prove is that the union of \(P_i\) for \(i \in [1, |S|]\), and \(P_u^{\text{Fringe}}\) for all \(u \in S\) is \(P\). This claim is straightforward because \(\langle v_{g_i}, \ldots, v_{f_i}\rangle\), for \(i \in [1, |S|]\), are
in core problems, and \( \langle v_{c_1}, ..., v_{g_i} \rangle \) and \( \langle v_{f_1}, ..., v_{c_{i+1}} \rangle \) are in fringe problems based on their definitions.

Now I will show the last part of the claim, which is the number of related nodes of \( P \) at level \( r+1 \). First, I will count the related nodes in the core problems and then the fringe problems. I will slightly overcount the related nodes in the core problems by counting all core path-relevant subproblems created by \( \rho_{\min} D_r \)- pivots. Notice that all \( \rho_{\min} D_r \)- pivots will be in \( S \) because the search distance is at least \( \rho_{\min} + 1 \). If I only consider part of \( S \) as pivots, I will increase the related nodes at level \( r+1 \).

Consider the related nodes in the core problem at level \( r+1 \). Any ancestor pivot that reaches the path in \( \rho_{\min} D_r \), reaches \( v_{\ell} \) in \( \rho_{\max} D_r \) because the length of \( P \) is at most \( D_r \). Using this fact, and Lemma 4.4.3, with probability \( 1 - n^{-0.7\lambda + 3} \), the following holds,

\[
|R_{\rho_{\min} D_r}^{-}(G, P)| \leq |R_{\rho_{\max} D_r}^{-}(G, v_{\ell})| \leq nk^{-r}.
\]

Assume that \( |R_{\rho_{\min} D_r}^{-}(G, P)| > 1.5/pr \), otherwise the claim holds trivially. Let \( Y_s \) be the number of ancestors at level \( r+1 \) in a logical path-relevant subproblem \( s \), and \( X_s \) be the \( \rho_{\min} D_r \) ancestors pivots in logical path-relevant subproblem \( s \). Note \( X_s \) is a binomial random variable of \( B(|R_{\rho_{\min} D_r}^{-}(G, P)|, pr) \), and \( E[X_s] = |R_{\rho_{\min} D_r}^{-}(G, P)| \cdot pr \leq \lambda k \log n \) with probability \( 1 - n^{-0.7\lambda + 3} \), where the probability comes from Lemma 4.4.3. Using a Chernoff bound, the number of ancestor pivots is bounded with high probability as follows,

\[
\Pr [X_s \geq 4\lambda k \log n] \leq e^{-2\lambda kn \log n} \leq n^{-2\lambda}.
\]

Hence, by taking a union bound over all \( v \in V \) and all \( r \), the rest of the proof holds with probability \( 1 - n^{-0.7\lambda + 4} \).
Notice that the scalar interval $|I| = 4\lambda^2k\log^2 n$. If $\lambda \geq 4$, then $|I| \geq 4X_s$. In Lemma 3.3.9 I will show that if $|I| \geq 4X_s$ then,

$$E[Y_s \mid X_s] \leq \frac{1.5}{X_s + 1} \cdot |R_{\rho_{\min,D^r}}^{-}(G, P)|.$$ 

The expectation of $Y_s$ is,

$$E[Y_s] = \sum_s E[Y_s \mid X_s] \cdot \Pr[X_s]$$

$$\leq 1.5|R_{\rho_{\min,D^r}}^{-}(G, P)| \cdot \sum_s \frac{1}{X_s + 1} \Pr[X_s]$$

$$\leq 1.5|R_{\rho_{\min,D^r}}^{-}(G, P)| \cdot E\left[\frac{1}{X_s + 1}\right]$$

$$\leq 1.5|R_{\rho_{\min,D^r}}^{-}(G, P)| \cdot \frac{1}{|R_{\rho_{\min,D^r}}^{-}(G, P)|} \cdot \frac{1}{\Pr[X_s]} = \frac{1.5}{p_r},$$

where the first line comes from the definition of expectation and Lemma 3.3.9, the second line comes from the definition of expectation, and the last line comes from $X_s$ being a binomial random variable. The descendants case is symmetric. In total,

$$\sum_{i=0}^{\lfloor S\rfloor} E[|R_{\rho_{\min,D^r}}(G'_{V_{ai}}, P_i)|] \leq \frac{3}{p_r},$$

with probability $1 - n^{-0.7\lambda+4}$.

Lastly, I will count the related nodes in the fringe problems. Notice that for all pivots $u, u \in R_{(\rho_{\max-1})D^r}(G, P)$, and earlier in the proof I showed $|R_{(\rho_{max-1})D^r}(G, P)| \leq |R_{\rho_{\max,D^r}}^+(G, v_0)| + |R_{\rho_{max,D^r}}^-(G, v_\ell)| \leq 2nk^{-r}$. By Lemma 3.3.4, the number of nodes that each node adds to its fringe problem is $1/(4\lambda k \log n)$ in expectation. Therefore,

$$\sum_{i=\lfloor S\rfloor + 1}^{2\lfloor S\rfloor} E[|R_{\rho_{\min,D^r}}(G'_{V_{u}^{\text{range}}}, P_i)|]$$

$$\leq |R_{(\rho_{\max-1})D^r}(G, P)| \cdot \frac{1}{4\lambda k \log n} \leq \frac{2}{4\lambda k^{r+1} \log n/n} \leq \frac{1}{p_r}. \quad \Box$$
Now I will show the helper lemma for the case the pivots are ancestors and descendants. Fineman [16] shows a similar result when there is just one pivot, and JLS [21] extends this to \( t \) pivots. In my case, I have the additional difficulty that each pivot searches for a different distance, but I am able to get the same result.

**Lemma 3.3.9.** Consider the path \( P = \langle v_0, v_1, ..., v_\ell \rangle \), where \( \ell \leq D_r \), and its \( \rho_{\text{min}}D_r \)-distance ancestor set \( R_{\rho_{\text{min}}D_r}^-(G, P) \) 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 \( R_{\rho_{\text{min}}D_r}^-(G, P) \). Let \( P_i \) be the path defined in Lemma 3.3.8. If the chosen interval \( |I| \geq 4t \), then

\[
\sum_{i=0}^{|S|} E[|R_{\rho_{\text{min}}D_r}^-(G'[V_{a_i}], P_i)|] \leq \frac{1.5}{t+1} |R_{\rho_{\text{min}}D_r}^-(G', P)| .
\]

*Proof.* For two nodes, \( u, w \), define the relation \( u \text{ knocks out } w \) to mean that if \( u \) is a pivot and \( w \) is not in level \( r \), then \( w \) is not path-relevant in a path-relevant subproblem in the level \( r+1 \) i.e. \( w \notin R_{\rho_{\text{min}}D_r}^-(G'[V_{a_i}], P_i) \) for all \( i \in [1, |S|] \). If \( u \) does not knock out \( w \), it does not mean \( w \) is path-relevant in a path-relevant subproblem in the next level. Other pivots may knock out \( w \), in which case \( w \) would not be path-relevant. In the proof, I will prove lemma 3.3.9 by three steps. In the beginning, recall \( u \) and \( w \) will choose an interval scalars \( \sigma_u \) and \( \sigma_v \) if they are chosen as pivots, respectively. I will show that for most interval scalars in \( I \), either \( u \) knocks out \( v \) or \( v \) knocks out \( u \). Second, based on the first claim, I am able to show for any fixed set \( S' \in R_{\rho_{\text{min}}D_r}^-(G', P), |S'| = t+1 \), if I choose \( t \) pivots in \( S' \), the left node will be a \( \rho_{\text{min}}D_r \)-ancestor in next level with probability \( 1.5/(t+1) \). Last, I need to use the second claim to prove if I choose \( t \) pivots from \( R_{\rho_{\text{min}}D_r}^-(G', P) \), the expectation in the lemma claim holds.

For any two vertices \( u, w \) and interval scalar \( \sigma \in I \), let \( \chi(u, w, \sigma) \) be the indicator that \( u \) doesn’t knocks out \( w \), when \( \sigma \) is the interval scalar for \( u \). I will show that for
u and w, there is at most one scalar \( \sigma \in I \) where \( \chi(u, w, \sigma) = 1 \) and \( \chi(w, u, \sigma) = 1 \). W.l.o.g. assume \( \text{dist}_{G'}(u, w) \leq \text{dist}_{G'}(w, u) \). Let \( h \) be \( \text{dist}_{G'}(u, w) \). Recall that the algorithm chooses a random interval defined by the interval scalar \( \sigma \), and then chooses another scalar \( \rho \) from the corresponding interval, which minimizes the size of the fringe problem. As a slight of abuse of notation, let the interval with corresponding \( \text{dist}_{G'}(u, w) \) be denoted \([\sigma_{\text{min}}, \sigma_{\text{max}}]\). Therefore, \( \rho \in [\sigma_{\text{min}}, \sigma_{\text{max}}] \). First, I will show that in the cases where \( \sigma_{\text{min}}D_r > h \) and \( \sigma_{\text{max}}D_r \leq h \), at most one of \( \chi(u, w, \sigma) \) and \( \chi(w, u, \sigma) \) equals 0. Therefore, only the interval \([\sigma_{\text{min}}, \sigma_{\text{max}}]\) can make \( \chi(u, w, \sigma) = 1 \) and \( \chi(w, u, \sigma) = 1 \).

For the first case where \( h < \sigma_{\text{min}}D_r \), Fineman [16] has given a similar proof that either \( u \) knocks out \( w \) or \( w \) knocks out \( u \). If \( w \) is a pivot, then \( u \) gets a \( w^{\text{Anc}} \) because \( u \leq_{\rho_uD_r} w \), where \( \rho_u \) is the scalar for \( u \) and \( \rho_u \geq \sigma_{\text{min}} \). However, in the proof of Lemma 3.3.8, I showed that none of subpath \( P_i \) in the core problem gets \( w^{\text{Anc}} \) label. Therefore, \( w \) knocks out \( u \).

The second case is \( h \geq \sigma_{\text{max}}D_r \). Let \( c_u = \text{Core}(P, u) \) and \( c_w = \text{Core}(P, w) \). Notice that for each \( \rho_{\text{min}}D_r \)-distance ancestor pivot, since the search distance is at least \( (\rho_{\text{min}} + 1)D_r \), \( c_u \leq \ell \) and \( c_w \leq \ell \). Consider two more indices, \( c'_u = \min\{j \mid u \preceq_{\rho_{\text{min}}D_r} v_j\} \), and \( c'_w = \min\{j \mid w \preceq_{\rho_{\text{min}}D_r} v_j\} \). Notice that \( u, w \in R_{\rho_{\text{min}}D_r}^-(G, P) \), and so \( c'_u \) and \( c'_w \) are always valid. Even if \( u \) is not a pivot, \( u \) can only be a \( \rho_{\text{min}}D_r \)-ancestor for a subpath \( P_i \) such that \( \text{tail}(P_i) \geq c'_u \). This is because the shortest path between two nodes never decreases in the algorithm. If \( u \) is not a \( \rho_{\text{min}}D_r \)-ancestor for any subpath \( P' \), \( u \notin R_{\rho_{\text{min}}D_r}^-(G'[V'], P') \) holds throughout the rest of the algorithm. Note that \( c_u \leq c'_u \) and \( c_w \leq c'_w \), because \( \rho_u \geq \rho_{\text{min}} + 1 \) and \( u \preceq_{(\rho_u-1)D_r} v_{c'_u} \) based on the definition of \( c'_u \). First assume \( c_u \leq c'_w \). If \( u \) is a pivot, the subpath for \( u \) is \( P_{iu} \). In Lemma 3.3.8 I have shown all paths \( P_i \) get \( u \)’s descendant label, where \( i \geq i_u \). However, \( w \) never gets a descendant label from \( u \) since \( h \geq \sigma_{\text{max}}D_r \). Thus, \( w \) could not be a \( \rho_{\text{min}}D_r \)-ancestor for all paths \( P_j \) where \( j \geq i_u \). On the other hand, \( w \)
could not be a $\rho_{\min}D_r$-ancestor for all paths $P_j$ where $j < i$, since $\text{tail}(P_j) < c_u \leq c'_w$ and I already know $w$ could not be a $\rho_{\min}D_r$-ancestor for subpath $P_j$ if $\text{tail}(P_j) < c'_w$. Therefore, $\chi(u, w, \sigma) = 0$ if $c_u \leq c'_w$. If $c_w \leq c'_u$, the situation will be symmetric. In order for $\chi(u, w, \sigma) = 1$ and $\chi(w, u, \sigma) = 1$ to hold at the same time for $h \geq \sigma_{\max}D_r$, it has to be the case that $c_u > c'_w$ and $c_w > c'_u$. Combining with the fact $c_u \leq c'_u$ and $c_w \leq c'_w$, this is impossible. Therefore, when $h > \sigma_{\max}D_r$, one of $\chi(w, u, \sigma)$ and $\chi(u, w, \sigma)$ is 0.

Next, for any $t+1$ ancestors set $S' \in R^-_{\rho_{\min}D_r}(G', P)$, the goal is to show that if I randomly choose $t$ pivots from $S'$, the leftover node $u_i$ is path-relevant in the next level with probability at most $1.5/(t+1)$. For convenience, denote $R^-_{\rho_{\min}D_r}(G', P)$ as $R^-$.

Let $E_{\text{left}}$ be the event that left node $u_i$ is path-relevant in the next level. Notice that the probability is

$$\Pr[E_{\text{left}}] = \frac{\sum_{i=1}^{t+1} \sum_{\sigma_{u_1}, \sigma_{u_2}, \ldots, \sigma_{u_{t+1}} \in I} \prod_{j \neq i} \chi(u_j, u_i, \sigma_{u_j})}{(t+1)|I|^t}.$$  

That’s because if $u_i$ is the left pivot and each $u_j$’s scalar is $\sigma_{u_j}$, where $j \neq i$, then $u_i$ is path-relevant in the next level if an only if all $u_j$ doesn’t knock out $u_i$ for $j \neq i$, i.e., $\chi(u_j, u_i, \sigma_{u_j}) = 1$ holds for all $j \neq i$. Let $A_{ji} = |\{\sigma | \chi(u_j, u_i, \sigma) = 1, \sigma \in I\}|$ be the number of interval scalars such that $u_j$ doesn’t knock out $u_i$, then I can rewrite the probability as

$$\Pr[E_{\text{left}}] = \frac{\sum_{i=1}^{t+1} \prod_{j \neq i} A_{ji}}{(t+1)|I|^t}.$$  

For each $u_i$ and $u_j$, there is at most one $\sigma$ making $\chi(u_j, u_i, \sigma) = 1$ and $\chi(u_i, u_j, \sigma) = 1$. Therefore I have $A_{ij} + A_{ji} \leq |I| + 1$. $\sum_{i=1}^{t} \prod_{j \neq i} A_{ji}$ will be maximized when one of $A_{ij}$ and $A_{ji}$ equals to $|I| + 1$. Therefore,

$$\Pr[E_{\text{left}}] \leq \frac{(|I| + 1)^t}{(t+1)|I|^t} \leq \frac{1}{t+1} \cdot (1 + \frac{1}{|I|})^{|I|/4} \leq \frac{1.5}{t+1}.$$
Finally, I use $\Pr \left[ E_{\text{left}} \right]$ to show the expectation when I choose $t$ pivots in $R^-$. Let $I(S, u)$ be the indicator that a node $u$ is path-relevant in the next level when $S$ is the set of chosen pivots. Thus, for any fixed $S'$, I have

$$\sum_{S \subseteq S', |S| = t} I(S, S' \setminus S) = (t + 1) \Pr \left[ E_{\text{left}} \right] \leq 1.5.$$

Summing up over all possible $S' \in R^-$,

$$\sum_{S' \subseteq R^-, |S'| = t + 1} \sum_{S \subseteq S', |S| = t} I(S, S' \setminus S) \leq 1.5 \left( \frac{|R^-|}{t + 1} \right).$$

The above can be rewritten as

$$\sum_{S \subseteq R^-, |S| = t \atop u \in R^\setminus S, S' = S \cup \{u\}} \sum_{S' \subseteq S' \setminus S} I(S, S' \setminus S) \leq 1.5 \left( \frac{|R^-|}{t + 1} \right).$$

The above formula first chooses $S$ then chooses a vertex $u \notin S$ while the previous formula first chooses $S'$ then chooses a vertex $u \in S'$. Notice that if $u \in S$, then $I(S, u) = 0$. Therefore,

$$\sum_{S \subseteq R^- \atop u \in R^- \setminus S} I(S, u) \leq \sum_{S \subseteq R^- \atop \exists u \in R^\setminus S, S' = S \cup \{u\}} \sum_{S' \subseteq S' \setminus S} I(S, S' \setminus S) \leq 1.5 \left( \frac{|R^-|}{t + 1} \right).$$

Hence, if the algorithm chooses $t$ ancestor pivots, the expected number of ancestors in the next level is

$$\sum_{i=1}^{|S|+1} E_{|S|=t}[R^-_{\rho_{\text{min}}D_r}(G'[V_{a_i}], P_i)] = E_{|S|=t}[\sum_{u \in R^-} I(S, u)] = \sum_{S \subseteq R^- \atop u \in R^-} I(S, u) \Pr [S] \leq 1.5 \left( \frac{|R^-|}{t + 1} \right) / \left( \frac{|R^-|}{t} \right)$$

$$\leq \frac{1.5}{t + 1} |R^-_{\rho_{\text{min}}D_r}(G', P)|.$$

□
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 $|P_i| \leq D/((\lambda^{r}k^{c/2})$. I construct new logical layer based on the rule I mentioned before. Next, based on the path-relevant subproblem tree, I will give a lemma about the expected number of related nodes and subproblems in each level of recursion.

**Lemma 3.3.10.** 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 $r \geq 0$,

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

**Proof.** To show the claim, I 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. The target is to show the following formula holds with probability $1 - n^{-0.7\lambda+4}$, for all $r$,

$$
E[Y_r] \leq 4\lambda^{r}nk^{c-\frac{r}{2}}
$$

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

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

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

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

$$
\bigcup_{r \leq \log_k n - L} \Pr \left[ Z_r \geq 32\lambda^{r}k^{c+\frac{r+1}{2}} \log^2 n \right]
\leq \frac{\log_k n - L}{2\log n} + n^{-0.7\lambda+4} \leq \frac{1}{2}.
$$
Next I 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 I 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 4.4.3, 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,

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

Next I can count \( Z_{r+1} \) and \( Y_{r+1} \) based on the logical layer. By Lemma 3.3.8, 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,

\[
E[Y_{r+1}] = E[\sum_s Y_s] = \sum_s E[Y_s] = \sum_s \sum_{Z'_r} E[Y_s \mid Z'_r] \Pr[Z'_r]
\]

\[
\leq 4 \frac{\sum_{Z'_r} Z'_r \Pr[Z'_r]}{\lambda^{r+1} \log n/n} \cdot (E[Z_r] + \lambda^r k^{c+r/2})
\]

\[
\leq 64\lambda^{r-1} nk^{c-\frac{r+1}{2}} \leq 4\lambda^{r+1} nk^{c-\frac{r+1}{2}}
\]

for \( \lambda \geq 4 \). 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 $k \geq 2$ then,

$$E[Z_{r+1}] = \sum E[Z_{r+1} \mid Y'_r] \cdot \Pr[Y'_r] = p_r \cdot \sum 2Y'_r \Pr[Y'_r] + E[Z'_r]$$

$$= \frac{2\lambda k^{r+1} \log n}{n} \cdot E[Y'_r] + E[Z'_r]$$

$$\leq \frac{2\lambda k^{r+1} \log n}{n} \cdot (4\lambda' nk^{c-r/2} + 2\lambda' nk^{c-r/2})$$

$$+ 15\lambda' k^{c+1/2} \log n + \lambda' k^{c+r/2}$$

$$\leq 15\lambda' k^{c+1+r/2} \log n.$$
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 3.3.12.** Consider any graph \( G' = (V, E) \) and an execution of \( \text{Hopset}(G') \) 
with parameters \( k, \lambda \) 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 \( u = \text{head}(\hat{P}) \) and \( v = \text{tail}(\hat{P}) \). By Lemma 3.3.11, 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 sub-
problem tree at level \( r \). Since the algorithm is repeated \( \lambda \log n \) times, there exists a 
path relevant tree such that \( \bigcap_{r \leq \log k n-L} Z_r \leq 32 \lambda^r k^{c+\frac{c+1}{2}} \log^2 n \) holds with probability 
\( 1 - n^{-\lambda} \), by Lemma 3.3.10. Therefore the hopbound is,

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

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

3.3.3 Approximation

In this section, I will show the approximation that the algorithm achieves. I have 
already showed that the path-relevant tree has \( n^{1/2+O(1/\log k)k_c+\frac{1-L}{2}} \log^2 n \) nodes, which 
means there exist a path \( P' \) that contains at most \( n^{1/2+O(1/\log k)k_c+\frac{1-L}{2}} \log^2 n \) hops. Now 
I 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 I 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, I have the error

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

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

Now I need to bound these two errors, I requires $128\lambda^2 k^{2-c} \log^2 n \cdot \ell \leq \epsilon l$ and $4096\lambda^2 L k^{5-L/2} \log^5 n \cdot \ell \leq \epsilon l$. If $k = \Omega(\log n)$, then $c \geq 4-\log_k \epsilon$ and $L \geq 15-2 \log_k \epsilon$.

If I set $c = 5-\log_k \epsilon$ and $L = 15-2 \log_k \epsilon$. The hopbound $\beta$ is at most $6\lambda \log_k n n^{1/2}$. The running time is $O(m k^{16} \log^4 n/\epsilon^2)$ and the hopset size is $O(n k^{16} \log^4 n/\epsilon^2)$. Combining all this together, the following corollary holds.

**Corollary 3.3.13.** 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 $O(n k^{16} \log^4 n/\epsilon^2)$ in running time $O(m k^{16} \log^4 n/\epsilon^2)$ with probability $1 - n^{-\lambda^2+2}$.

**Proof of Theorem 3.3.1.** From Theorem 3.3.2 and Corollary 3.3.13, Theorem 3.3.1 follows directly. \hfill $\square$
3.3.4 Tradeoff of the Hopsets

In section 3.3.3, I show that the \((\beta = n^{1/2+O(1/\log k)}, \epsilon)\)-hopset of size \(O(nk^{16} \log^4 n/\epsilon^2)\) can be constructed in running time \(O(mk^{16} \log^4 n/\epsilon^2)\), w.h.p. In this section, I will show that I can get a trade-off between the hopbound, hop size, and the running time. Notice that I only have a lower bound requirement for \(c\) and \(L\). If I leave \(c = 5 - \log_k \epsilon\) and increase the value of \(L\), the approximate ratio still holds, and the hopbound is decreasing at the expense of increasing the hopset size. More specifically, if I want the hopbound to be \(6\lambda^{\log_k n} n^{1/2}/\rho\), I can set \(L = 2 \log_k \rho - 2 \log_k \epsilon + 15\). The running time is \(O(mp^2k^{16} \log^4 n/\epsilon^2)\) and the hopset size is \(O(np^2k^{16} \log^4 n/\epsilon^2)\).

**Corollary 3.3.14.** For any unweighted directed graph \(G = (V, E)\) and any \(\rho \in [1, n^{1/2+O(1/\log k)}]\), \(\text{HOPSET}(G)\) with the above parameter returns a \((n^{1/2+O(1/\log k)})/\rho, \epsilon)\)-hopset of size \(O(n\rho^2k^{16} \log^4 n/\epsilon^2)\) in running time \(O(mp^2k^{16} \log^4 n/\epsilon^2)\) with probability \(1 - n^{-\lambda + 2}\).

Corollary 3.3.14 can be used to build parallel algorithms with higher work and lower span.

### 3.4 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. The 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 I will present the algorithm, and in Section 3.4.2 I provide the analysis.
3.4.1 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 \(n\).

Algorithm 3 Hopset algorithm for weighted directed graphs. \(k, \lambda\) and \(L\) are parameters.

1: function WHOPSET\((G = (V, E))\)  
2: \(H \leftarrow \emptyset\)  
3: repeat \(\lambda \log n\) times  
4: \(\text{for each } j \in [-1, \log(nW)]\)  
5: \(\text{for each } v \in V\)  
6: \(\text{for each } i \in [0, \log_k n]\)  
7: \(\text{With probability } (\lambda k^{i+1} \log n)/n, \text{ set } \ell(v) \text{ to } i,\)  
8: if setting successful.  
9: \(\text{if } \ell(v) \leq L \text{ then}\)  
10: \(\text{for each } u \in R_{2^{j+1}}^+(G, v) \text{ add edge } (v, u) \text{ to } H \text{ with weight } dist_G(v, u)\)  
11: \(\text{for each } u \in R_{2^{j+1}}^-(G, v) \text{ add edge } (u, v) \text{ to } H \text{ with weight } dist_G(u, v)\)  
12: \(H \leftarrow H \cup \text{HSRecurse}(G, D = 2^j k^{-c}, r = 0)\)  
13: return \(H\)

3.4.2 Analysis

The goal of this section is to prove Theorem 3.4.1.

**Theorem 3.4.1.** For any weighted directed graph \(G = (V, E)\), there exists a randomized algorithm that computes a \((\beta = n^{1/2+o(1/\log k)}, \epsilon)\)-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, I will show the difference in the runtime in Lemma 3.4.2 and then the hopbound and approximation.

**Lemma 3.4.2.** One execution of $\text{WHopset}(G = (V, E))$ 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 3.3.2. The only difference 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 $O(m + n\log n)$ [11]. 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 $O(nk^{L+1}\log^3 n\log(nW))$. \hfill $\square$

Next, I consider the hopbound of the weighted case. I 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 $w(P) > D_r = \frac{D}{k^{r/2}}$ then split $P$ into $q$ disjoint subpaths $P = P_1, P_2, ..., P_q$ such that $(\text{head}(P_i), \text{tail}(P_{i+1})) \in P$ 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 $w(P_i) + w(\text{head}(P_i), \text{tail}(P_{i+1})) > D_r$, 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, I 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 $j = -1$, I am considering the path $p$ with $w(p) < 1/2$. However, there is only $\epsilon$ 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 $c$, $\text{WHOPSET}(G = (V, E))$ will return a $(n^{1/2+o(1)}, \epsilon)$- hopset for $G$. For the final error to be $\epsilon$, set $c = 5 - \log k \epsilon$ and $L = 15 - 2 \log k \epsilon$. Combining the above analysis, gives us Theorem 3.4.1.

On the other hand, the tradeoff for the unweighted also works for the weighted case, if I set $L = 2 \log k \rho + 15 - 2 \log k \epsilon$ and leave $c$ unchanged, the hopsets size will increase, while the hopbound is decreasing.

**Corollary 3.4.3.** For any weighted directed graph $G = (V, E)$, there exists a randomized algorithm that computes a $(n^{1/2+O(1/\log k)}/\rho, \epsilon)$-hopset of size $O(n \rho^2 k^{16} \log^3 n \log(nW)/\epsilon^2)$. The randomized algorithm runs in $O(n \rho^2 k^{16} \log^4 n \log(nW)/\epsilon^2)$ time with probability $1 - n^{-\lambda+2}$.

### 3.5 Parallel Algorithm

In this section, I show how to extend the weighted hopsets algorithm to a work-efficient, low span parallel algorithm. First, I will explain the difficulties of the hopsets algorithm in the parallel setting and give the high-level idea of overcoming these difficulties. Then I describe the details of the parallel algorithm for hopsets in Section 3.5.1. Finally, in Section 3.5.2, I 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, I use the rounding technique from Klein and Subramanian [23]. Consider a path from \( v_0 \) to \( v_\ell \), \( \hat{P} = \langle v_0, v_1, ..., v_\ell \rangle \). For each edge \( e \in \hat{P} \), \( w(e) \) is rounded up to the nearest integer multiple of \( \delta w(\hat{P})/\ell \), where \( \delta \) is a small number to be set later. Since \( \hat{P} \) contains \( \ell \) 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 \( \delta \) 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 \( R_{p,v,D}(G, v) \) and \( R_{p,v,D}^-(G, v) \) in a call to HSRecurse\((G, D = O(\ell/\delta), r)\). The cost of the depth-first search depends only on \( \ell \) 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 \( 2\beta \) hops, where \( \beta \) 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, I 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 \( 2\beta \). One more execution gives the \( \beta \) hopbound.
Algorithm 4 Parallel hopset algorithm for weighted directed graphs. $\delta, k, \rho, \lambda, c, L$ are parameters.

1: function PHopset$(G = (V, E))$
2: $H \leftarrow \emptyset$
3: $\beta \leftarrow 6\lambda \log k n^{1/2}/\rho$
4: repeat $\lambda \log n$ 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: $\tilde{w}(e) = \begin{cases} +\infty & \text{if } w(e) \geq 2^{i+1} \\ \left\lfloor \frac{w(e)}{\hat{w}} \right\rfloor & \text{if } w(e) < 2^{i+1} \\ 1 & \text{if } w(e) = 0 \end{cases}$
10: for each $v \in \hat{V}$
11: for each $i' \in [0, \log n]$
12: With probability $(\lambda k^{i'+1} \log n)/n$, set $\ell(v)$ to $i'$,
13: if setting successfully.
14: if $\ell(v) \leq L$ then
15: for each $u \in R_{8(1+\delta)\beta/\delta}(G, v)$ add edge $(v, u)$ to $\hat{H}'$ with weight $\text{dist}_{\hat{G}}(v, u)$
16: for each $u \in R_{8(1+\delta)\beta/\delta}(G, v)$ add edge $(u, v)$ to $\hat{H}'$ with weight $\text{dist}_{\hat{G}}(u, v)$
17: $H \leftarrow H \cup (\hat{w} \cdot \hat{H}') \cup (\hat{w} \cdot \text{HSRECURSE}(\hat{G}, D = 4(1+\delta)\beta/(\delta k^c), r = 0))$
18: $E \leftarrow E \cup H$
19: return $H$
3.5.1 Algorithm Description

In this section, I describe the parallel algorithm, PHOPSET(G), shown in Algorithm 4. The parallel algorithm extends the hopsets algorithm for weighted graphs in Section 3.4. 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. I will describe these two steps in more detail.

One key modification to Algorithm 4 is as follows. In Lines 17-18, 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.

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 3.3.2.

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

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

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

if \( k^eD = 4(1 + \delta)\beta/\delta \). Since \( \tilde{w} \) 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, \( \tilde{w} \) is ignored in the rounding step and added back when \( \text{HSRECURSE}(G, D, r) \) returns the hopset.

Adding hopset edges to the graph. After a recursive call to \( \text{HSRECURSE}(G, D, r) \), Line 18 in Algorithm 4 adds the edges returned by \( \text{HSRECURSE}(G, D, r) \) to the original graph \( G \). \( \text{HSRECURSE}(G, D, r) \) returns a \((\beta, \epsilon)\)-hopset for any path with length at most \( 2\beta \) 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 \( |P'| = \max(|P|/2, 2\beta) \).

3.5.2 Parallel Hopbound and Hopset Size

**Lemma 3.5.1.** Consider any graph \( G' = (V, E) \) and an execution of \( \text{PHOPSET}(G') \).

For any \( P \) where \( |P| \leq 2\beta \), after the rounding code in Lines 6-9, suppose \( \text{HSRECURSE}(G, D, r = 0) \) returns a \((1 + \epsilon')\)-approximate path \( P' \) containing at most \( \beta \) hops with probability at least \( 1/2 \). If Lines 5-16 in \( \text{PHOPSET}(G') \) 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 \( w(\hat{P}') \leq (1 + \delta)^j(1 + \epsilon')^jw(P) \).

**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 \( 1 - \frac{1}{2^{\lambda\log n}} = 1 - n^{-\lambda} \), \( \text{HSRECURSE}(G, D, R) \) 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, I 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 \( \beta \) 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 \(|P| \leq 2\beta \), \( \text{HSRecurse}(G, D, r) \) with corresponding \( \hat{w} \) returns a \((\beta, \epsilon')\)-hopset for \( P \). By repeating Lines 5-16 \( \lambda \log^2 n \) times, Lemma 3.5.1 can be applied to all possible paths. The maximum path weight will increase each round, but it will be no greater than \((1 + \epsilon)^{\log n} n W \leq n^2 W \). Thus a maximum path weight of \( n^2 W \) covers all possible paths. Finally, to get a \((\beta, \epsilon)\)-hopset, set \( \delta = \epsilon/(8 \log n) \) and \( \epsilon' = \epsilon/(8 \log n) \). If \( k = \Omega(\log n) \), then \( c = 5 - \log_k \epsilon, \rho = 1 \) and \( L = 17 - 2 \log_k \epsilon \) is sufficient. The constant \( 1/8 \) in \( \epsilon' \) will cancel out with the \( \lambda^{-L} \) in the error formula.

Recall that \( \text{HSRecurse}(G, D, r = 0) \) 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 3.5.2.** For any weighted directed graph \( G = (V, E) \), \( \text{PHopset}(G) \) 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 \( 1 - n^{-\lambda+3} \).
3.5.3 Work and Span

Here I consider PHOPSET(G) in the work-span model [11]. 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 [9]. See Fineman [16] and JLS [21] for details of the parallel implementation. From the proof of Theorem 3.3.2, 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,

$$O(\sum_{j=1}^{\lambda \log^2 n} (m + jnk^{18} \log^2 n \log(nW) / \epsilon^2)k^{18} \log^2 n \log(nW) / \epsilon^2))$$

$$= O(mk^{18} \log^4 n \log(nW) / \epsilon^2 + nk^{36} \log^8 n \log^2(nW) / \epsilon^4).$$

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 3.5.3.** For any directed graph $G = (V, E)$ with non-negative real edge weights, 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
\(O(m \log^2 n \log(nW)/\epsilon^2 + n \log^4 n \log^2(nW)/\epsilon^4)\) work and \(n^{1/2+o(1)}/\epsilon\) span with high probability.

Proof. Combining above analysis and Corollary 3.5.2, the theorem holds with \(k = \Theta(\log n)\) and appropriate \(\lambda\).

**Theorem 3.5.4.** 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 3.5.3, \(\text{PHopset}(G)\) 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 shortest paths [23] completes the proof.

### 3.5.4 Work and Span Tradeoff

I already construct a nearly linear size hopsets with nearly linear work in the parallel model. Sometimes, I can afford higher work to get lower span. The tradeoff begins with the extension of Corollary 3.5.1. Let \(\rho\) be a parameter decided by input and I set up a different parameter for \(L = 17 - 2 \log_k \epsilon + 2 \log_k \rho\) and keep all other parameters unchanged. The proof follows the proof of Corollary 3.5.1 but use the hopsets tradeoff, which is implied in Corollary 3.3.14 and 3.4.3.

**Corollary 3.5.5.** For any weighted directed graph \(G = (V, E)\), \(\text{PHopset}(G)\) with above parameter returns a \((n^{1/2+o(1)/\log_k} / \rho, \epsilon)\)-hopset of size \(O(n \rho^2 k^{18} \log n \log(nW)/\epsilon^2)\) with probability \(1 - n^{-\lambda+3}\), where \(\rho \in [1, \sqrt{n}]\).

Once I have Corollary 3.5.5, I are going to show the work and span tradeoff to construct hopsets. For the work part, I add edges back, which is much more dense in
the tradeoff case. The total work is,

$$O(\lambda \log^2 n \sum_{j=1}^{\lambda \log^2 n} (m + jn^2k^{18} \log^2 n \log(nW)/\epsilon^2)\rho^2k^{18} \log^2 n \log(nW)/\epsilon^2)$$

$$= O(m^2k^{18} \log^4 n \log(nW)/\epsilon^2 + n^4k^{36} \log^8 n \log^2(nW)/\epsilon^4).$$

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

**Theorem 3.5.6.** For any weighted directed graph $G = (V, E)$, there exists a randomized parallel algorithm for weighted graphs that computes a $(n^{1/2+o(1)}/\rho, \epsilon)$-hopset of size $O(n^2 \log^2 n \log(nW)/\epsilon^2)$. The algorithm has $O(m^2 \log^2 n \log(nW)/\epsilon^2 + n^4 \log^4 n \log^2(nW)/\epsilon^4)$ work and $n^{1/2+o(1)}/(\epsilon \rho)$ span with high probability.
4.1 Introduction

This section presents an improved algorithm for approximate single-source shortest paths (SSSP) in the CONGEST model and broadcast CONGEST model. The algorithm follows the framework from Forster and Nanongkai [17] for distributed shortest paths. One of the steps in their framework involves computing shortest paths to a carefully selected subset of the vertices by simulating a parallel algorithm for SSSP in CONGEST. I achieve the improved bound by replacing the algorithm used in this step, instead adapting the parallel algorithm in Chapter 3 to the CONGEST model.

4.1.1 Results and Technique

The main result in this Chapter is captured by the following theorem.

Theorem 4.1.1. In the CONGEST and broadcast CONGEST model, there exists a randomized algorithm that solves $(1 + \epsilon)$-approximate single-source shortest-paths problem for directed $n$-node graph $G$ with non-negative real weights, in $\tilde{O}((\sqrt{n} + D + D^{2/5}n^{2/5+o(1)})\log W/\epsilon^2)$ rounds, with high probability, where $D$ is the undirected diameter of $G$ and $W$ is the ratio between heaviest and non-zero lightest edge weight.

Across much of the range of network diameters, my algorithm’s round complexity beats previous algorithms by a polynomial factor (albeit a very small one). Note that
I only solve the approximate version of the problem, whereas some of the prior art solves that exact version. I

4.2 Preliminaries

The following lemma is a standard result for distributed computation in the CONGEST and broadcast CONGEST model.

**Lemma 4.2.1.** [27] 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.

4.3 Algorithm

Next I present an overview of the algorithm, which extends the FN [17] framework. The algorithm is parameterized by \( \alpha \), to be set later. Steps 1, 3 and 5 are the same as FN [17], and step 2 is similar. In step 2, FN computes distance estimates from each skeleton node to each node in the original graph. The step 2 does the same computation and additionally computes the distance from each node in the original graph to each skeleton node. The additional distances estimates are used in the computation of step 4. The main difference in the algorithm is step 4. Both algorithms solve approximate SSSP on the skeleton graph, however I use a different algorithm to compute SSSP. The algorithm for computing step 4 is discussed in the next section.

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 a pair of nodes \( u, v \), define a \( (1 + O(\epsilon)) \)-approximate \( g \)-hop distance estimate \( \tilde{d}(u, v) \) to be an estimate of \( d_G^{(g)}(u, v) \) such that \( d_G^{(g)}(u, v) \leq \tilde{d}(u, v) \leq (1 + O(\epsilon))d_G^{(g)}(u, v) \). 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 $w_s(u, v) = \tilde{d}(u, v)$. For nodes $u, v \in S$, both $u$ and $v$ know $w_s(u, v)$.

4. Solve approximate SSSP on the skeleton graph $G_S$ with $s$ as the source, i.e. for each $v \in S$, compute $d'(s, v)$, where $d_{G_S}(s, v) \leq d'(s, v) \leq (1 + O(\epsilon))d_{G_S}(s, v)$.

5. For each $v \in V$, compute $\hat{d}(s, v) = \min_{u \in S} (d'(s, u) + \tilde{d}(u, v))$.

In steps 2 and 3 of the algorithm, I require that both nodes $u$ and $v$ know the distance estimate $\tilde{d}(u, v)$. FN does not have this requirement. Also, the distance estimates computed in step 2 should be consistent, meaning that the distance estimate $\tilde{d}(u, v)$ that $u$ knows should be equal to the distance estimate $\tilde{d}(u, v)$ that $v$ knows.

The correctness of the algorithm follows from Forster and Nanongkai\cite{17}.

**Theorem 4.3.1.** For any directed input graph $G = (V, E, w)$ with fixed source node $s$, the algorithm above consisting of Steps 1–5 computes, for every node $v \in V$, a distance estimate $\hat{d}(s, v)$ such that $d_G(s, v) \leq \hat{d}(s, v) \leq (1 + O(\epsilon))d_G(s, v)$.

In next subsection, I will bound the running time of the algorithm. I want to mention, the framework of Forster and Nanongkai\cite{17} works in the broadcast CONGEST model. To compute the cost of the algorithm, I only need to bound the running time of step 4.

### 4.3.1 Cost of the Algorithm

Steps 1, 3 and 5 are the same as Forster and Nanongkai’s algorithm\cite{17}. In Step 2, Forster and Nanongkai’s algorithm computes distance estimates from each skeleton node to each node in the original graph. The step 2 does the same computation and additionally computes the distance from each node in the original graph to each
skeleton node. The additional distances estimates are used in the computation of step 4.

The main difference in my algorithm is step 4. Both algorithms solve approximate SSSP on the skeleton graph, however I use a different technique from Forster and Nanongkai’s algorithm. I adapt the hopset algorithm in Chapter 3 to solve approximate SSSP on the skeleton graph. Theorem 3.5.6 constructs a \((n^{1/2+o(1)}/\rho, \epsilon)\)-hopset, and then solves parallel approximate shortest paths on the graph with the hopset edges added. For a directed graph \(G = (V, E)\), a \((\beta, \epsilon)\)-hopset is a set of weighted edges \(E'\) such that, for each pair of nodes \(u, v \in V\), there exists a path \(p\) from \(u\) to \(v\) that contains at most \(\beta\) edges and \(d_G(u, v) \leq d_{G'}(p) \leq (1 + \epsilon)d_G(u, v)\), where \(G' = (V, E \cup E')\). For step 4, I construct a \((\beta, \epsilon)\)-hopset using Algorithm 4 where \(\beta\) is a parameter I decide later. Then I simulate Klein and Subramanian’s hop-limited BFS algorithm for shortest paths [23] on the skeleton graph with the hopset edges added to the graph to solve approximate SSSP. Now I will describe computing shortest paths on the limited-depth skeleton graph. I give more details of how to simulate algorithm 4 in the CONGEST model in section 4.4.

One of the difficulties in computing SSSP on the skeleton graph is that an edge \((u, v) \in G_S\) may not be an edge in the original graph and thus not have a direct communication link. I will require that each node in the skeleton graph knows its incoming and outgoing edges. Once I have a limited depth skeleton graph, I can simulate BFS as follows.

**Lemma 4.3.2.** Given a graph \(G = (V, E)\) with diameter \(D\), a skeleton graph \(G_T\) over a subset of nodes \(T \subset V\) with integer weights and a source node \(s \in T\), there is an algorithm such that each node \(v \in R^+_h(G_T, s)\), including \(s\) itself, learns the distance \(d_{G_T}(s, v)\) in \(O(Dh + |R^+_h(G_T, s)|)\) rounds and \(O(|R^+_h(G_T, s)|)\) congestion on each edge in the CONGEST model.
Proof. I simulate BFS on the skeleton graph. The algorithm is divided into levels, and at each level $i$, the goal is for nodes at distance $i$ to learn their distance from $s$. To start, each node $v \in T \setminus \{s\}$ sets $d(s, v) = \infty$, and $s$ sets $d(s, s) = 0$. At level $i \in [0, h]$, if a node $v$ learns its distance $d(s, v) = i$, it will broadcast $d(s, v) = i$ to the whole graph. By Lemma 6.4.5, each level takes $O(D + K_i)$ time where $K_i = |R^+_i(G_T, s) \setminus R^+_{i-1}(G_T, s)|$. Thus, each node $v$ can learn its distance in $O(Dh + |R^+_h(G_T, s)|)$ rounds. After $h$ levels, all nodes whose distance have been updated broadcast their distances and $s$ learns the distance updates. This broadcast can be done in $O(D + |R^+_h(G_T, s)|)$ rounds. The total amount of information sent is $O(|R^+_h(G_T, s)|)$, and thus the algorithm takes $O(Dh + |R^+_h(G_T, s)|)$ rounds and $O(|R^+_h(G_T, s)|)$ congestion on each edge. 

Once I finish adding hopsets edges to the graph, I can use Lemma 4.3.2 bound the running time of Klein and Subramanian’s hop-limited BFS algorithm [23]. Assume that the hopsets diameter is $O(\alpha^{1/2+o(1)}/\rho)$, it takes $\tilde{O}((D\alpha^{1/2+o(1)}/\rho + \alpha) \log W/\epsilon)$ to simulate Klein and Subramanian’s algorithm based on Lemmam 4.3.2, where $\log W/\epsilon$ comes from the fact I need to run $O(\log_{(1+\epsilon)}(nW))$ layer of BFS in Klein and Subramanian’s algorithm [22]. The last peace is how to construct a $(\alpha^{1/2+o(1)}/\rho, \epsilon)$-hopsets in the distributed setting, I here give the running time and leave the detail in section 4.4.

**Theorem 4.3.3.** Given a graph $G = (V, E)$ with diameter $D$, and a $n$-node skeleton graph $G_T$ over a subset of nodes $T \subset V$, for a source node $s \in T$, there is an algorithm constructs $(\alpha^{1/2+o(1)}/\rho, \epsilon)$ hopsets and takes $\tilde{O}(D\alpha^{1/2+o(1)} \log W/(\rho \epsilon) + \alpha \rho^2 \log W/\epsilon^2)$ rounds w.h.p. in the CONGEST model.

Now I can put everything together to bound the running time.
Steps 1-3 and 5 can be performed the same as FN [17]. The complexity of the algorithm is as follows. 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 $\tilde{O}(\alpha + n \log W / (\alpha \epsilon) + D)$ rounds. FN only computes forwards distance estimates, but the backwards 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 Theorem 4.3.3, $(\alpha^{1/2+o(1)}/\rho, \epsilon)$-hopsets can be implemented in $\tilde{O}(\alpha^2 \log W + \frac{D \alpha^{1/2+o(1)}}{\rho \epsilon} \log W)$ rounds, where $\rho$ is a parameter which is in range $[\tilde{O}(1), \alpha^{1/2+o(1)}]$. Then it takes $\tilde{O}((D \alpha^{1/2+o(1)}/\rho + \alpha) \log W / \epsilon)$ to solve approximate SSSP by simulating Klein and Subramanian’s algorithm. Finally step 5 is computed internally for each node.

The total number of rounds for the algorithm is $\tilde{O}(D + \frac{n}{\alpha \epsilon} \log W + \frac{\alpha^2 \log W}{\epsilon^2} + \frac{D \alpha^{1/2+o(1)} \log W}{\rho \epsilon})$, which reduces to as follows:

**Case 1.** If $D = o(n^{1/4})$, set $\rho = \tilde{\Theta}(1)$ and $\alpha = \tilde{\Theta}(\sqrt{n})$. The whole algorithm takes $\tilde{O}(\sqrt{n} \log W / \epsilon^2)$ rounds.

**Case 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 $\tilde{O}(D \log W / \epsilon^2)$ rounds.

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

Combining all three cases together, the algorithm solves approximate SSSP in $\tilde{O}(\sqrt{n} + D + D^{2/5} n^{2/5+o(1)} \log W / \epsilon^2)$ rounds. This shows the running time of Theorem 4.1.1.

### 4.4 Implementation of Step 4 in CONGEST Model

The hopsets main algorithm is the same as Algorithm 4, except that I run Algorithm 5 when I call HSRECURSE(). This algorithm is essentially a simulation of the parallel
Algorithm 5 Distributed hopset algorithm for weighted directed skeleton graphs. \( \delta, k, \lambda, c, L \) are parameters.

1: function \( \text{HSITERATE}(\hat{G}) \)
2: \( S_0 = \{\hat{G}\}, \ H' \leftarrow \emptyset \)
3: for each \( r \in [0, \log_k n] \)
4: \( S_{r+1} = \emptyset, \ D_r \leftarrow D/(\lambda^r k^r/2) \)
5: for each \( G = (V, E) \in S_r \)
6: for each \( v \in V \) with \( \ell(v) = r + L \) \( \triangleright \) Run multiple source BFS on all \( \hat{G} \)
7: for each \( u \in R^+_{32\lambda^2 k^2 D_r \log^2 n}(\hat{G}, v) \) add edge \( (v, u) \) to \( H' \) with weight \( \text{dist}_{\hat{G}}(v, u) \)
8: 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 \( \text{dist}_{\hat{G}}(u, v) \)
9: for each \( v \in \hat{V} \) with \( \ell(v) = r \) \( \triangleright \) Construct next level skeleton graphs
10: Choose \( \sigma_v \) uniformly at random from \([1, 4\lambda^2 k^2 \log^2 n]\), set \( \rho_{\max} = 20\lambda^2 k^2 \log^2 n \)
11: for each \( u \in R^+_{\rho_{\max} D_r}(\hat{G}, v) \), add edge \( (v, u) \) with weight \( \text{dist}_{\hat{G}}(v, u) \)
12: for each \( u \in R^-_{\rho_{\max} D_r}(\hat{G}, v) \), add edge \( (v, u) \) with weight \( \text{dist}_{\hat{G}}(u, v) \)
13: Minimize \( |R_{(\rho_v+1)D_r}(\hat{G}, v) \setminus R_{(\rho_v-1)D_r}(\hat{G}, v)| \) such that \( \rho_v \in [16\lambda^2 k^2 \log^2 n + 4k(\sigma_v - 1), 16\lambda^2 k^2 \log^2 n + 4k\sigma_v) \)
14: for each \( u \in R^+_{\rho_v D_r}(\hat{G}, v) \) add label \( v^{\text{Des}} \) to vertex \( u \)
15: for each \( u \in R^-_{\rho_v D_r}(\hat{G}, v) \) add label \( v^{\text{Anc}} \) to vertex \( u \)
16: for each \( u \in R^+_{\rho_v D_r}(\hat{G}, v) \cap R^-_{\rho_v D_r}(\hat{G}, v) \) add label \( X \) to vertex \( u \)
17: \( V_{\text{fringe}} \leftarrow R_{(\rho_v+1)D_r}(\hat{G}, v) \setminus R_{(\rho_v-1)D_r}(\hat{G}, v) \)
18: \( S_{r+1} = S_{r+1} \cup \{\hat{G}[V_{\text{fringe}}]\} \triangleright \) If \( u \) in \( v \)’s fringe node, mark it
19: for each \( u \in V \) that has a \( X \) label, remove \( u \)
20: \( V_1, V_2, ..., V_t \leftarrow \text{partition based on labels} \)
21: for each \( i \in [1, t] \)
22: \( S_{r+1} = S_{r+1} \cup \{\hat{G}[V_i]\} \triangleright \) if \( u \) is in \( \hat{G}[V_i] \), mark it
23: return \( H' \)
hopset algorithm 2. The input of the algorithm is a skeleton graph, and the output of the algorithm is the skeleton graph with a \((n^{1/2+o(1)}/\rho, \epsilon)\) hopset added to the graph. The parameters \(\delta, k, \lambda, c, L, \rho\) are set the same as in Theorem 3.5.6. Choosing a larger \(L\) results in a smaller hopbound but larger size hopset. The correctness of the algorithm follows directly from Chapter 3, but I will show the running time of the algorithm. The remainder of this section will show the following theorem.

**Theorem 4.4.1.** Given a graph \(G = (V, E)\) with diameter \(D\), and a \(n\)-node skeleton graph \(G_T\) over a subset of nodes \(T \subset V\), for a source node \(s \in T\), Algorithm 4 with Algorithm 5 takes \(\tilde{O}(Dn^{1/2+o(1)} \log W/(k^{L/2} \epsilon) + nk^{L+1} \log W/\epsilon^2)\) rounds w.h.p. in the CONGEST model.

By setting \(k = O(\log n)\) and \(L = 17 - 2 \log_k \epsilon + 2 \log_k \rho\), for a \(\alpha\)-node graph, the bound from Theorem 4.4.1 is \(\tilde{O}(D\alpha^{1/2+o(1)} \log W/(\rho \epsilon) + \alpha^2 \log W/\epsilon^2)\), which is the bound in Theorem 4.3.3. To prove Theorem 4.4.1, I will use the following Lemma which is a standard result in the CONGEST model.

**Lemma 4.4.2.** [18] 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 I run all algorithms together. There is a distributed algorithm that can execute \(A_1, ..., A_k\) in \(\tilde{O}(\text{dilation} + \text{congestion})\) rounds in the CONGEST model.

Algorithm 4 first sets up the hopset \(H\) and the hopbound \(\beta\). It repeats Lines 4-18 \(\lambda \log^2 n\) times. Line 4 starts a loop that attempts to shortcut paths \(\hat{P}\) containing \(\beta\) to \(2\beta\) edges, where \(w(\hat{P}) \in [2^i, 2^{i+1})\). Next, in Lines 6-7, the algorithm rounds each edge up. After this step, a path \(\hat{P}\) will contain only integer weighted edges and the
new length of $\hat{P}$ is $\hat{w}(\hat{P})$ which is at most $O(\beta)$. Thus, when the algorithm later does BFS, the depth $\beta$ can be bounded.

Next the algorithm starts adding edges to the hopset. Lines 9-10 gives each node a label $l(v)$ and from Line 14 to 16, each node will do BFS with depth $\tilde{O}(\beta/\delta)$. Recall that Lemma 4.3.2 shows how to run single-source BFS. By combining Lemma 4.3.2 with Lemma 6.4.6, the algorithm can run BFS from multiple sources. Using these two Lemmas, the number of rounds for Lines 11-16 is $\tilde{O}(Dh + \sum_{v, \ell(v) \leq l} |R^+_h(G, s) \cup R^-_h(G, s)|)$, where $h = 8(1 + \epsilon)\beta/\delta = O(\beta/\epsilon)$. Since each node reached in a BFS adds an edge to the hopset, bounding the size of the hopset gives a bound on the number of nodes reached in all the searches. Fortunately, Based on Theorem 3.3.2, I can bound the size of the hopset, which gives an upper bound on the congestion of $\sum_{v, \ell(v) \leq l} |R^+_h(G, s) \cup R^-_h(G, s)| = \tilde{O}(nk^{L+1}/\epsilon^2)$.

Next, Algorithm 4 calls Algorithm 5. Algorithm 5 divides the graph into smaller graphs and adds edges on the smaller skeleton graphs. At the start of the algorithm 5, there is only one skeleton graph $\hat{G}$. Then in each iteration, the algorithm runs Lines 4-22 on each skeleton graph. Lines 6-8 are the same as Lines 14-16 of Algorithm 4, except for different nodes. Since the congestion is already bounded, the upper bound of the running time is the same. Lines 9-22 try to construct smaller skeleton graphs. For each pivot, i.e. nodes $v$ where $\ell(v) = r$, the algorithm first chooses a "good" distance, where "good" is defined in Line 9. In Line 9, each pivot runs BFS to depth $\rho_{\max}D_r$. Again, I can use the analysis in Chapter 3 to bound the congestion.

Lemma 4.4.3. Consider an execution of Line 9-14 HSITERATE($\hat{G}$). W.h.p. at least $1 - n^{-0.7l+3}$, the following holds for all $v \in \tilde{G}$,

$$|R^+_{\rho_{\max}D_r}(\tilde{G}, v)| \leq nk^{-r}, |R^-_{\rho_{\max}D_r}(\tilde{G}, v)| \leq nk^{-r}.$$
Lemma 4.4.3 implies that for each $r$, and each $\tilde{G}$, the congestion for each $v$ in Lines 19-20 is at most $O(nk^{-r})$. Then I can count the number of nodes in all skeleton graphs $\tilde{G}$ that run BFS. If I knew the total number of nodes for each $r$, I could count the number of nodes with label $\ell(v) = r$, since the probability of each node being label $\ell(v) = r$ is $\tilde{O}(k^r/n)$. The difficulty of counting the nodes in this way comes from the overlapping skeleton graphs, and thus the total number of nodes in all $\tilde{G} \in S_r$ could be quite large. In the following Lemma, Lemma 3.3.5 gives an upper bound of nodes in all $\tilde{G} \in S_r$ for each $r$.

**Corollary 4.4.4.** Consider an execution of Line 5-22 $\text{HSITERATE}(\hat{G})$. For all $r \in [0, \log k n]$, the number of nodes in all $\tilde{G} \in S_r$ is $\tilde{O}(n)$ with high probability.

Thus, the congestion for all pivots in Lines 10-12 is $\tilde{O}(n)$. This implies that multiple source BFS performed by all the pivots runs in $\tilde{O}(Dh + n)$ rounds. After $\rho_v$ is decided and broadcast to each node $u$, node $u$ adds appropriate label $v^{\text{Des}}, v^{\text{Anc}},$ or $X$ to itself. Lines 14-16 construct the fringe skeleton graph and Lines 17-19 partition the graph into smaller graphs. One last complication is that when the algorithm runs on the smaller skeleton graphs, the graphs may overlap with one another. The algorithm must distinguish different skeleton graph searches. The algorithm can overcome this issue by adding marks to each smaller skeleton graph. Also, if a node is in a fringe problem, it marks itself fringe and marks which fringe problem it is in. Likewise for the core problem, a node marks itself core and marks which labels it gets. When then algorithm runs BFS, each node must append its marks to the messages it sends out. Lemma 3.3.6 has the following bound on the size of the marks for each $r$.

**Corollary 4.4.5.** Consider an execution of Line 5-22 $\text{HSITERATE}(\hat{G})$. For all $v \in \tilde{V}, r \in [0, \log k n]$, w.h.p., the number of pivots $\ell(u) = r$, such that $v \in R(\rho_u+1)_{Dr}(\hat{G}, u)$ is $O(k)$.
Thus, the overhead for marks at each level is $O(k)$ and $\tilde{O}(k)$ for all $r$. Once I finish Algorithm 5, Lines 17-18 of Algorithm 4 add the hopsets edges back to the original graph $G$. This can be done easily since I can broadcast all hopset edges.

By combining all these steps together, Line 4-18 take $\tilde{O}(Dn^{1/2+O(1/\log k)}/(k^{L/2}\epsilon) + nk^{L+1}/\epsilon^2)$ rounds. There is an additional $\log W$ factor from guessing the weight of the path up to $n^2W$. This gives the running time in Theorem 4.4.1.

### 4.4.1 Extension to Broadcast CONGEST Model

Although the BFS in Lemma 4.3.2 works in the broadcast CONGEST model, I use Lemma 6.4.6 to combine BFS from different source. However, Lemma 6.4.6 doesn’t hold in the broadcast CONGEST model. To make Algorithm 4 work, I have to explain how to run multi-source BFS in broadcast CONGEST model. I give the following multi-source BFS in the broadcast CONGEST model.

**Lemma 4.4.6.** Given a graph $G = (V, E)$ with diameter $D$, and a set of skeleton graphs $T = \{G_{T_1}, G_{T_2}...,\}$ with integer edge weights, those graph nodes might overlap. Assuming that I can use $\tilde{O}(1)$ information to distinguish each skeleton graph $G_{T_i} \in T$. Now for each $G_{T_i}$, given a set $S_i \subset V(G_{T_i})$ of source nodes, there is an algorithm such that for each $S_i$ and $s \in S_i$, each node $v \in R_h^+(G_{T_i}, s)$ learns the distance $d_{G_{T_i}}(s, v)$ in $\tilde{O}(Dh + \sum_{G_{T_i} \in S} \sum_{s \in S_i} |R_h^+(G_{T_i}, s)|)$ rounds.

**Proof.** I simulate multi-source BFS on the skeleton graph. To start, for each skeleton graph $G_{T_i}$ and each source node $s \in S_i$, I set $d_{G_{T_i}}(s, s) = 0$ and $d_{G_{T_i}}(s, v) = \infty$ $v \in V(G_{T_i}) \setminus \{s\}$. At level $i \in [0, h]$, if a node $v \in V(G_{T_i})$ learns its distance $d(s, v) = i$, it will broadcast $d_{G_{T_i}}(s, v) = i$ to the whole graph. By Lemma 6.4.5, each level takes $O(D + K_i)$ time where $K_i = \sum_{G_{T_i} \in S} \sum_{s \in S_i} |R_i^+(G_{T_i}, s)\setminus R_{i-1}^+(G_{T_i}, s)|$. At the end of $h$ iteration, each node $v \in R_h^+(G_{T_i}, s)$ can learn its distance. Notice that to broadcast
the distance, each node has to append the skeleton graph identifier, which takes \(\tilde{O}(1)\) extra overheads. In total, it takes \(\tilde{O}(Dh + \sum_{T_i \in S} \sum_{s \in S_i} |R^+_h(G_{T_i}, s)|)\) rounds.

Lemma 4.4.6 implies that the number of nodes in \(\sum_{T_i \in S} \sum_{s \in S_i} |R^+_h(G_{T_i}, s)|\) decides the running time. Fortunately, I have bounded this number in CONGEST model, which is \(\tilde{O}(nK^{L+1}/\epsilon^2)\). This implies the following theorem.

**Corollary 4.4.7.** Given a graph \(G = (V, E)\) with diameter \(D\), and a \(n\)-node skeleton graph \(G_T\) over a subset of nodes \(T \subset V\), for a source node \(s \in T\), Algorithm 4 with Algorithm 5 takes \(\tilde{O}(Dn^{1/2+o(1/\log k)} \log W/(k^{L/2}\epsilon) + nk^{L+1} \log W/\epsilon^2)\) rounds w.h.p. in the broadcast CONGEST model.
Chapter 5

Exact Distance-Limited Single-Source Shortest Paths

5.1 Introduction

This section provides parallel algorithms for a distance-limited SSSP problem. For this variant of the problem, the edge weights are all nonnegative integers, but the problem is still nontrivial even if all weights are from \{0, 1\}. The goal is to return the correct shortest-path distance to all vertices having shortest-path distance \( \leq L \) from the source, where \( L \) is part of the input to the problem. Moreover, the algorithm should also identify which vertices have shortest-path distance strictly more than \( L \) from the source.

Section 5.2 presents my parallel algorithm for distance-limited SSSP with nonnegative integer weights. I solve the problem assuming \( L = O(n) \), as otherwise parallel Dijkstra’s [4, 12] is more efficient. My algorithm has \( \tilde{O}(m) \) work and \( \sqrt{L} \cdot n^{1/2 + o(1)} \) span, with high probability.

I note that the main difficulty in this problem arises from the presence of 0-weight edges mixed with positive weights, as paths through 0-weight edges do not add to the distance. Without the 0s, it is not too hard to solve the problem even more efficiently using a generalization of parallel BFS.
5.2 Distance-Limited SSSP with Non-negative Integer Weights

Given a directed graph $G = (V, E)$ with source $s \in V$, nonnegative integer edge weights, and distance $L \leq n$, the problem is to return the shortest-path distances $d(v) = \text{dist}(s, v)$ for all $v$ with $\text{dist}(s, v) \leq L$, and $d(v) = \infty$ otherwise. I also want to report a shortest-paths tree to recover shortest paths, which can be accomplished through a postprocessing step discussed in Section 5.2.3.

5.2.1 Algorithm Overview

Similar to the peeling algorithm in the previous section, this algorithm will solve shortest paths in order by distance and peel off solved nodes. Once a node is finalized, its distance is set and not considered for the rest of the algorithm. The algorithm will use a $(1 + \epsilon)$-approximate shortest paths algorithm to guess the distances of nodes in the graph. Using the distance estimate, each unfinalized node $v$ is assigned to a $2^i$-sized interval for some integer $i$. As long as the ASSSP never returns an incorrect answer, the true shortest-path distance to $v$ always falls within its assigned interval. Section 5.2.3 discusses how to cope with the possibility that the ASSSP algorithm fails to achieve the approximation. After each layer of the peeling, the algorithm refines the distance intervals, by again running ASSSP. Since the shortest paths have gotten shorter, the approximation becomes better, and so the node can be assigned to an interval of smaller length. Once the interval size is a small enough constant, the distance can be solved directly.

It is too expensive to refine all distance estimates in each round, so the algorithm must choose when to include each node.
5.2.2 Algorithm Description

Let $D$ be the smallest power of 2 strictly greater than $L$. The algorithm operates on intervals $[d, d + 2^i)$ of length $2^i$, for $0 \leq i \leq \lg(2D)$, where the intervals are aligned to multiples of $2^{i-1}$, i.e., $d = k2^{i-1}$. Unfinished nodes are assigned to intervals that (baring ASSSP failures) contain their true distance, and they are moved to smaller intervals as the algorithm progresses. Initially the algorithm runs 2-approximate ASSSP; all nodes with estimate $> 2D$ (and hence true distance $> D$) are finalized to a distance of $\infty$; all other nodes are assigned to $[0, 2D)$.

The rest of the algorithm proceeds in rounds $0, 1, \ldots, D$, where nodes with distance $d$ are finalized during round $d$. In each round, the value $d$ encroaches (reaches the left side) of some intervals. For each of these intervals $I = [d, d + 2^i)$, from largest to smallest, the goal is to “refine” the distance estimates of all nodes assigned to $I$ to smaller intervals or finalize nodes with size-1 intervals.

I compute the shortest-path tree as straightforward a postprocessing step in Section 5.2.3.

Refine

The function \texttt{Refine}(d, 2^i) takes as input the interval $[d, d + 2^i)$. It builds a graph $G'$ on nodes whose interval overlaps $[d, d + 2^i)$. More details of building the graph are discussed next. The key idea is that since all unfinalized nodes have distance $d$, I can shift distances downward by $d$; i.e., distances of $d$ in $G$ are translated to distances of 0 in $G'$; roughly speaking, this allows us to apply an algorithm with multiplicative approximation to improve the additive approximation. The next step is to run ASSSP on $G'$, which improves the distance estimates for those nodes assigned to $[0, d + 2^i)$. 

81
Those nodes are reassigned to one of the 3 overlapping subintervals of size $2^{i-1}$, as shown in Lines 14-20 of Algorithm 6.

**Build Graph $G'$**

Add each node to the $V'$ who's interval overlaps $[d, d + 2^{i})$. Create graph $G' = G[V']$. Add a source node $s'$ to $V'$. For each node $v$ with $d(v) = +\infty$, and each incoming edge $(u, v)$, where $d(u)$ is not infinity, add an edge from $s'$ to $v$ with weight $d(u) + w(u, v) - d$.

**5.2.3 Verification and Shortest Paths Tree**

The algorithm for parallel approximate shortest paths works with high probability, meaning that it never gives an underestimate of the shortest paths, but may fail to achieve the $(1 + \epsilon)$ approximation. In this section, I will show how to verify that my algorithm is correct. After running the algorithm, first contract cycles 0-weight edges, and then look at each nodes incoming edges. For each node $v \in V$, verify that $d(v) = \min_{(u,v)}(d(u) + w(u, v))$, and if any node fails then the algorithm has failed and must be repeated.

**Lemma 5.2.1.** Assume I set source node $d(s) = 0$, then for all $v \in V$, $d(v) = \text{dist}(s, v)$ if and only if for all $v$, I have $d(v) = \min_{(u,v)}(d(u) + w(u, v))$.

**Proof.** ($\Leftarrow$) Based on the definition of shortest path, for any $v$, $d(v) = \text{dist}(v) = \min_{(u,v)}(\text{dist}(s, u) + w(u, v)) = \min_{(u,v)}(d(u) + w(u, v))$.

($\Rightarrow$) Proof by contradiction. I first want to show if $d(v) = \min_{(u,v)}(d(u) + w(u, v))$ for $v$ and $d(s) = 0$, then $d(v) \geq \text{dist}(v)$. Consider the path $(v_0, v_1, ..., v_t = v)$, such that $d(v_i) = d(v_{i-1}) + w(v_{i-1}, v_i)$. Each edge $w(v_{i-1}, v_i) > 0$ is in the graph, so $d(v_0)$ to $d(v_t)$ is increasing and I can set $v_0 = s$. Notice that $d(v_t) = \sum w(v_{i-1}, v_i)$ is weight of a path from $s$ to $v$, so $d(v) \geq \text{dist}(v)$. 

82
Let $S$ be set of node $x$ with $d(x) \neq \text{dist}(s, x)$. Let $v$ the the node in $S$ with smallest $\text{dist}(v)$. Let $u$ be the parent of $v$ in the shortest path tree. Since $w(u, v) > 0$, I have $\text{dist}(u) < \text{dist}(v)$ and $d(u) = \text{dist}(u)$. Then, $d(v) \leq d(u) + w(u, v) = \text{dist}(u) + w(u, v) \leq \text{dist}(v).$ and this contradicts to the fact that $d(v) \geq \text{dist}(v)$ and the definition of $S$.

**Shortest Paths Tree**

I would like to output the shortest path tree for the contracted graph (zero weight cycles are contracted). The algorithm can be extended to find pointers for all nodes in the graph. I follow a similar strategy as the verification algorithm. First contract any zero weight cycles, and then each node looks at its incoming edges to find its parent in the tree. Specifically, for each node $v$, set $\text{parent}(v)$ to be $u$ such that $d(v) = d(u) + w(u, v)$.

**5.2.4 Limited Distance Shortest Paths Analysis**

Next I will show that the algorithm correctly computes the shortest paths distances for nodes up to distance $D$. The algorithm for approximate shortest paths that I use as a black box achieves the approximation with high probability. In the following lemma I assume the approximate shortest paths runs successfully, since in Section 5.2.3 I showed how to overcome the issue that approximate shortest paths does not achieve the desired approximation.

**Lemma 5.2.2.** Consider a call to $\text{LIMITEDSP}(G)$ for some graph $G = (V, E)$. For any node $v \in V$ with $d(v) \leq D$, the shortest path distance is correctly computed.
Algorithm 6 LimitedSP
Input: Graph $G = (V, E)$ and source node $s$
Output: Shortest path distance $d(v)$ for each node $s$ from $v$ up to distance $L$, and $\infty$ for nodes with distance greater than $L$

1: function LimitedSP($G = (V, E), s$)
2: for each $v \in V$ $d(v) \leftarrow +\infty$
3: $d(s) = 0$
4: $d' \leftarrow \text{ASSSP}(G, s, \epsilon = 1)$
5: for each $v \in V$ with $d'(v) \leq 2D$ add $v$ to interval $[0, 2D)$
6: for $d = 0$ to $D$:
7: for $i = \lg(2D)$ to 0:
8: if $d$ is a multiple of $2^{i-1}$ then
9: Refine($d, 2^i$)
10: function Refine($d, 2^i$)
11: Build graph $G' = (V', E')$
12: $d' \leftarrow \text{ASSSP}(G', s', \epsilon < 1/4)$
13: for each $v \in V'$ with $d'(v) = 0$ $d(v) \leftarrow d$
14: for each $v \in V'$ with $I(v) = [d, d + 2^i)$
15: if $d'(v) \in [0, 2^{i-1})$ then
16: add $v$ to interval $[d, d + 2^{i-1})$
17: else if $d'(v) \in [2^{i-1}, 3 \cdot 2^{i-2})$ then
18: add $v$ to interval $[d + 2^{i-2}, d + 3 \cdot 2^{i-2})$
19: else
20: add $v$ to interval $[d + 2^{i-1}, d + 2^i)$

Proof. For any given node $v$, recall $\text{dist}(s, v)$ is the shortest path distance. Let $I(v)$ be the interval $v$ is located in. I change $I(v)$ if and only if $I(v) = [d, d + 2^i)$ when I call Refine($d, 2^i$). I will use induction on Refine($d, 2^i$) on $d$ and $i$ to show

- for any node $v$ with $\text{dist}(s, v) < d$, $d(v) = \text{dist}(s, v)$
- for any node $v$ with $\text{dist}(s, v) \leq D$, $\text{dist}(s, v) \in I(v)$.

The base case is $d = 0$ and $i = \lg(2D)$. For any node $v$ with $\text{dist}(s, v) \leq D$, $I(v) = [0, 2D)$, I run approximate shortest paths on the whole graph. In this case,
\(d'(v) = 0\) if and only if \(\text{dist}(s, v) = 0\). I set \(d(v) = \text{dist}(s, v) = 0\) if \(d'(v) = 0\). For any node \(v\) with \(\text{dist}(s, v) \leq D\), if \(d'(v) < 2^{i-1}\), I know that \(\text{dist}(s, v) \leq d'(v) < 2^{i-1}\); if \(2^{i-1} \leq d'(v) < 3 \cdot 2^{i-2}\), I know that \(\text{dist}(s, v) \leq d'(v) \leq 3 \cdot 2^{i-2}\) and \(\text{dist}(s, v) \geq d'(v)/(1 + \epsilon) \geq 2^{i-2}\); if \(d'(v) \geq 3 \cdot 2^{i-2}\), I know that \(\text{dist}(s, v) \geq d'(v)/(1 + \epsilon) \geq 2^{i-1}\).

Combining these three cases, node \(v\) will be added to \(I(v)\) such that \(\text{dist}(s, v) \in I(v)\).

Now assume that for \(d\) and \(i\), the claim holds, consider \(d\) and \(i - 1\), for the first claim, for any node with \(\text{dist}(s, v) < d\), \(d(v) = \text{dist}(s, v)\). For the second claim, for any node \(v\) with \(\text{dist}(s, v) \leq D\), \(\text{dist}(s, v) \in I(v)\). When I call \text{REFINE}(d, 2^{i-1})\), I change \(I(v)\) if and only if \(I(v) = [d, d + 2^{i-1}]\). Consider a node \(v\) with \(I(v) = [d, d + 2^{i-1}]\) and a path \(p\) from \(s\) to \(v\). Let \(u\) be the last node on the path with \(\text{dist}(s, u) < d\) and \(u'\) be the next node on this path. I will add an edge from \(s'\) to \(u'\) with weight \(d(u) + w(u, u') - d\). Let \(l\) be the weight of path from \(u'\) to \(v\). Notice that weight of \(p'\) is \(w(p') = d(u) + w(u, u') - d + l \geq \text{dist}(s, v) - d\). When \(p\) is the shortest path, for any node \(w\) on path \(p\) between \(u'\) and \(v\), I know \(d \leq \text{dist}(s, w) \leq \text{dist}(s, v) < d + 2^{i}\). Thus \(I(w)\) overlaps \([d, d + 2^{i}]\) and I add \(w\) to the graph, and then \(w(p) = \text{dist}(s, v) - d\).

Combining these two points, I know in the new graph, the shortest path distance for \(v\) is \(\text{dist}(s, v) - d \in [0, 2^{i}]\). If \(d'(v) \leq 2^{i-1}\), then I know \(\text{dist}(s, v) - d \leq d'(v) \leq 2^{i-1}\) and \(\text{dist}(s, v) \in [d, d + 2^{i-1}]\); If \(2^{i-1} \leq d'(v) \leq 3 \cdot 2^{i-2}\), I know \(\text{dist}(s, v) - d \geq d'(v)/(1 + \epsilon) \geq 2^{i-2}\) and \(\text{dist}(s, v) - d \leq d'(v) \leq 3 \cdot 2^{i-2}\); Last, if \(d'(v) \geq 3 \cdot 2^{i-2}\), I know that \(\text{dist}(s, v) - d \geq d'(v)/(1 + \epsilon) \geq 2^{i-1}\). In either case, I have \(\text{dist}(s, v) \in I(v)\).

The last thing is to show for any \(d\) and \(i = 0\), I set \(d(v) = d\) if \(\text{dist}(s, v) = d\). Notice that based on the assumption, for any node \(u\) with \(\text{dist}(s, u) \leq D\), \(\text{dist}(s, u) \in I(u)\).

If \(\text{dist}(s, v) = d\), then I add \(v\) to the graph. Similar to the above argument, in the new graph, the shortest path distance for \(v\) is 0 and thus \(d'(v) = 0\) and I will set \(d(v)\) to be \(d\) for any node \(v\) with \(\text{dist}(s, v) = d\). \(\Box\)
Next I will show the work and span the of the algorithm. Each call to \textsc{Refine}(d, 2^i) builds a graph and runs approximate shortest paths. The main goal is to show that each node is not added to too many of these graphs. By bounding the number of graphs a node is added to, I am able to bound the total size of these graphs.

\textbf{Lemma 5.2.3.} Consider a call to \textsc{LimitedSP}(G) for some graph \( G = (V, E) \). For a node \( v \in V \), while \( v \) is assigned to a particular interval \( X \), \( v \) is added to \( O(\lg D) \) graphs \( G' \) in \textsc{Refine}.

\textit{Proof.} Node \( v \) is only added to a graph \( G' \) in \textsc{Refine}(d, 2^i) when interval \( X \) overlaps the interval \([d, d + 2^i]\). Let the interval \( X = [c2^j, (c+1)2^j] \) have length \( 2^j \), for some integer \( c \). Consider the following two cases of calls to \textsc{Refine}(d, 2^i) for size \( 2^i \) intervals. The two cases are \( j \leq i \), and \( j > i \). In both cases I will show at most three intervals of size \( 2^i \) overlap \( X \), which implies \( X \) overlaps at most \( O(\lg D) \) intervals, and \( v \) is added to at most \( O(\lg D) \) graphs in calls to \textsc{Refine}(d, 2^i).

\textbf{Case 1:} \( j \leq i \). In this case, there will be at most three intervals that intersects \( X \), i.e., \([d, 2^i]\), \([d + 2^{i-1}, 2^i]\) and \([d + 2^i, 2^i]\) since \( j \leq i \) and interval \([d, d + 2^{i+1}] \) must cover \( X \) if \([d, 2^i]\) intersects \( X \).

\textbf{Case 2:} \( j > i \). For this case I will show that at most one interval of size \( 2^i \) overlaps \( X \). Consider the first \( d \) such that \textsc{Refine}(d, 2^i) intersects with \( X = [c2^j, (c+1)2^j] \), I have \( d \leq c2^j \). That’s because when I refine \textsc{Refine}(d, 2^i), it’s impossible such that for some \( u \) with interval \( I(u) = [c2^j, c2^j + 2^j] \), \( c2^j < d \). If such node \( u \) exists, notice that when I update interval in \textsc{Refine}(d', 2^i), I never put a node to some interval starting earlier than \( d' \), so \( I(u) = [c2^j, c2^j + 2^j] \) must be updated in some \textsc{Refine}(a', 2^{j+1}) with \( a' \leq c2^j \). Then when I call \textsc{Refine}(c2^j, 2^j), I will refine node \( u \) and it’s no longer in \( I(u) \).
Assume that the first time \( \text{REFINE}(d, 2^i) \) intersects with \( X \), I have \( d \leq c2^j \). Since \( 2^{i-1} < 2^{j-1} \), and both are powers of 2, each multiple of \( 2^{j-1} \) is also a multiple of \( 2^{i-1} \). Therefore there is a size \( 2^i \) interval starting at \( c2^{j-1} \). This interval has size \( 2^i \), and any that start later will be refined after \( X \) since the intervals are refined in order by distance and decreasing size. For intervals that start before \( X \), there is one size \( 2^i \) interval that starts at \( c2^{i-1} - 2^{i-1} \) which overlaps \( X \). The next size \( 2^i \) interval to the left starts at \( c2^{i-1} - 2 \cdot 2^{i-1} = c2^{j-1} - 2^i \), and since it has length \( 2^i \), the interval is \([c2^{j-1} - 2^i, c2^{j-1}]\), so it does not overlap \( X \). Any intervals starting at smaller multiples of \( 2^{i-1} \) also do not overlap \( X \), by the same reasoning. Thus \( X \) overlaps one interval of size \( 2^i \) for each \( i < j \).

**Lemma 5.2.4.** Consider a call to \( \text{LIMITEDSP}(G) \) for some graph \( G = (V, E) \). For a node \( v \in V \), \( v \) is added to \( O(\lg^2 D) \) graphs \( G' \) in calls to \( \text{REFINE}(d, 2^i) \).

**Proof.** If node \( v \) has its initial distance estimate \( d(v) > 2D \), then \( v \) is not added to any intervals and therefore no graphs \( G' \) either. Otherwise, \( v \) starts in the interval \([0, 2D]\). Each time it is added to a graph in Refine, either it stays in the same interval or moves to a smaller interval. By Lemma 5.2.3, \( v \) gets added to \( O(\lg D) \) refinement graphs for each interval it is in. Since the interval sizes are monotonically decreasing, and there are \( O(\lg D) \) of them, \( v \) is added to \( O(\lg^2 D) \) refinement graphs.

I next turn to the work and span of the algorithm. Some of this relies on the straightforward parallel details, but the bulk of the work falls in ASSSP. I can leverage Lemma 5.2.4 to argue that the in total, these calls are not too expensive.

Before proving the work and span, I will show a data structure that I will use in the implementation.
Parallel sets. There exist implementations of parallel sets that 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 \( n \geq m \) \[3\]. The parallel set can also enumerate all elements in a size \( n \) set in \( O(n) \) work and \( O(\log n) \) span \[3\].

Vector of parallel sets. I construct a vector of parallel sets \( VS \), where each set has an identifier. Assume that there are polynomial bounded elements in total across all sets. The vector of sets \( VS \) supports the following.

- Initialization, I can set up a vector, each item of the vector contains a pointer to the set. The set is identified by the identifier. All sets are set to be empty at the beginning, the work is \( O(\text{number of sets}) \) and the span is \( \tilde{O}(1) \).

- Given \( t \) sets, where each set contains \( x_i \) elements, I can add the elements of the \( t \) sets into \( VS \) in \( O(\sum x_i) \) work and \( \tilde{O}(1) \) span because I can merge each set separately.

- Given the identifiers of \( t \) sets, I can merge the elements from all \( t \) sets into a vector with \( O(x) \) work and \( \tilde{O}(1) \) span, where \( x \) is the total number of elements across the \( t \) sets. To copy all elements, I first compute the number of elements in each set, then I run prefix sum to compute the location of each set should be transferred and access each sets elements in parallel.

- Given the identifiers of \( t \) sets, I can empty all elements in those \( t \) sets in \( O(x) \) work and \( \tilde{O}(1) \) span, where \( x \) is the total number of elements across the \( t \) sets.

Note that, to manipulate on the vector of parallel sets, I must know the identifier of each set to locate the pointer to the set.

Now I can show the following lemma which bounds the work and span of the algorithm.
Lemma 5.2.5. Algorithm 6 has $\tilde{O}(m)$ work and $n^{1/2+o(1)}L^{1/2}$ span.

Proof. Algorithm 6 first runs the approximate shortest paths algorithm which takes $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span. Next, it assigns each node with to an interval which can be done in $O(n)$ work and $O(1)$ span. Last, it calls $\text{Refine}(d, 2^i) \tilde{O}(D)$ times.

The subroutine $\text{Refine}(d, 2^i)$ first identifies all nodes whose interval intersects with $[d, d+2^i)$. Let $x_i$ and $y_i$ be number of nodes in the overlapping intervals and number of edges neighboring the nodes. I will show later that the algorithm copies all nodes in $\tilde{O}(x_i + 2^i)$ work and $\tilde{O}(1)$ span. After identifying those nodes, I can construct the graph with $\tilde{O}(x_i + y_i)$ work and $\tilde{O}(1)$ span. The approximate shortest path algorithm also depends on $x_i$ and $y_i$, and takes $\tilde{O}(y_i)$ work and $x_i^{1/2+o(1)}$ span. Lastly, each node in the interval $[d, d+2^i)$, will be reassigned to a different interval. To reassign the nodes, I can sort the nodes based on $d'$ value. Sorting takes $\tilde{O}(x_i)$ work and $\tilde{O}(1)$ span. Moving the nodes to different intervals relies on the implementation of the sets. I will later show this step can be performed in $\tilde{O}(x_i)$ work and $\tilde{O}(1)$ span. In total, each call to $\text{Refine}(d, 2^i)$ takes $\tilde{O}(y_i + 2^i)$ work and $x_i^{1/2+o(1)}$ span, where $x_i$ and $y_i$ are the number of nodes and edges, respectively, in $G'$.

Notice that for each $(d, 2^i)$, I have $d/2^i$ different calls to $\text{Refine}$ with interval size $2^i$, so the $2^i$ term will contribute $\tilde{O}(D)$ to the work in total. By Lemma 5.2.4, each node is added to $G'$ at most $O(\lg^2 D)$ times, so $\sum x_i = O(n \lg^2 D) = O(n \lg^2 D)$, and $\sum y_i = O(m \lg^2 D + L) = \tilde{O}(m + L) = \tilde{O}(m)$ because of the fact that $L = O(n)$. The total work is $\tilde{O}(m + D) = \tilde{O}(m)$. Notice there are at most $O(D \lg D)$ calls to $\text{Refine}$, and the span of the algorithm is $\sum x_i^{1/2+o(1)} = n^{1/2+o(1)}L^{1/2+o(1)} + L = n^{1/2+o(1)}L^{1/2}$.

The remaining problem is to maintain a set for each interval. I will use the vector of parallel sets data structure to maintain a vector of sets for each possible interval $[d, d+2^i)$. Notice that the identifier for each set is the interval $[d, d+2^i)$ for different $d$ and $i$ and the identifier for each set can be sorted by the starting point of the interval.
If the starting point is the same, I sort by ending point of the interval. Initialization takes $\tilde{O}(D)$ work and $\tilde{O}(1)$ span. Then I add all elements to the interval $[0, 2D)$. It takes $\tilde{O}(1)$ work and span to specify the identifier of the set and $\tilde{O}(n)$ work and $\tilde{O}(1)$ span to add all elements.

In a call to $\text{Refine}(d, 2^i)$, since the interval is sorted by starting point, the interval that intersects with $[d, d + 2^i)$ and starts no earlier than $d$ is continuous and I can identify all identifiers of intervals in $\tilde{O}(2^i)$ work and $\tilde{O}(1)$ span. Once I get all the interval that intersects with $[d, d + 2^i)$, it takes $O(|V'|)$ work and $\tilde{O}(1)$ span to copy all elements in the sets. After I identify all nodes in $G'$, the graph can be built by using parallel sort to group together the nodes that are in the subgraph. The last piece is to empty all elements for interval $[d, d + 2^i)$ and add those elements to three different intervals. This can be done in $O(|V'|)$ and $\tilde{O}(1)$ span.

From Lemmas 5.2.2 and 5.2.5, I conclude:

**Theorem 5.2.6.** There exists a parallel algorithm solving nonnegative $L$-distance-limited SSSP with work $\tilde{O}(m)$ and span $n^{1/2+o(1)}L^{1/2}$ span, with high probability. That is, consider a directed graph $G = (V, E)$ with nonnegative integer weights and source $s \in V$. The algorithm outputs for each node $v \in V$, $d(v) = \dist(s, v)$ if $\dist(s, v) \leq L$, and infinity otherwise.
6.1 Introduction

6.1.1 Results

My results leverages the breakthroughs of hopsets in Chapter 3 and a simple but powerful rounding technique, iterative gradual rounding [7]. I say that a graph is a \((h, \epsilon)\)-hopset graph if for any pair of nodes \(u, v\), \(\text{dist}(u, v) \leq \text{dist}^h(u, v) \leq (1+\epsilon)\text{dist}(u, v)\), where \(\text{dist}(u, v)\) is the shortest path distance and \(\text{dist}^h(u, v)\) 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.

Klein and Subramanian [24] provide an algorithm 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 \(v \in V\), \(\text{dist}(v)/2 \leq \tilde{d}(v) \leq \text{dist}(v)\), (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.

Recently, Chechik and Mukhtar [7] proposed an algorithm that computes the distance estimate with a relaxed version of the triangle inequality in the CONGEST
model. Using their iterative gradual rounding technique in a different way, I can efficiently compute the distance estimate for a hopset graph. My main result is as follows.

**Theorem 6.1.1.** Given a \((h, \epsilon = \frac{1}{4\log(2nK)})\)-hopset graph \(G\) with non-negative integer weight from \(\{0, 1, 2, ...K\}\), there is a deterministic algorithm computing the above distance estimate for exact SSSP with \(\tilde{O}(m \log K)\) work and \(\tilde{O}(h \log^2 K)\) span.

This result forms the basis for my improved parallel SSSP algorithms. Theorem 3.5.3 give an efficient construction for \((h = n^{1/2+o(1)}, \epsilon)\)-hopset. I are able to solve the parallel exact SSSP algorithm efficiently.

**Theorem 6.1.2.** 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 paths problem with \(\tilde{O}(m \log W)\) work and \(n^{1/2+o(1)} \log W\) span with high probability.

I can also apply Theorem 6.1.1 in the distributed model. My main result for distributed SSSP algorithms is given by Theorem 6.1.3. Notice that there is a known lower bound of \(\Omega(\sqrt{n} + D)[27]\), and my algorithm matches the lower bound when \(D = o(n^{1/4})\).

**Theorem 6.1.3.** 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.
6.1.2 Overview of the Technique

Here I provide an overview of the approach and give a high-level idea of why my approach works. Throughout this section, I refer to the input directed graph \( G = (V, E, w) \) and a source node \( s \).

As a minor deviation from Klein and Subramanian [24], I use overestimates instead of underestimates. That is, the goal is to produce a real-weight distance estimate \( \hat{d}(v) \) for each node \( v \in V \), such that (a) for each \( v \in V \), \( \text{dist}(v) \leq \hat{d}(v) \leq 2\text{dist}(v) \), (b) for each \( (u, v) \in E \), \( \hat{d}(u) + 2w(u, v) \geq \hat{d}(v) \). The overestimate can be used to construct the underestimate, whereas the overestimate is more familiar in the context of the approximate shortest paths. In fact, any solution to approximate single-source shortest-paths gives us the property specified by inequality (a). The main challenge of the algorithm is to produce a solution that also satisfies the relaxed triangle inequality of (b).

Main Technique

To convey the main idea for 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 \( d'(u) \geq \text{dist}(u) \) 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, I find it convenient to reason about weights of paths rather than single edges, and thus the 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 \( \hat{d}(v) = \min_{u \in V, u \leadsto v} (d'(u) + w(u \leadsto v)) \) for each node \( v \), where \( u \leadsto v \) is an arbitrary path. Notice that this formula general-
izes 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, I 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 round $i$, I round-up the edge weights $w(e)$ to $w_i(e)$ and apply the update rule $\hat{d}(v) = \min_{u \in V, \text{paths } u \rightarrow v} (d'(u) + w_i(u \rightarrow v))$. I 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, $d'(u) \geq \text{dist}(u)$ for each node $u \in V$ and $w_1(e) \in [w(e), (1 + \epsilon)w(e)]$, the final $\hat{d}(u)$ will satisfy both properties of the distance estimate.

By carefully designing how to round edges, I can make it so updating the current round estimate $\hat{d}$ given the previous round estimate $d'$ can be done efficiently in parallel. My rounding method basically follows the iterative gradual rounding method [7]. Here, I give a more natural formula to show the high-level idea. The 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 $i$ ($i$ from $\log N$ down to 1), I use the following weights for each $e \in E$:

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

Note that in the last round $i = 1$, for any $h = \Omega(1)$, I have $w(e) \leq w_1(e) \leq 2w(e)$, and the distance estimate can be used to compute the SSSP.

The first term $\Theta\left(\frac{2^i}{h \log N}\right)$ 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 \( w_i(u \leadsto v) \), and this algorithm is efficient as long as the number of hops in the path is small. That is, I 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 I prove is that when \( w_i(u \leadsto v) > \Omega(2^i) \), the estimate achieved in round \((i + 1)\) is already better than any potential update performed in the next round \( i \). (Recall that I am 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 I am operating on a hopset graph. In particular, this term increases in each subsequent round. 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, I am able to argue that for very 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. It does not. While I 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 by using a directed hopset instead of an undirected one.
Difference from Chechik and Mukhtar[7]

My algorithm follows the general blueprint of Chechik and Mukhtar[7], but I highlight the following differences:

- **Limited distance search vs. unlimited search.** Although in the implementation, both my algorithm and Chechik and Mukhtar’s algorithm update distances with distance-limited searches in the graph, Chechik and Mukhtar’s 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 break an edge. Chechik and Mukhtar show a different form of the triangle inequality than I do; theirs is distance limited, which makes it more complicated. I instead analyze the algorithm with respect to an unbounded search, and then I 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.** My algorithm and analysis explicitly leverage hopsets, and I use the hopset construction as a black box. Not only does this reduce the complexity of the algorithm and proof, but it also means that my 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[23], 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.
6.2 The High-level Algorithm

My exact SSSP algorithm is an efficient implementation of an algorithm which I present in this section. This section presents the algorithm at a high level and does not consider implementation details. Instead, it specifies the conditions that must be met when executing each step. I first show that producing distance overestimates with the triangle inequality is sufficient to solve exact SSSP. I next give an algorithm using iterative gradual rounding that, in effect, reduces the problem of producing a valid distance estimate to the problem of producing a hopset.

Exact SSSP algorithm Once I have the above reduction, to solve the exact SSSP for digraphs with edge weight from \( \{0, 1, ..., W\} \), I can transfer a general graph to a hopset graph using hopsets in Chapter 3 and 4, compute the distance estimate and repeat the whole process \( \log W \) times.

6.2.1 Reduce Exact SSSP to Distance Estimate

Given a directed graph \( G = (V, E, w) \) with non-negative integer weights from \( \{0, 1, 2, ..., K\} \) and a source node \( s \), the goal is to compute a real distance estimate \( \hat{d}(v) \) for each node \( v \) such that

\[
\begin{align*}
\text{(6.2.1.1)} & \quad \text{for } v \in V, \quad \text{dist}(v) \leq \hat{d}(v) \leq 2\text{dist}(v), \\
\text{(6.2.1.2)} & \quad \text{for } (u, v) \in E, \quad \hat{d}(u) + 2w(u, v) \geq \hat{d}(v).
\end{align*}
\]

Once I compute \( \hat{d}(v) \), I set \( \tilde{d}(v) = \left\lfloor \hat{d}(v)/2 \right\rfloor \). Note that \( \text{dist}(v) \) is an integer, I have

- for \( v \in V, \ \tilde{d}(v) \leq \left\lfloor \text{dist}(v)/2 \right\rfloor = \text{dist}(v), \ \tilde{d}(v) \geq \left\lceil \text{dist}(v)/2 \right\rceil \geq \text{dist}(v)/2, \\
- for \ (u, v) \in E, \ \tilde{d}(u) + w(u, v) = \left\lfloor \hat{d}(u)/2 \right\rfloor + w(u, v) = \left\lfloor \hat{d}(u)/2 + w(u, v) \right\rfloor \geq \left\lfloor \hat{d}(v)/2 \right\rfloor = \tilde{d}(v).
The new distance estimate \( \tilde{d} \) is sufficient to solve the exact SSSP using the reduction of Klein and Subramanian [24]. I want to mention that in the original graph, the maximum edge weight is \( W \), but when I compute the distance estimate, the maximum edge weight in the graph is \( K \) and I can assume \( K = O(n) \) (see Lemma 4.1 in Klein and Subramanian [24] or Theorem 2.1 in Forster and Nanongkai [17]).

6.2.2 Reduce Distance Estimate to Hopsets

Given a graph \( G = (V, E, w) \), assume that \( G \) is a \((h, \epsilon)\)-hopset graph, that is, for any pair of nodes \( u, v \), \( \text{dist}(u, v) \leq \text{dist}^{(h)}(u, v) \leq (1 + \epsilon) \text{dist}(u, v) \). \( \epsilon \) is a small value which will be determined later. Now I give distance estimate algorithm for hopset graphs in Algorithm 7.

Algorithm 7 runs in \( \beta + 1 \) rounds, where the last round (lines 8-11) is designed for 0-weight edges. At first, 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. In each round \( i \), I will round up edges and estimate the distance \( d_i \) based on the estimate of the last round \( d_{i+1} \). To update \( d_i(v) \), I choose the node that minimizes \( d_{i+1}(u) \) plus the weight of an arbitrary path from \( u \) to \( v \) with respect to the weight function \( w_i \). I will explain how to implement the updates later. Last, in round \( (i = 0) \), I only consider the zero-weight edges in the graph. I set \( \hat{d}(v) \) as the minimum value of \( d_1(u) \), where \( u \in \text{Anc}(G = (V, E, w_0), v) \). To simplify the symbol, I treat lines 8-11 as \( i = 0 \) iteration and \( \hat{d}(v) \) as \( d_0(v) \). I also set \( \alpha(i) = (1 + \delta/\beta)^{\beta - i} \).

Before moving to the proof, I want to clarify some properties of the algorithm:

**Claim 6.2.1.** The following properties hold:

(6.2.1a) For any node \( v \in V \) and \( i \in [0, \beta] \), \( d_{i+1}(v) \geq d_i(v) \);

(6.2.1b) For any path \( u \leadsto v \) and \( i \in [0, \beta] \), \( d_i(v) \leq d_i(u) + w_i(u \leadsto v) \);
Algorithm 7 Producing a distance estimate with the relaxed triangle inequality

Input: An \((h, \epsilon)\)-hopset graph \(G = (V, E)\), with edge weights \(w_G : E \to \{0, 1, \ldots, K\}\) and a source node \(s \in V\)

Output: For all \(v\), a distance estimation \(\hat{d}(v)\) satisfying two conditions.

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 \(e \in E\) \(w_i(e) \leftarrow \frac{2^i}{16^i h^i} + (1 + \delta/\beta)^{\beta-i} \cdot w(e)\)
6: for \(v \in V\)
7: \(d_i(v) \leftarrow \min_{u \in V, u \sim v \in E}(d_i(u) + w_i(u \sim v))\)
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 \(v \in V\) \(\hat{d}(v) \leftarrow \min_{u \in V, u \sim v \in E}d_1(u) + w_0(u \sim v)\)

(6.2.1c) For any \(i \in [0, \beta]\), \(\alpha(i + 1) \leq (1 + \delta/\beta) \cdot \alpha(i + 1) \leq \alpha(i)\) and \(\alpha(0) \leq \frac{3}{2}\).

**Proof.** (6.2.1a) That is because \(d_i(v) \leq d_{i+1}(v) + w_i(v \sim v) = d_{i+1}(v)\).

(6.2.1b) Let \(x\) be the node such that \(d_i(u) = d_{i+1}(x) + w_i(x \sim u)\), then \(d_i(v) \leq d_{i+1}(x) + w_i(x \sim u \sim v) \leq d_i(u) + w_i(u \sim v)\).

\(\square\)

Now, I show that the distance estimations satisfy (6.2.1.1):

**Lemma 6.2.2.** For every \(v \in V\), I have \(\text{dist}(v) \leq \hat{d}(v) \leq 2\text{dist}(v)\)

**Proof.** First, I show \(\hat{d}(v) \geq \text{dist}(v)\). Note that in \(G\), I only round up each edge, and each \(d_i(v)\) is at least the shortest path distance with respect to \(w\). Therefore, \(\hat{d}(v) \geq \text{dist}(v)\).
Next, I want to show \( \hat{d}(v) \leq 2 \text{dist}(v) \). There are two cases, if \( \text{dist}(v) = 0 \), then there is a path \( s \rightsquigarrow v \) that contains only zero-weight edges. Consider the line 8–11, the path \( s \rightsquigarrow v \) is preserved with respect to \( w_0 \) and \( \hat{d}(v) = d_1(s) + w_0(s \rightsquigarrow v) = 0 \).

Otherwise, I know \( \text{dist}(v) \geq 1 \). Note that in graph \( G \), there exists a path \( p \) from \( s \) to \( v \) such that \( p \) contains at most \( h \) hops and \( w(p) \leq (1 + \epsilon) \text{dist}(v) \). Based on (6.2.1b), I know \( \hat{d}(v) \leq d_1(v) \leq d_1(s) + w_1(p) \leq w_1(p) \). I only need to bound \( w_1(p) \):

\[
\begin{align*}
\log_2 w_1(p) &\leq \frac{2}{16} \cdot h \cdot \frac{\gamma}{\beta} \cdot h + \alpha(1) \cdot w(p) \leq 1/2 + \alpha(0) \cdot \text{dist}(v) \quad \text{(if } \epsilon < \delta/\beta) \\
&\leq \frac{1}{2} \cdot \text{dist}(v) + \frac{3}{2} \cdot \text{dist}(v) \leq 2 \text{dist}(v).
\end{align*}
\]

Next, I want to show that (6.2.1.2) is also satisfied.

**Lemma 6.2.3.** For every \((u, v) \in E\), \( \hat{d}(u) + 2 \cdot w(u, v) \geq \hat{d}(v) \).

**Proof.** If \( w(u, v) = 0 \), based on (6.2.1b), I know \( \hat{d}(v) = d_0(v) \leq d_0(u) + w_0(u, v) = d_0(u) = \hat{d}(u) \).

Otherwise \( w(u, v) \geq 1 \), consider the node \( y \) such that \( \hat{d}(u) = d_1(y) \). Let \( y \rightsquigarrow u \) be the path that contains only edges of zero-weight, and \( y \rightsquigarrow v \) be the path concatenating \( y \rightsquigarrow u \) and the edge \((u, v)\). Then \( w(y \rightsquigarrow v) = w(u, v) \) and there must be a path \( p \) from \( y \) to \( v \) containing \( h \) hops and \( w(p) \leq (1 + \epsilon) \cdot w(y \rightsquigarrow v) = (1 + \epsilon)w(u, v) \). Consider \( w_1(p) \),

\[
\begin{align*}
\log_2 w_1(p) &\leq \frac{2}{16} \cdot h \cdot \frac{\gamma}{\beta} \cdot h + \alpha(1) \cdot w(p) \leq 1/2 + \alpha(0) \cdot w(u, v) \\
&\leq \frac{1}{2} \cdot w(u, v) + \frac{3}{2} \cdot w(u, v) \leq 2w(u, v)
\end{align*}
\]

for any \( \epsilon \leq \delta/\beta \). Based on (6.2.1a) and (6.2.1b), \( \hat{d}(v) \leq d_1(v) \leq d_1(y) + w_1(p) \leq d_1(y) + 2 \cdot w(u, v) = \hat{d}(u) + 2 \cdot w(u, v) \)

\[ \square \]
6.3 Updating Distance Estimate

I have already shown the correctness of Algorithm 7. However, updating \( d_i(v) \) must take into account all possible paths and is overly expensive if it is implemented in parallel and distributed models.

In Algorithm 7, I have two different cases. From lines 4-7, I need to update \( d_i(v) \) in the graphs without any zero-weight edges, and from lines 8-11, the graph contains only zero-weight edges. For the graph without any zero-weight edges, I will first upper bound the path weight I am exploring; since each edge weight also has a lower bound, the search depth can be bounded. For the zero edge weight graph, I can take advantage of the graph that contains only zero edge weight and use a divide-and-conquer method to update \( d_i(v) \) in this graph. In Chechik and Mukhtar [7], they use a similar method in the broadcast CONGEST model. I will present the implementation pseudocode in this section and leave the parallel and distributed details in Section 6.4.

6.3.1 Updating without Zero-weight Edges

I first update \( d_i \) in the graphs without any zero-weight edges. Based on how I round up each edge, I know that each edge is lower bounded by \( \Omega(2^i/(h\beta)) \). The following lemma states that if I use path \( u \rightsquigarrow v \) to update \( d_i(v) \) in round \( i \), \( w_i(u \rightsquigarrow v) \) can also be bounded by \( O(2^i) \).

**Lemma 6.3.1.** For any node \( u,v \in V \), path \( u \rightsquigarrow v \) and \( i \in [1,\beta] \), if \( d_i(v) = d_{i+1}(u) + w_i(u \rightsquigarrow v) \), then \( w(u \rightsquigarrow v) < 2^i \) and \( w_i(u \rightsquigarrow v) \leq 2^{i+1} \).

**Proof.** First, I show that \( w(u \rightsquigarrow v) < 2^i \). When \( i = \beta \), then \( w(u \rightsquigarrow v) \leq nW < 2^\beta \). 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 \( w(p) \leq (1+\epsilon) \text{dist}(u,v) \leq (1+\epsilon) \cdot w(u \rightsquigarrow v) \). Based on (6.2.1a) and (6.2.1b), I have \( d_i(v) \leq d_{i+1}(v) \leq d_{i+1}(u) + w_{i+1}(p) \). Combining with
assumption $d_i(v) = d_{i+1}(u) + w_i(u \rightsquigarrow v)$, I know $w_i(u \rightsquigarrow v) \leq w_{i+1}(p)$. Now I give a lower bound for $w_i(u \rightsquigarrow v)$ and an upper bound for $w_{i+1}(p)$. I have $w_i(u \rightsquigarrow v) \geq \alpha(i) w(u \rightsquigarrow v)$ for the lower bound. For the upper bound,

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

So,

$$w_i(u \rightsquigarrow v) \leq w_{i+1}(p) \Rightarrow \alpha(i) w(u \rightsquigarrow v) \leq \frac{2^i}{16 \beta} + \alpha(i + 1) (1 + \epsilon) w(u \rightsquigarrow v)$$

$$\Rightarrow \alpha(i) (\delta/\beta - \epsilon) w(u \rightsquigarrow v) \leq \frac{2^i}{16 \beta}$$

$$\Rightarrow w(u \rightsquigarrow v) < 2^i$$

for any $\epsilon \leq \frac{1}{4\beta}$. Once I show that $w(u \rightsquigarrow v)$ is bounded, since I use $u \rightsquigarrow v$ update $d_i(v)$, I have

$$w_i(u \rightsquigarrow v) \leq w_i(p) \leq \frac{2^i}{16 \cdot h \cdot \beta} h + \alpha(i) \cdot (1 + \epsilon) \cdot w(u \rightsquigarrow v) \leq 2^{i+1}$$

Now I can bound the distance estimate difference in round $i$.

**Lemma 6.3.2.** For any node $u, v \in V$ and $i \in [1, \beta)$ and path $u \rightsquigarrow v$, if $d_i(v) = d_{i+1}(u) + w_i(u \rightsquigarrow v)$, $d_{i+1}(v) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1}]$. Furthermore, for any node $x$ on the path $u \rightsquigarrow v$, I have $d_{i+1}(x) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1}]$.

**Proof.** Based on (6.2.1a), I have $d_{i+1}(v) \geq d_i(v) \geq d_{i+1}(u)$. On the other hand, there is a path $p$ from $u$ to $v$ that contains at most $h$ hops and $w(p) \leq (1 + \epsilon) \cdot w(u \rightsquigarrow v) \leq (1 + \epsilon) \cdot 2^i$. Based on (6.2.1b), I know that $d_{i+1}(v) - d_{i+1}(u) \leq w_{i+1}(p)$. I only need to bound $w_{i+1}(p)$:

$$w_{i+1}(p) \leq \frac{2^{i+1}}{16 \cdot h \cdot \beta} \cdot h + \alpha(i + 1) \cdot (1 + \epsilon) \cdot w(u \rightsquigarrow v) \leq 2^{i+1}$$
for any $\epsilon \leq \delta/\beta$. Combining the above inequalities, I have $d_{i+1}(v) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1}]$.

Now for any node $x$ on the path $u \leadsto v$. Let $u \leadsto x$, $x \leadsto v$ be the subpaths of $u \leadsto v$ splitted by $x$, I have $d_i(x) = d_{i+1}(u) + w_i(u \leadsto x)$. Otherwise, if $d_i(x) \neq d_{i+1}(u) + w_i(u \leadsto x)$. Since $d_i(x) \leq d_{i+1}(u) + w_i(u \leadsto x)$, I can assume that $d_i(x) < d_{i+1}(u) + w_i(u \leadsto x)$. However, based on (6.2.1b), $d_i(v) \leq d_i(x) + w_i(x \leadsto v) < d_{i+1}(u) + w_i(u \leadsto v)$ and this contradicts the fact $d_i(v) = d_{i+1}(u) + w_i(u \leadsto v)$. Since I have $d_i(x) = d_{i+1}(u) + w_i(u \leadsto x)$, $d_{i+1}(x) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1}]$.

\[\square\]

For iteration $i \in [1, \beta]$. Let $u$ be the node that I use to update $d_i(v)$, that is, $d_i(v) = d_{i+1}(u) + w_i(u \leadsto v)$. Note that for any node $x$ on the path $u \leadsto v$ (including $v$), I must have $d_{i+1}(x) \in [d_{i+1}(u), d_{i+1}(u) + 2^{i+1}]$. I can partition the graph according to the value $d_{i+1}$, let $A_j = \{y \mid d_{i+1}(y) \in [(j-1) \cdot 2^{i+1}, (j+1) \cdot 2^{i+1}]\}$ be the set of nodes whose $d_{i+1}$ is between $(j-1) \cdot 2^{i+1}$ and $(j+1) \cdot 2^{i+1}$, then update the value $d_i$ in the induced graphs $G(A_1), G(A_2), ..., G(A_{\lceil nK/2^i \rceil})$.

To see why the above partition works, consider an arbitrary node $v$. Assume that I use $u \leadsto v$ to update $v$ and $d_{i+1}(u) \in [k \cdot 2^{i+1}, (k+1) \cdot 2^{i+1})$. For any node $x$ on the path $u \leadsto v$, I have $d_{i+1}(x) \in [k \cdot 2^{i+1}, (k+2) \cdot 2^{i+1})$. In other words, the path $u \leadsto v$ is included in the graph $G(A_{k+1})$. Moreover, each node will be included in at most two induced graphs, which leads only to constant overhead.

The last part is to update the value of $d_i$ on each induced graph $G(A_j)$. Algorithm 8 updates the distance estimate in each induced graph. Note that for all $d_{i+1}(u) \in [(j-1) \cdot 2^{i+1}, (j+1) \cdot 2^{i+1})$ for $u \in A_j$, and $w_i(e) \geq \frac{\epsilon}{16 \cdot k \cdot \beta}$, I can use “BFS” to update $d_i$ value.
Algorithm 8 Distance estimate updates on induced graph

Input: Graph \( G(A_j) = (A_j, E(A_j), w_i) \), for any \( v \in V \), \( d_{i+1}(v) \in [(j \cdot 2^{i+1}, (j + 1) \cdot 2^{i+1}) \). 
Output: For all \( v \in A_j \), a distance estimation \( d_i(v) \)

1: function UpdateInducedGraphs \((G(A_j) = (V, E, w_i), d_{i+1})\)
2: \[ \Delta = 2^i \cdot \frac{2^i}{16 \cdot h \cdot \beta} \]
3: for \( k \in [0, \lceil 2^{i+2}/\Delta \rceil] \) \( F(k) \leftarrow \emptyset \)
4: for \( v \in V \) \( d_i(v) \leftarrow d_{i+1}(v), F(\text{INDEX}(d_i(v))) \leftarrow F(\text{INDEX}(d_i(v))) \cup \{v\} \)
5: for \( k \in [0, \lceil 2^{i+2}/\Delta \rceil] \) and \( F(k) \neq \emptyset \)
6: for \( v \in F(k) \) that \( v \) is not finalized
7: mark \( v \) as finalized
8: for \( (v, x) \in E \) and \( x \) that are not finalized
9: \( d_i(x) = \min(d_i(x), d_i(v) + w_i(v, x)) \)
10: \( F(\text{INDEX}(d_i(x))) \leftarrow F(\text{INDEX}(d_i(x))) \cup \{x\} \)
11: function Index \((d_i(v))\)
12: \[ \Delta = 2^i \cdot \frac{2^i}{16 \cdot h \cdot \beta}, b = (j - 1) \cdot 2^{i+1} \]
13: return \( \lceil (d_i(v) - b)/\Delta \rceil \)

The algorithm runs in rounds; at the beginning, the algorithm records each node’s index, which is based on the \( d_{i+1} \) value, and put the node in corresponding \( F(k) \). \( F(k) \) holds all nodes whose distance estimate once was between \([k \Delta, (k + 1) \Delta)\). Then in the \( k\)-th iteration, if \( F(k) \) contains node \( v \) and \( v \) is not finalized, I finalize \( d_i(v) \) and update all outgoing edges and the corresponding nodes.

Now I show that the distance estimation is correctly updated in the induced graphs.

**Lemma 6.3.3.** Consider a call to UpdateInducedGraphs \((G(A_j) = (A_j, E(A_j), w_i))\), for each node \( v \in A_j \), I have \( d_i(v) = \min_{u \in A_j, u \sim v \in E(A_j)} (d_{i+1}(u) + w_i(u \sim v)) \).

**Proof.** Proof by induction on \( k \). I want to show that at the \( k\)-th round, for any node \( v \) with \( d_i(v) < b + k\Delta \), \( d_i(v) = \min_{u \in A_j} (d_{i+1}(u) + w_i(u \sim v)) \). Note that at any time
of updating $d_i(v)$, I always have $d_i(v) \geq \min_{u \in A_j}(d_{i+1}(u) + w_i(u \sim v))$, so I only need to show $d_i(v) \leq \min_{u \in A_j}(d_{i+1}(u) + w_i(u \sim v))$ 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 $d_i(v) = d_{i+1}(v)$. Otherwise, there will be some node $u$ such that $d_{i+1}(u) = d_i(v) - w_i(u \sim v) \leq d_i(v) - \Delta < b$, which contradicts the fact that $d_{i+1}$ is at least $b$.

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

I have already shown that if I use $u$ to update $v$, then the path $u \sim v$ must be in some induced graph. There will be multiple $d_i(v)$ since $v$ might be in two induced graphs, and I choose the minimum value as $d_i(v)$. If I apply Lemma 6.3.3, $d_i(v)$ is correctly updated for any node $v \in V$. This gives us the following theorem.

**Theorem 6.3.4.** If I call UpdateInducedGraphs on subgraphs $G(A_1), G(A_2), ...,$ $G(A_{\lceil nK/2 \rceil})$ and set $d_i(v)$ to the minimum value of $d_i$ returned by different subgraphs, then $d_i(v) = \min_{u \in V, u \sim v \in E}d_{i+1}(u) + w_i(u \sim v)$.

**Proof.** Notice that path $u \sim v \in E(A_j)$ for some $j$. Based on Lemma 6.3.3,

$$d_i(v) = \min_{i \in [1, \lceil nW/2 \rceil], u \in V, u \sim v \in E(A_i)}d_{i+1}(u) + w_i(u \sim v) = \min_{u \in V, u \sim v \in E}d_{i+1}(u) + w_i(u \sim v).$$

105
Algorithm 9 Distance estimate updates on zero-weight graphs

Input: Graph $G = (V, E, w)$ such that for any $e \in E$, I have $w(e) = 0$; $D$ is the set containing possible $d_1$ values

Output: For all $v$, a distance estimation $\hat{d}(v)$

1: function UPDATEZEROWEIGHTGRAPHS($G = (V, E, w_0), D$)
2:    $d_{mid} = \text{the median of } D$
3:    if $|D| = 1$ then
4:        for $v \in V$ $\hat{d}(v) \leftarrow d_{mid}$
5:    else
6:        $S \leftarrow \{v \mid v \in V, d_1(v) \leq d_{mid}\}$
7:        $V_l \leftarrow \text{Des}(S)$, $V_u = V \setminus V_l$
8:        $D_l \leftarrow \{d \mid d \in D, d \leq d_{mid}\}$, $D_u = D \setminus D_l$
9:        UPDATEZEROWEIGHTGRAPHS($G(V_l), D_l$)
10:       UPDATEZEROWEIGHTGRAPHS($G(V_u), D_u$)

6.3.2 Updating on Zero-weight Graph

I still have to update the zero-weight graph. In this graph, I will remove all nonzero-weight edges and set $\hat{d}(v)$ to the minimum $d_1$ value of its ancestor nodes. The algorithm uses a divide-and-conquer method to solve it. In the algorithm 9, I will maintain a set $D$ such that $D$ contains all possible $d_1$ values for $v \in V$. At the beginning, for each node $v$, the algorithm calls 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, I 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 $d_{mid}$. I will show how to use BFS compute $V_l$, which is the descendant nodes of $S$. $V_u$ is the node in $V \setminus \text{Des}(S)$. The Algorithm 9 recurs on induced graphs with $D_l$ and $D_u$ set.

The following lemma plays a key role in showing the correctness and the implementation.
Lemma 6.3.5. Given nodes $u, v \in V$ and graph $G = (V, E, w_0)$. If $\hat{d}(v) = d_1(u)$, then there is a zero weight path $p$ from $u$ to $v$ in $E(G)$ that contains at most $h$ hops. Furthermore, consider any call `UpdateZeroWeightGraphs(G', D')` in the recursion tree, if $v \in V(G')$, then $d_1(u) \in D'$ and the path $p \in E(G')$.

Proof. Assume that I path $u \rightsquigarrow v$ update $v$, then $w_0(u \rightsquigarrow v) = 0$ and $w(u \rightsquigarrow v) = 0$. Note that the graph is a hopset graph, I know there will be a path $p$ from $u$ to $v$ containing at most $h$ hops and $w(p) \leq (1 + \epsilon)w(u \rightsquigarrow v) = 0$. Since $p$ contains only zero-weight edges, it is also in $G$.

To show the second part, I will use induction for each subproblem. At the beginning, I know $D$ contains all possible $d_1$ value and all nodes are in the graph. When I call `UpdateZeroWeightGraphs(G, D)`, if $v \in V(G)$. Based on induction, I have $p \in E(G)$, $d_1(u) \in D$. I want to show that all nodes on $p$ go to the same subproblem and the corresponding $D$ set contains the $d_1(u)$ value. Note that for any node $x$ on $p$, $\hat{d}(x) = d_1(u)$. Otherwise, if $\hat{d}(x) < d_1(u)$, I will have $\hat{d}(v) \leq \hat{d}(x) < d_1(u)$, which is a contradiction.

Let $\tilde{S}$ be set containing first half elements of sorted $D$ and let $x$ be an arbitrary node on path $p$. If $u \in \tilde{V}_i$, then $x \in \text{Des}(u) \subset \text{Des}(\tilde{V}_i) \subset \text{Des}(\tilde{S}) = \tilde{V}_i$ and $d_1(u) \in \tilde{D}_i$. Otherwise $u \in V_u$, based on the induction, I have $d_1(u) \in \tilde{D}$, so $d_1(u) > d_{mid}$. If $x \in \tilde{V}_i$, I know that $\hat{d}(x) \leq d_{mid} < d_1(u)$, and this is a contradiction. So $x \in \tilde{V}_u$. In either case, $p$ goes to the same subproblem and $d_1(u)$ will be in the corresponding $D$ set.

Using the lemma 6.3.5, I can show the correctness of the Algorithm 9.
Lemma 6.3.6. If I call \texttt{UpdateZeroWeightGraphs}((V, E, w_0), \{d_1(v) \mid v \in V\}), then \(d'(v) = \min_{u \in V, u \sim v \in E} d_1(u) + w_0(u \sim v)\). In addition, the depth of the recursion is \(O(\log n)\).

Proof. To bound the recursion depth. The partition of the nodes might be unbalanced. However, note that the size of \(D\) drops by half in each level of the recursion, so the depth of the recursion is at most \(O(\log n)\).

Based on lemma 6.3.5, I know that there is a zero weight path \(p\) from \(u\) to \(v\). Finally, I will call \texttt{UpdateZeroWeightGraphs}(G', D') such that \(D'\) contains only one element. If \(v \in V(G')\), \(d_1(u) \in D\) and I will set \(d'(v)\) to \(d_1(u)\).

All codes in Algorithm 9 are straightforward, except that I need to compute \(V_i \leftarrow \text{Des}(S)\). In each \texttt{UpdateZeroWeightGraphs}(G', D'), if \(v \in V_i\), there must be a path \(p\) from \(u \in S\) to \(v\) that contains at most \(h\) edges. I can add a virtual source node \(s'\) and edges from \(s'\) to all nodes in \(S\), then a BFS from the source node \(s'\) with depth \(h + 1\) is enough to detect all nodes in \(V_i\).

6.4 Translating to Parallel and Distributed Models

In this section, I describe the details of my exact SSSP algorithm in the parallel and distributed model. In the parallel model, I only need to repeat constructing the hopset graph and computing the distance estimate some times. In the distributed model, I have to use the framework of framework of Forster and Nanongkai and compute exact SSSP on the skelenton graph instead of the original graph.
6.4.1 Implementation in Parallel Model

I 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, I first put all nodes in an array and sort the array based on $d_{i+1}(v)$. For each $A_j$, I can simply mark the first and last nodes and compute the size of $A_j$. I 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 8 is naturally parallelized. In each induced graph, I need to search with depth $O(2^i/\Delta) = O(h\beta)$, and each edge will be accessed once. I only need to focus on the following point; I cannot afford to transverse all the elements in $F$. If $F(k)$ is empty, I should skip it. Otherwise, the work is at least $\Omega(h\beta)$ in each induced graph. I aim to take $\tilde{O}(m_j)$ work for each induced graph. To implement $F(k)$ efficiently, I can use the parallel ordered set as a tool. Specifically, there exist implementations of parallel ordered sets [3] that support the following operations:

- Initialization, given a vector of size $n$, I can construct a new ordered set in $\tilde{O}(n)$ work and $\tilde{O}(1)$ span and get an identifier of the ordered set.
- I 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 $n \geq m$. 

109
• I can enumerate all elements in a size $n$ set in $O(n)$ work and $\tilde{O}(1)$ span.

• I can extract the minimum value with $O(1)$ work and $\tilde{O}(1)$ span.

• Given a key, I 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 I can use the parallel ordered set to implement $F(k)$. In lines 3-4 of Algorithm 8, I simply compute $\text{INDEX}(d_i(v))$ and initialize the parallel ordered set with element $(\text{index}(d_i(v)), v)$. In lines 5 - 10, to transverse the next non-empty $F(k)$, I can first extract the minimum value $(k, v)$ from the parallel ordered set, then I will split the parallel ordered set by using the key $(k + 0.5, *)$ and get all the items whose index once was $k$. Then I 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 I need a vector to distinguish all finalized nodes. I 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. I 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 I do not need to insert them back into the parallel set. I 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 $\text{index}$ for each node, but the work is still bounded, since I 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 6.4.1.** Algorithm 8 can be implemented in $\tilde{O}(m)$ work and $\tilde{O}(h \log K)$ span in the work-span model.

*Proof.* Omitted. $\square$
Updating \( \hat{d} \) on zero-weight graph. I can use sorting to compute \( D \) and \( S \). As discussed in Section 6.3.2, I need a \( O(h) \)-depth BFS to compute \( \mathcal{V}_l \), which can be achieved with \( \tilde{O}(m_j + h) \) work and \( O(h) \) span [24], where \( m_j \) is the edge number in the induced graph. Note that to avoid \( O(h) \) work on each induced graph, when \( m_j < h \), I do \( O(m_j) \)-depth BFS and the work is \( \tilde{O}(m_j) \). To recurse on the induced graph, I 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 I have \( O(\log n) \) level of recursion, this gives us the following lemma.

**Lemma 6.4.2.** Algorithm 9 can be implemented in \( \tilde{O}(m) \) work and \( \tilde{O}(h \log K) \) span in the work-span model.

*Proof.* Omitted.

Combining together. Note that when I update the distance estimation, I have to run \( O(\log K) \) rounds of Algorithm 8 and \( O(1) \) rounds of the Algorithm 9.

**Lemma 6.4.3.** Given a \( (h, \epsilon = \frac{1}{4 \log (2nK)}) \)-hopset graph \( G \) with non-negative integer weight 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 Lemma 6.4.1 and Lemma 6.4.2, the lemma holds when \( \epsilon \leq \frac{1}{45} \).

To make a general graph a \( (h, \epsilon) \)-hopset graph, I can use the parallel hopset results from Theorem 3.5.3.

**Theorem 3.5.3.** For any directed graph \( G = (V, E) \) with non-negative real edge weights, 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.
To solve the exact SSSP for digraphs with edge weight from \{0, 1, ..., W\}, I have to transfer a general graph to a hopset graph, compute the distance estimate, and repeat the whole process \(\log W\) times. Fortunately, when I construct the hopset and compute the distance estimate, I can assume \(K = O(n)\) and set \(\epsilon = \frac{1}{4\log(2nK)}\); this gives us the following theorem.

**Theorem 6.4.4.** There is a 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.

**Proof.** The theorem is implied by Lemma 6.4.3 and Theorem 3.5.3.

### 6.4.2 Implementation in Distributed Model

I consider the SSSP problem in the CONGEST and broadcast CONGEST model, 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. For SSSP in the CONGEST model, each node \( v \) must learn its distance estimate \( \tilde{d}(s, v) \), but these distances should not be communicated back to \( s \). 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, I assume that all nodes already know \( n \). I heavily use the concept of skeleton graph. When I say skeleton graph, I 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 I 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 6.4.5.** [27] 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 6.4.6.** [18] Consider \( k \) distributed algorithms \( A_1, \ldots, A_k \). Let \textit{dilation} be such that each algorithm \( A_i \) finishes in \( \text{dilation} \) rounds if it runs individually. Let \textit{congestion} be such that there are at most \( \text{congestion} \) messages, each of size \( O(\log n) \), sent through each edge (counted over all rounds), when I run all algorithms together. There is a distributed algorithm that can execute \( A_1, \ldots, A_k \) in \( \tilde{O}(\text{dilation} + \text{congestion}) \) rounds in the CONGEST model.
Framework of Forster and Nanongkai [17] Next, I present an overview of the exact SSSP algorithm, which follows the Forster and Nanongkai [17] framework. The algorithm is parameterized by $\alpha$, 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 [17] provide two methods, simulating Dijkstra algorithm and recursing exact SSSP on the skeleton graphs. Chechik and Mukhtar [7] improve the exact SSSP result by using iterative gradual rounding in a different way. I 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 $d_{G_S}(s, v)$. 

114
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 I 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 $G_s \cup G$.

I want to mention that this framework computes a distance estimate that can be used to solve the exact SSSP. To solve the exact SSSP for graph $G$, I need to repeat $O(\log W)$ times the above algorithm, but each time I repeat, I 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, I will show that Step 4 can be implemented in the following rounds.

**Lemma 6.4.7.** In the CONGEST model, there is a randomized algorithm solving exact SSSP problems on a $\tilde{O}\left(\alpha \right)$ node skeleton graphs with nonnegative integer edge weight from $\{0, 1, 2, ..., K\}$ in $\tilde{O}\left((\alpha \rho^2 + \frac{D\alpha^{1/2+o(1)}}{\rho}) \log K\right)$ 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 6.4.7, it takes $\tilde{O}(\alpha \rho^2 + \frac{D\alpha^{1/2+o(1)}}{\rho})$ rounds to solve the exact SSSP on 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 I need to repeat the entire algorithm $\log W$ times. I have three cases.

**Case 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.

**Case 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.

**Case 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 of Theorem 6.1.3.** 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.

**Solving exact SSSP on the skeleton graphs**

In this subsection, I show an algorithm to compute the distance estimate for an $\alpha$-node skeleton graph $G$. At the end of the algorithm, each node $v$ only needs to know its own $\hat{d}(v)$. Then based on Lemma 6.4.5, 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. I first need to give each $A_j$ an identifier. There are at most $\alpha$ nodes; I have at most $\alpha$ 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 6.4.5, each round takes $O(D + x_k)$ rounds and $O(x_k)$ congestion on each edge, where $x_k$ is the unfinished node in $F(k)$.
and . In total, Algorithm 8 takes $\tilde{O}(Dh\beta + n_j)$ rounds and $O(n_j)$ congestion sent through each edge, where $n_j = |A_j|$ is the number of nodes in the induced graph $G(A_j)$. Using Lemma 6.4.6, it takes $\tilde{O}(Dh \log K + \alpha)$ rounds to run Algorithm 8 on all induced graphs.

Updating $\hat{d}$ on zero-weight graph  
Algorithm 9 is a recursive algorithm. At each level of recursion, I 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 9 takes $\tilde{O}(Dh + \alpha)$ rounds. This gives us the following lemma.

**Lemma 6.4.8.** Given an $\alpha$-node $(h, \epsilon = \frac{1}{4\log(2n\hat{K})})$-hopset 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.** I have to run Algorithm 8 $\log \hat{K}$ times and Algorithm 9 once, which gives us $\tilde{O}(Dh \log^2 \hat{K} + \alpha \log \hat{K})$ running time. \hfill \square

In Theorem 4.4.1, I show the following hopsets results for skeleton graphs,

**Theorem 4.4.1.** 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 $\tilde{O}(\alpha \rho^2 \log \hat{K}/\epsilon^2)$, I can use Lemma 6.4.5 to broadcast edges in the hopset and construct a new skeleton graph with the hopset property. In the new skeleton graph, I can compute the distance estimate and the running time is given by Lemma 6.4.8. Combining Lemma 6.4.8 and Theorem 4.4.1 gives us Lemma 6.4.7.
Proof of Lemma 6.4.7. Note that the maximum edge weight is $K$, but when I compute the distance estimate each round, the maximum edge weight is $\hat{K} = O(n)$. The log $K$ factor comes from the fact that I need to repeat Algorithm 7 log $K$ times. □
References


Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. 

*57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 128–137, Los Alamitos, CA, USA, 


*The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’19, pages 333–341, New York, NY, USA, 


