Algorithm for Large Scale 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

Katina Russell, M.S.

Washington, DC
July 8, 2022
ALGORITHMS FOR LARGE SCALE GRAPHS

Katina Russell, M.S.

Dissertation Advisor: Jeremy T. Fineman, Ph.D.

ABSTRACT

Constructing efficient algorithms for graph problems is a fundamental problem in computer science theory. This dissertation studies algorithms for large scale graphs and focuses on directed graph problems. We consider directed graph problems in several models of computation with the goal of improving performance for large scale graphs. The models of computation include the external memory model, the work-span model for parallel algorithms, and the asymmetric RAM model. The hypothesis is there exist provably efficient algorithms for directed graphs problems for large scale graphs.

The following results are presented. First, an I/O-efficient algorithm for topologically sorting a directed acyclic graph, as well as an algorithm for identifying and topologically sorting the strongly connected components of a directed graph. Both algorithms cost \( O(E/B \log M/B V/B \cdot \log^4 V) \) I/Os and are the first I/O-efficient algorithms for these problems for sparse graphs. Second, this work shows an algorithm for constructing a \((1+\epsilon)\)-approximate directed hopset of size \(\tilde{O}(n)\) in \(\tilde{O}(m)\) work and \(n^{1/2+o(1)}\) hopbound. Our parallel version of the algorithm can be used to solve parallel approximate single-source shortest paths in \(\tilde{O}(m)\) work and \(n^{1/2+o(1)}\) span. Next we show a parallel algorithm for distance-limited shortest paths on directed acyclic graphs with 0 and -1 edge weights. The algorithm computes single-source shortest paths for nodes with distance at least \(-L\), and runs in \(\tilde{O}(m)\) work and \(O(L^{1/2}n^{1/2+o(1)})\) span. Finally, we give write-efficient algorithms for breadth-first search, depth-first search
and strongly connected components. The standard RAM algorithms for these problems write the solution for each node. Instead, we write only a subset of the nodes to save writes, at the expense of more reads to answer a query. Our result is sublinear size data structures that can answer queries for each of the three graph problems.

INDEX WORDS: Graph problems, directed graphs, parallel algorithms, external memory model, asymmetric memory models
# Table of Contents

## Chapter

1. Introduction .......................................................... 1
   1.1 External-Memory Algorithms ...................................... 3
   1.2 Parallel Algorithms ............................................. 7
   1.3 Asymmetric RAM Algorithms .................................... 10

2. Notation and Terminology ........................................... 12
   2.1 Bounds ........................................................... 12
   2.2 Graphs .......................................................... 12

3. I/O-Efficient Topological Sort .................................... 14
   3.1 Introduction .................................................... 14
   3.2 Topological Sort Algorithm .................................... 20
   3.3 Topological Sort Analysis ...................................... 31
   3.4 Strongly Connected Components ............................... 40
   3.5 Conclusions ................................................... 56

4. Directed Hopsets and Parallel Approximate Shortest Paths ....... 59
   4.1 Introduction .................................................... 59
   4.2 Preliminaries ................................................... 63
   4.3 Unweighted Sequential Hopsets Algorithm ..................... 64
   4.4 Hopsets Analysis ............................................... 68
   4.5 Hopsets for Weighted Graphs ................................. 97
   4.6 Parallel Algorithm for Hopsets ............................... 100

5. Parallel Distance Limited Shortest Paths on Acyclic Graphs with \{0,-1\} Edge Weights ................................................... 107
   5.1 Introduction .................................................... 107
   5.2 Algorithm ...................................................... 109
   5.3 Analysis ....................................................... 114

6. Asymmetric-RAM Algorithms ........................................ 124
   6.1 Introduction .................................................... 124
   6.2 Breadth-First Search .......................................... 127
   6.3 Depth-First Search ............................................. 134
   6.4 Strongly Connected Components ................................ 143

REFERENCES ............................................................. 150
Chapter 1

Introduction

Graph algorithms are a fundamental problem in computer science theory. In this work, we study algorithms for graph problems with the goal of improving performance runtime for large scale graphs. There are three main areas that we study to improve performance for large scale graphs.

One issue for large scale graphs is that the data can be too large and has to be stored on an external memory. In this case, the cost of accessing the data can be much higher than the cost of an in-memory operation. The RAM model falls short in capturing the performance of an algorithm on data that is stored liked this, so we consider problems in the I/O model. The I/O model or external memory model [2] is a model of computation that models locality, and consists of a two-level memory hierarchy. There is a small cache where data can be processed, and an unbounded large memory. The data is arranged in to $B$-sized blocks that must be transferred from the external memory to the internal memory in order to be processed. Each block transfer is called an I/O and the cost of an algorithm in this model is the number of I/Os. A good I/O algorithm utilizes the locality of the blocks and avoids performing too many I/Os. In this dissertation, we construct I/O-efficient algorithms to reduce expensive memory accesses for large graphs stored in external memories.

Second, the runtime for an algorithm running on a large graph can be too slow. To address this problem, we introduce extra processors so that we can spread the work out among the processors to get speed-up. However we need efficient parallel
algorithms to ensure that we use the extra processors effectively. For a parallel algorithm the work is the total number of operations and the span is the longest chain of sequentially dependant operations. Our goal is to develop algorithms that have work similar (within polylogarithmic factors) to the best sequential algorithm and reduce the span as low as possible. In doing so, we can increase parallelism, which is the number of processors we can use in order to get speed up.

Finally we must cope with new memory technologies. There are existing non-volatile memory technologies [4, 32, 43, 61] which have a higher cost for writes than for reads. Non-volatile memory has lower energy costs because they no not need power to maintain a memory state, unlike DRAM which needs power to hold data. However, for non-volatile memory reading data from memory is much faster than writing data to memory. The cost can be higher by an order of magnitude or more in terms of energy, latency and bandwidth [11, 13]. To improve the runtime in these types of memory, we want to develop algorithms that perform less writes. The higher write cost is modeled by the asymmetric RAM model [15], which has a small symmetric memory, and an unbounded memory where the reads have unit cost and the writes have cost $\omega$. We design algorithms in this model that reduce the number of writes, at the expense of more reads, to improve the runtime of algorithms for data on stored in non-volatile memories.

We consider graph problems in three memory models each with the goal of addressing performance issues at scale. For many graph problems more progress has been made on undirected graphs while directed result lags behind. This work focuses on directed graphs where recent results have allowed us to make some progress. This dissertation will show the following hypothesis.

Hypothesis 1. There exist provably efficient algorithms for directed graph problems for large scale graphs.
Next we will discuss the problems that we study in this dissertation including background information and an overview of the results. In Chapter 2 we will show some notation and terminology used throughout this work. Then the main results start in the following chapter.

1.1 External-Memory Algorithms

1.1.1 External-Memory Model [2]

The I/O model [2], also called the external-memory model or disk-access-machine model, is a standard theoretical model of computation for understanding the performance of algorithms on large data sets by capturing some notion of locality. The I/O model [2] is a two-level memory hierarchy comprising a size-$M$ cache (also called internal memory) and an external memory of unbounded size. All data, both in cache and in external memory, is organized in size-$B$ blocks, so the cache consists of $M/B \geq 1$ blocks. Data can only be processed in the cache, meaning that data must be transferred from the external memory to cache when needed. These data transfers are performed at the granularity of blocks; each block transfer is called an I/O. The cost of an algorithm in the I/O model, often called the I/O cost, is the number of I/Os performed. Computation itself is free.

Some examples of bounds in this model are scanning a list of $N$ items costs $\text{scan}(N) = \Theta(N/B)$, and sorting $N$ items costs $\text{sort}(N) = \Theta(N \log_{M/B}(N/B))$ [2]. A key separation between the RAM model and the I/O model is the difference in cost between models for permuting. In the RAM model, permuting an array is as cheap as scanning. In the I/O model, for most settings of machine parameters permuting is generally as expensive as sorting. Specifically, permuting has I/O cost $\Theta(\min \{N, \text{sort}(N)\})$ [2], which for typical values resolves to the sort bound. (The
$N$ term corresponds to foregoing an I/O-efficient algorithm entirely — simply run the RAM algorithm and pay an I/O for every operation.) The cost of sorting thus often serves as a lower bound on the I/O cost for problems that can be solved in linear time in the RAM model. Many basic graph problems on sparse graphs (directed or undirected), including topological sort, have $\Omega(sort(V))$ lower bounds in the I/O model [25].

1.1.2 Related Work on the External Memory Model

There is a large body of work, e.g., [1, 9, 10, 20, 25, 26, 42, 47, 50, 52, 53, 55] on graph algorithms in the I/O model. See [60] or [62] for good surveys on the topic. For undirected graphs, many problems can be solved in $O(sort(E))$ I/Os. (In fact, for dense graphs the logarithmic term in the sort bound can be improved slightly through sparsification [36].) For example, connectivity problems such as connected components, minimum spanning forest, biconnectivity, list ranking, independent sets, and maximal matching can all be solved in $O(sort(E))$ I/Os [25], with high probability. If randomization is not allowed, there are several deterministic algorithms [1, 9, 25, 47, 55], which tend to be at worst logarithmic factors from the sort bound. For undirected breadth-first search, Mehlhorn and Meyer’s solution runs in $O(\sqrt{EV/B} + sort(E) + SF(V, E))$ I/Os, where $SF(V, E)$ is I/O the bound for computing a spanning forest.

The directed analog of the connectivity problems are the reachability problems such as single-source reachability, topological sort, and strongly connected components. The best known bounds for these problems are significantly worse than for their undirected counterparts. The algorithm due to Chiang et al. runs in $O(V + VE/(MB) + sort(E))$ I/Os [25], and the one due to Buchsbaum et al. runs in $O((V + E/B) \log(V/B))$ [20]. These two algorithms were the best existing algorithms
but both have a $|V|$ term in their I/O cost, which is not I/O-efficient in general. If the graphs are restricted to be planar graphs, however, many of these problems and more can be solved in $O(sort(E))$ I/Os [8, 9, 49].

The bounds for problems relating to paths, such as shortest paths and depth-first search, tend to be worse than the connectivity problems. Many algorithms for shortest-path problems [50, 52, 53] have I/O costs that include a $\sqrt{VE/B} = \Omega(E/\sqrt{B})$ term. For undirected single-source shortest paths, Meyer and Zeh’s algorithm runs in $O(\sqrt{VE/B} \log(W/w) + MST(V, E))$ I/Os, where $MST(V, E)$ is the I/O bound for computing minimum spanning forest, and $W$ and $w$ are the maximum and minimum edge weights, respectively [52]. Meyer and Zeh later removed the dependence on the edge weights to achieve a cost of $O(\sqrt{VE/B} \log V + MST(V, E))$ I/Os [53]. For the directed version of the problem, Kumar and Schwabe develop an I/O-efficient priority queue and use it in Dijkstra’s algorithm for a solution that runs in $O(V + E/B \log(E/B))$ I/Os [47]. Jiang and Kasper improve the I/O-efficient priority queue which leads to the improved cost of $O(\sqrt{VE/B} \log V + MST(V, E))$ I/Os [42]. All-pairs shortest paths can be solved in $O(V \cdot sort(E))$ in unweighted, undirected graphs [10], and $O(V \sqrt{VE/B} + \frac{VE}{B} \log(E/B))$ in weighted undirected graphs [26]. For depth-first search the best algorithms run in $O(V + VE/(MB) + sort(E))$ I/Os [25], and $O((V + E/B) \log(V/B))$ I/Os [20].

1.1.3 Our Results

In Chapter 3, we will show an algorithm for topological sort in the I/O model. The topological sort problem is given a directed acyclic graph (DAG), $G = (V, E)$ to produce an ordering of the vertices such that for each edge $(u, v) \in E$, $u$ precedes $v$ in the ordering. Topological sort is a fundamental problem in for directed acyclic graphs and it also used for Time Forward Processing, which which is a technique for
solving boolean circuits on a DAG. Time forward processing requires that the graph is topologically sorted which motivated the need for an I/O-efficient topological sort algorithm.

In the RAM model topological sort can be solved in linear time by using depth-first search. Using the RAM depth-first search algorithm in the I/O model gives an algorithm with I/O cost $O(|V| + |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Previous work on this problem [20, 25] try to make the depth-first search RAM approach I/O-efficient, but each have at least one I/O per vertex. Chiang et al. [25] achieve an I/O cost of $O(V + \frac{E}{B} \log \frac{M}{B}(E/B) + \frac{VE}{MB})$, and Bushsbaum et al. show an algorithm with I/O cost $O((V + E/B) \log V/B)$. Ajwani et al. [3] propose an algorithm that is meant to work well in practice. They use an iterative approach in their solution and achieve an I/O cost of $O(\ell \cdot \frac{E}{B} \log \frac{M}{B}(E/B))$, where $\ell$ is the length of the longest path in the graph. Our result is an algorithm which has I/O cost $O(\frac{E}{B} \log \frac{M}{B}(E/B) \log^{5} V)$, with high probability [21, 24]. This is more efficient than previous works for sparse graphs, and eliminates having one I/O per vertex. We also show how to compute strongly connected components for general directed graphs in the same cost bound. For a directed graph, two nodes $u$ and $v$ are strongly connected if there is a path from $u$ to $v$ and a path from $v$ to $u$. Strongly connected components are maximal subgroups where each pair of nodes in a subgroup are strongly connected. We our algorithm for strongly connected components is an extension of the topological sort algorithm.
1.2 Parallel Algorithms

1.2.1 Work-Span Model [31]

The work-span model [31] is a algorithm cost model parallel algorithms. In this model, the algorithms do not need to explicitly schedule the work or model processors. Instead the algorithms can express parallel processing through parallel loops, such as the parallel foreach. There is no assumption on the ordering of tasks in a parallel foreach, or any synchronization between parallel tasks but they will all be completed before the next step (outside of the parallel foreach loop) in the algorithm.

We use work and span to measure the performance of a parallel algorithm. The work of an algorithm is the total number of primitive steps it performs, or the runtime of the algorithm if there is one processor. The span, also called the depth, of an algorithm is the longest chain of sequentially dependant operations, or the runtime of the algorithm if there are infinite processors.

An ideal parallel algorithm has work that is close to the work for the best sequential algorithm, i.e. within polylogarithmic factors, and the span as low as possible. The parallelism of an algorithm is work/span, which represents the number of processors we can use to still get speedup. It has been shown that an algorithm with \( W(n) \) work and \( S(n) \) span can be scheduled to run in \( O(W(n)/p + S(n)) \) time, where \( p \leq W(n)/S(n) \) [18].

1.2.2 Background and Motivation

One of the most basic graph problems is the single-source shortest paths problem. That is, given a graph \( G \) and source node \( s \) to compute the shortest path from \( s \) to each node in \( G \). The problem has been well studied in the sequential setting and
for a $n$-node, $m$ edge graph with non-negative integer edge weights, can be solved in $(m + n \log n)$ time using Dijkstra’s algorithm [31].

Given that the sequential algorithm runs in nearly linear time, an ideal parallel algorithm for single-source shortest paths would run in $\tilde{O}(m/p)$ time for $p$ processors, where $\tilde{O}$ hides polylogarithmic factors. However even for unweighted, undirected graphs, the best algorithms have work that is polynomially higher than we want or linear span meaning the algorithm is inherently sequential. The problem has been well [16, 19, 33, 44, 51, 56, 58] studied but no ideal solutions exist.

For directed graphs with non-negative integer edge weights running Dijkstra’s algorithm has $O(m + n \log n)$ work and $O(m + n \log n)$ span. There is a parallel version of Dijkstra’s algorithm which has $O(m + n \log n)$ work and $O(n)$ span [19]. Ullman and Yannakakis are able to achieve sublinear span, though at the expense of high work. Their algorithm works on unweighted graphs and runs in $O(mn^{1/2})$ work and $O(n^{1/2})$ span [58]. Klein and Subramanian extend Ullman and Yannakakis’ algorithm to run on graphs with integer edge weights and runs in the same bounds [44]. Spencer’s algorithm exhibits a tradeoff between work and span, parameterized by $p$, and runs in $\tilde{O}(m + np^2)$ work and $\tilde{O}(n/p)$ span [56]. Finally, there is another work-span tradeoff algorithm which runs in $\tilde{O}(mt + mn/t + n^3/t^3)$ work and $\tilde{O}(t)$ span for $1 \leq t \leq n$ [46].

Since the exact shortest paths seems difficult to solve, research has turned to generalizations of the problem. The approximate version of the problem has been found to be more promising for undirected graphs. The $(1 + \epsilon)$ approximate version of the shortest paths problem is given a graph $G$ and source node $s$, to compute a distance estimate $d'(v)$ from $s$ to each node $v$ in the graph such that $d(v) \leq d'(v) \leq (1 + \epsilon)d(v)$, where $d(v)$ is the true shortest path distance from $s$ to $v$. For undirected
graphs, there exist solutions which have $\tilde{O}(m)$ work and $\tilde{O}(1)$ span [5, 48]. However for the directed version of the problem less progress has been made.

### 1.2.3 Hopsets and Parallel Approximate Shortest Paths

In Chapter 4, we study parallel approximate shortest paths for directed graphs. Our result is an algorithm for $(1 + \epsilon)$ approximate shortest paths that runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span [22]. This is the first algorithm to simultaneously achieve nearly (within polylogarithmic factors) linear work and sublinear span for directed graphs.

To solve approximate shortest paths we construct a directed hopset. For a directed graph $G = (V, E)$, a $(\beta, \epsilon)$-hopset is a set of edges $E'$ such that, if $E'$ is added to the graph, then the following is true for graph $G' = (V, E \cup E')$. For each pair of nodes $u$ and $v$ there exists a path from $u$ to $v$, where the number of hops on the path is at most $\beta$, and $d_G(u, v) \leq d_{G'}(u, v) \leq (1 + \epsilon)d_G(u, v)$, where $d_G(u, v)$ and $d_{G'}(u, v)$ are the distances from $u$ to $v$ in $G$ and $G'$, respectively. The hopbound is $\beta$, and the size of the hopset $|E'|$ is the number of edges in $E'$. Once we have a hopset we can solve approximate shortest paths by using Klein and Subramanian’s distance limited shortest paths algorithm [44]. Our algorithm for parallel construction of a directed hopset has the same work and span as the parallel approximate shortest paths bounds.

### 1.2.4 Parallel Distance Limited Shortest Paths on Acyclic Graphs with $\{0,-1\}$ Edge Weights

Next in Chapter 5 we consider another relaxation of the exact single-source shortest paths problem in the parallel setting. Given an acyclic graph with 0 or -1 weighted edges, we would like to compute the shortest paths from a source node up to a certain distance limit $L$. That is, we compute the shortest path distance for any node with shortest path distance at least $-L$, and return $-\infty$ for all other nodes. This is
equivalent to solving longest paths with 0 and 1 edge weights in DAGs. The algorithm can also be used for finding long chains (up to $L$) in a DAG.

In the sequential setting, this problem can be solved by topologically sorting the DAG, and then processing the nodes one by one. This solution gets $O(m+n)$ work but would have linear span. Our algorithm runs in $\tilde{O}(m)$ work while achieving sublinear span, $L^{1/2}n^{1/2+o(1)}$.

1.3 Asymmetric RAM Algorithms

There are technologies emerging for memory reads can be performed much faster than writes [4, 32, 43, 61]. The write cost can be an order of magnitude (or more) more expensive in terms of energy, latency and bandwidth [11, 13]. By reducing the number of writes, even at the expense of increasing the number of reads, we can improve performance. These technologies motivate the need for write-efficient algorithms, which the following model captures.

1.3.1 Asymmetric RAM Model [15]

The asymmetric RAM model is a model of computation that was introduced by Blelloch et al. [15]. The $(M, \omega)$-ARAM model consists of a small symmetric memory of size $M$, and an unbounded memory where reads have unit cost and writes have cost $\omega \gg 1$. The cost of an algorithm in this model is the number of reads in the large memory plus $\omega$ times the number of writes to the large memory.

1.3.2 Asymmetric-RAM Data Structures

In Chapter 6 we consider several graph problems in the Asymmetric-RAM model. We study breadth-first search, depth-first search and strongly connected components. In
each of the these problems the solution involves writing the solution out for each node which costs $\Theta(n)$ writes. There are existing algorithms for each of these three problems that have cost $\Theta(\omega n + m)$ [15]. In order to save writes, we instead want to be able to answer queries for each of these problems. We consider graphs with bounded degree $d$, meaning the maximum degree in the graph is $d$.

For breadth-first search, given a graph $G$ and source node $s$, we would like to be able to answer a query on a node for its BFS distance from $s$. For DFS, we would like to be able to answer a query for the discovery and finish time of a node. We assume there is an ordering on edges which gives us a particular DFS that we are simulating, otherwise the discovery and finish times can be inconsistent. Lastly, for strongly connected components, given a query of two nodes we would like to answer whether the two nodes are strongly connected.

For each of these three problems we build a sublinear size data structure that allows us to answer queries. The data structures are built on the idea of an implicit representation of the BFS or DFS labeling. By carefully selecting a sublinear number of nodes we can guarantee that a query node is near a recorded node from which it can get necessary information to answer the query. Our results are as follows. For BFS, we construct a data structure in $O(n\omega \log d / \log \omega)$ reads and $O(n \log d / \log \omega)$ writes which answers queries in $O(\sqrt{\omega})$ reads. For DFS, the cost of building the data structure is $O(n\sqrt{\omega} \log d / \log \omega)$ reads and $O(n \log d / \log \omega)$ writes. After building the data structure we can traverse the graph in DFS order with the same cost as building the data structure. The cost of answering a query is $O(\sqrt{\omega})$ reads and zero writes. For strongly connected components the cost of building the data structure is $O(n\sqrt{\omega} \log d / \log \omega)$ reads and $O(n \log d / \log \omega)$ writes, and the cost of answering a query is $O(\sqrt{\omega})$ reads and zero writes.
Chapter 2

Notation and Terminology

This chapter will provide some terminology and notation that will be used throughout this work.

2.1 Bounds

We use with high probability to mean that the probability of success is $1 - 1/n^c$, for any constant $c > 0$. We use soft-O notation, $\tilde{O}$, which builds on big-O notation, to suppress polylogarithmic factors. In particular, $\tilde{O}(f(n))$ is equivalent to $O(f(n) \log^c f(n))$ for some constant $c$.

2.2 Graphs

A graph $G = (V, E)$ is made up of a set of nodes $V$ and a set of edges $E$. A graph can be weighted or unweighted. In a weighted graph, there is a weight function $w : E \rightarrow \mathbb{R}$, and the weight of an edge $e$ is denoted $w(e)$. For an unweighted graph, the weight of each edge is one. For a graph $G = (V, E)$, we will generally use $n = |V|$ to denote the number of nodes, and $m = |E|$ to denote the number of edges. For a subset of nodes $V' \subseteq V$, the induced subgraph $G[V']$ on $V'$ is a graph consisting of all the nodes $V'$, and all the edges $(u, v)$ where $u \in V'$ and $v \in V'$.

A path $P = (v_0, v_1, ...v_\ell)$ is a sequence of vertices that are joined by edges i.e., $(v_i, v_{i+1}) \in E$, for all $i \in [0, \ell - 1]$. A path is simple if each node on the path is
unique. The weight or length of a path is the sum of weights of edges on the path
\[ w(P) = \sum_{i=1}^{\ell} w(v_{i-1}, v_i). \]
The size of a path is the number of edges on the path
\[ \text{size}(P) = \ell. \]

For nodes \( s \) and \( t \), the shortest path from \( s \) to \( t \) is the minimum weight path among all paths that start at \( s \) and end at \( t \). The shortest path from \( s \) to \( t \) is denoted \( \text{dist}_G(s, t) \), or \( \text{dist}(s, t) \) when \( G \) is clear from context. If there is no path from \( s \) to \( t \), then \( \text{dist}(s, t) = \infty \). A \((1 + \epsilon)\) approximate shortest path is a path from \( s \) to \( t \) where the distance of the approximate path \( d'(s, t) \) is such that \( \text{dist}(s, t) \leq d'(s, t) \leq (1 + \epsilon)\text{dist}(s, t) \).

A cycle is a path that starts and ends with the same vertex, and contains at least one edge. A directed graph with no cycles is called a directed acyclic graph (DAG).

For two nodes \( u \) and \( v \), if there is a path from \( u \) to \( v \), then we say that \( u \) is an ancestor of \( v \), and that \( v \) is a descendant of \( u \). The set of ancestors and descendants of node \( u \) are denoted \( \text{Anc}(u) \) and \( \text{Des}(u) \), respectively. Note that \( u \) is considered an ancestor and descendant of itself i.e., \( u \in \text{Anc}(u) \) and \( u \in \text{Des}(u) \).
Chapter 3

I/O-Efficient Topological Sort

This chapter is a reproduction of *I/O-Efficient Algorithms for Topological Sort and Related Problems* [21, 24], which is published in *Symposium on Discrete Algorithms 2019*, and *ACM Transactions on Algorithms 2022*. This dissertation includes an improvement of removing one $O(\log V)$ factor from the cost.

3.1 Introduction

Ullman and Yannakakis [59] and Chiang et al. [25] initiated the study of graph algorithms in the *I/O model* [2] over 20 years ago. Despite decades of research and many efficient algorithms for undirected graphs, there are essentially no I/O-efficient algorithms known for even the most basic problems on sparse directed graphs. Perhaps the most coveted is an algorithm for topologically sorting a directed acyclic graph (DAG). A topological sort of a DAG $G = (V, E)$ is an ordering of the vertices such that for every edge $(u, v) \in E$, $u$ precedes $v$ in the ordering.

This chapter presents the first algorithm for topologically sorting a DAG that is I/O-efficient even for sparse graphs. Not only is topologically sorting a fundamental problem on DAGs, but it is also a key subroutine in another general I/O-efficient technique known as time-forward processing [6, 25]. Due to the lack of a good general-purpose algorithm for topological sort, time-forward processing has only generated provably good results for restricted graph classes such as planar graphs [7, 38, 49].
3.1.1 Related Work

Topological sort. There are two classic linear-time algorithms for topological sort in the RAM model, either repeatedly peeling off the vertices with in-degree 0, or performing a depth-first search and outputting the vertices in reverse order of finish time [31]. The best I/O algorithms known are based on the depth-first search approach, for which there are two algorithms. Neither is efficient for sparse graphs.

Chiang et al. [25] provide an algorithm with I/O cost $O(V + \text{sort}(E) + \frac{VE}{MB})$, and Buchsbaum et al. [20] give an algorithm with I/O cost $O((V + \frac{E}{B}) \log(V/B))$. Both of these bounds include at least a cost of $|V|$, indicating that the algorithm may have to perform a random access or I/O for each vertex. For sparse graphs, notably $|E| = \Theta(V)$, both of these algorithms are worse than simply running an ordinary RAM DFS and paying an I/O for every operation.

Due to the lack of provably efficient algorithms for topological sort, some research has focused on engineering practically efficient algorithms [3, 63]. For example, Ajwani et al. [3] use an iterative approach that follows the same general strategy as our algorithm.

Related work beyond I/O algorithms. In the RAM model, strongly connected components (SCCs) can be identified in linear time [31, 57] by performing depth-first search.

Our algorithms shares some similarities with other topological-sort or SCC algorithms that perform recursive decompositions of the graph [28, 30, 37] instead of depth-first search. Coppersmith et al. [30] describe a randomized divide-and-conquer algorithm for computing the strongly connected components that runs in $O(E \log V)$ expected time in the RAM model. Cohen et al. [28] use a labeling scheme, which has a similar recursive structure, to solve an incremental topological sort where edges are
added to the graph over time. Fineman’s [37] parallel algorithm, which also starts from similar ideas, solves the static reachability problems with $O(E \cdot \text{poly}(\log V))$ work and $O(V^{2/3} \cdot \text{poly}(\log V))$ span/depth, with high probability.

The recursive structure of our topological-sort algorithm is most similar to that of Cohen et al. [28] in that the subproblems are defined by performing forward searches from each vertex. Like Fineman’s algorithm [37] but unlike the others, our algorithm performs the label propagation / graph search to a bounded distance, but the specific notions of distance are different. Many of the specific details, such as how distances are chosen, also resemble features in Fineman’s algorithm [37]. This fact should not be surprising given that there are relationships between parallel algorithms and I/O algorithms (see, e.g., [25] for discussion).

Though there are some similarities in the details between the parallel algorithm [37] and the I/O algorithm presented herein, these similarities are somewhat superficial; the primary challenges in each setting are actually quite different. Notably, our I/O efficient algorithm leverages time-forward processing, which is not efficient in the parallel model. In contrast, the parallel algorithm strongly exploits random accesses, which are not efficient in the I/O model.

Time-forward processing. Time-forward processing, originally described by Chiang et al. [25], is a technique that allows for efficient evaluation of circuit-like problems on DAGs. Each vertex (or edge) starts with some value $w(v)$ or $w(u, v)$. The goal is to compute some label $L(v)$ on each vertex, where $L(v)$ depends only on the initial values $w$ and the labels $\{L(u)|(u, v) \in E\}$ on $v$’s immediate predecessors. If the graph is topologically sorted, and certain technical restrictions are met on the function being computed, then the DAG circuit evaluation can be performed in $O(\text{sort}(E))$ I/Os by time-forward processing [6, 25]. The first solution [25] has additional restrictions on
the relative size of the cache, but Arge’s [6] solution removes those restrictions by solving the problem with an I/O-efficient priority queue called a Buffer Tree.

One challenging aspect about graph problems in the I/O model is that vertices cannot generally be processed one by one without sacrificing I/O efficiency. Instead, vertices must be processed (roughly speaking) in parallel by applying various sort and scan steps. Time-forward processing is useful in part because it simulates the effect of processing vertices one by one. Thus, information can propagate arbitrarily far in the graph, provided that the graph is topologically sorted.

3.1.2 Results

This chapter gives the following results, all having I/O cost $O(sort(E) \cdot \log^4 V)$, with high probability, on a graph $G = (V, E)$. For conciseness, we assume throughout that $|E| = \Omega(V)$.

• (Sections 3.2 and 3.3.) A randomized algorithm for topologically sorting a DAG.

• (Section 3.4.) A randomized algorithm for identifying and topologically sorting the strongly connected components (SCCs). Although this result subsumes topologically sorting a DAG, the algorithm includes additional complications and is thus presented separately.

• Using the topological sort algorithm coupled with time forward processing [6, 25] yields efficient solutions to other problems on DAGs, such as shortest paths, with the same I/O cost.

• Again applying time-forward processing [6, 25], the SCC algorithm implies a solution to the single-source reachability problem on directed graphs. Specifically, given a directed graph (not necessarily acyclic) and source vertex $s$, the
set of vertices reachable from $s$ can be be identified in $O(sort(E) \cdot \log^4 V)$ I/Os, with high probability.

3.1.3 Overview of the Approach

The general approach adopted here for a topological sort, loosely based on the IterTS algorithm described by Ajwani et al. [3], is as follows. Initially assign each vertex $v$ a label $L(v)$. Those labels induce an ordering over vertices. (For both our algorithm and IterTS, the labels correspond to a permutation of vertices, but in principle there could be ties.) Adopting the terminology from [3], an edge $(u, v)$ is satisfied if $L(u) < L(v)$ and violated otherwise. The goal is to adjust labels over time such that eventually all edges are satisfied.

To understand what makes the problem difficult, consider the following naive realization of the general strategy. Use $L_i(v)$ to denote the label of $v$ in round $i$. Initially assign all vertices $v$ the label $L_0(v) = 0$. In each round $i$, update every vertex $v$’s label to $L_i(v) = \max \{L_{i-1}(u) + 1 | (u, v) \in E\}$. (This type of update can be implemented by standard techniques, obtaining the updated label for all vertices via a constant number of sorts.) Although $v$’s label increases to ensure that $L_i(v) > L_{i-1}(u)$, the edge $(u, v)$ only becomes satisfied if $L_i(v) > L_i(u)$; if $u$’s label also increases during this round, then the edge may not be satisfied. In fact, with this algorithm the edge $(u, v)$ would only become satisfied during the round $\ell$ for $\ell$ equal to the length of the longest path to $u$. The end result is an algorithm with $O(V \cdot sort(E))$ worst-case I/O cost. Granted, this realization is particularly naive, but it seems difficult to beat. Indeed, IterTS [3], which applies heuristics to achieve good performance in practice, encounters this bottleneck.
Note that it is trivial to satisfy roughly half the edges immediately by randomly permuting all the vertices and labeling vertices by their rank in the permutation. The challenge is in improving the labeling beyond that point.

3.1.4 Algorithm Overview

An issue with the naive algorithm is that, in some sense, its label updates are too aggressive. Perhaps counter-intuitively, directly ensuring that \( L_i(v) > L_{i-1}(u) \) for all edges does not seem to lead to efficient algorithms. Instead, our algorithm temporarily gives-up on satisfying certain edges, which makes it easier to satisfy the other edges.

Our algorithm (described more fully in Section 3.2) performs the following type of recursive partitioning of the graph, somewhat inspired by [28]. Each vertex chooses a random priority value. That priority is propagated some (random) distance forward in the graph. Each vertex adopts the highest priority it has seen, with potentially many vertices adopting the same priority. (This step is performed in a way that ensures that the endpoints of already-satisfied edges remain in priority order.) Next, vertices are collected into groups where all vertices in the group have equal priority. The groups are ordered in increasing order of priority, and finally the algorithm recurses on the vertex-induced subgraphs for each group.

The analysis considers any particular violated edge \((u, v)\). The main claim is that in one full execution of this recursive algorithm, \((u, v)\) has at least a constant probability of becoming satisfied. Repeating the recursive algorithm a logarithmic number of times gives the high-probability result for all edges.

The proof itself is counter-intuitive but also simple in hindsight. Consider a particular violated edge \((u, v)\). Initially both \(u\) and \(v\) are in the same recursive subproblem. Ties on priority are good in the sense that they keep \(u\) and \(v\) in the same recursive subproblem. Eventually, at some recursive step, \(u\) and \(v\) adopt different priorities and
are placed in different recursive subproblems, which fixes the status of \((u, v)\) for the remainder of the execution; the edge becomes satisfied if \(u\)'s subproblem is ordered before \(v\)'s, and the edge is said to be *broken* if \(v\)'s subproblem is ordered before \(u\)'s. The two key components of the analysis are the following: (1) at each level of recursion, the probability that the edge becomes broken is proportional to \(1/K\), where \(K\) is the number of distances selected from, and (2) after enough levels of recursion, the edge is very likely to cross subproblem boundaries. By selecting distances randomly from a large-enough range, the probability of an edge becoming broken is low enough that the edge is likely to cross subproblem boundaries before it has too many chances of becoming broken. If the edge crosses subproblem boundaries but is not broken, then it must be satisfied.

Extension to strongly connected components. The extended algorithm propagates priorities both backwards and forwards, contracting groups of vertices reached in both directions. The analysis follows a similar framework, but the presence of cycles complicates various aspects of the algorithm and analysis.

3.1.5 Roadmap

The remainder of the chapter is organized as follows. Section 3.2 presents the algorithm for topological sort, and Section 3.3 analyzes that algorithm. Section 3.4 gives the algorithm for strongly connected components and its analysis.

3.2 Topological Sort Algorithm

This section describes the algorithm for topologically sorting a directed acyclic graph \(G = (V, E)\). The algorithm is analyzed in Section 3.3.
Algorithm 1 I/O Efficient Topological Sort

1: function TOPOLOGICALSORT\((G = (V , E ))\)
2: \hspace{1em} repeat until the vertices \(V\) are topologically sorted
3: \hspace{2em} RECURTS\((G, 1, |V|, 0)\)

4: function RECURTS\((G, i, j, depth)\) \(\triangleright\) Reorders the subarray \(V[i \ldots j]\) of vertices
5: \hspace{1em} \(G' = G[i \ldots j]\)
6: \hspace{2em} if depth \(\geq \lambda\) or \(i = j\) then return
7: \hspace{2em} \(d_{\text{max}} = (\lambda - \text{depth}) \cdot K\)
8: \hspace{2em} \(d_{\text{min}} = d_{\text{max}} - K\)
9: \hspace{2em} Choose \(d\) uniformly at random from \([d_{\text{min}}, d_{\text{max}}]\)
10: \hspace{2em} Choose a uniformly random permutation of priorities \(\{\rho(v)\}\)
11: \hspace{2em} For all \(v\), compute \(l(v) = \max \{\rho(u) : u \preceq_d v\ \text{in} \ G'\}\)
12: \hspace{2em} Sort vertices \(V[i \ldots j]\) lexicographically by \(\langle l(v) , \text{index}(v)\rangle\)
13: \hspace{2em} Partition \(V[i \ldots j]\) into maximal groups \([i_1, j_1], [i_2, j_2], \ldots, [i_t, j_t]\) of a single label \((i_r = j_{r-1} + 1)\)
14: \hspace{2em} for each \(r = 1\) to \(t\)
15: \hspace{3em} RECURTS\((G, i_r, j_r, \text{depth} + 1)\)

The graph is initially provided with the vertices in arbitrary order. As is typical for I/O algorithms, the graph representation is an array \(V\) of vertices and an array \(E\) of edges. The records of vertices and edges are as follows. Each vertex is represented by a unique ID, and each edge is represented by the IDs of its endpoints. Because the algorithm will sort the edge array many times, there need not be any assumption on the initial ordering or grouping of edges.

The goal of the algorithm is to gradually reorder vertices such that all edges are eventually satisfied, defined next. For each vertex, \(\text{index}(v)\) denotes the index of \(v\) in the vertex array, i.e., \(v = V[i]\) means \(\text{index}(v) = i\).

**Definition 3.2.1.** An edge \((u, v) \in E\) is **satisfied** if the \(\text{index}(u) < \text{index}(v)\) in the current vertex ordering. Otherwise, \((u, v)\) is **violated.**
The algorithm is designed to ensure that once an edge becomes satisfied, it remains satisfied for the rest of the execution.

Algorithm 1 presents a high-level description of the algorithm, ignoring the low-level details necessary to transform the algorithm to an I/O-efficient realization. The main algorithm topologically sorts the graph by performing a sequence of executions of a recursive algorithm, called \textsc{RecurTS}. The goal with each execution of the recursive algorithm is to reorder vertices to satisfy some, ideally a constant fraction, of the violated edges. The main algorithm terminates when all edges have been satisfied, i.e., when the vertices in $V$ are in topological-sort order. Section 3.2.1 describes \textsc{RecurTS} in more detail, and Section 3.2.2 briefly describes how to make \textsc{RecurTS} I/O efficient.

3.2.1 The Recursive Algorithm

At a high level, the recursive algorithm chooses random priorities for each vertex, propagates the priorities some distance in the graph, reorders vertices according to the highest priority observed, and finally recurses on subgraphs induced by vertices of the same priority. Before describing the algorithm in more detail, we first clarify the notion of distance adopted by the algorithm.

Distances. All distances discussed in this chapter are with respect to the number of violated edges, i.e., interpreting violated edges as having weight 1 and satisfied edges as having weight 0. If there exists a path from $u$ to $v$ that includes at most $d$ violated edges, then we say $u$ can reach $v$ at distance $d$, denoted $u \preceq_d v$. We also say that $u$ is a $d$-hop predecessor of $v$.

The relation $\preceq_\infty$ is the standard notion of reachability, and when the vertices are in topological-sort order $\preceq_0$ and $\preceq_\infty$ are equivalent. Note that unlike $\preceq_\infty$, when $d$
is a finite fixed distance $\leq d$ is not transitive; however, $x \leq_{d_1} y$ and $y \leq_{d_2} z$ implies $x \leq_{d_1+d_2} z$.

Vertex-induced subgraphs. Each recursive call operates on a contiguous subarray of vertices and the subgraph induced by those vertices. We use $G[i \ldots j]$ to denote the vertex-induced subgraph of $G$, induced by vertices in $V[i \ldots j]$.

Global parameters. The algorithm is parameterized by $K$ and $\lambda$. The value $\lambda$ specifies the maximum recursion depth, which will be discussed at the end of this subsection. The value $K$ specifies the number of possible distances from which to select a random distance. There is a tradeoff here. Choosing larger $K$ decreases the probability of an edge becoming broken, thereby increasing the number of edges that become satisfied. On the other hand, larger $K$ also leads to higher I/O cost. A good value is selected in Section 3.3.

The algorithm. For now, ignore the recursion depth, $\lambda$, and the specific range of distances. The algorithm RECURTS($G, i, j, \text{depth}$) operates on the induced subgraph $G[i \ldots j]$ as follows. Choose a distance $d$ uniformly at random from a contiguous range of $K$ possible distances. Assign each vertex $v$ a distinct random priority $\rho(v)$. For each $v$, let $l(v)$ denote the highest priority from among $v$’s $d$-hop predecessors, i.e., $l(v) = \max \{ \rho(u) : u \preceq_d v \text{ in } G[i \ldots j] \}$. Sort the vertices by $l(v)$ using a stable sort. This is the only place in the algorithm where vertices are reordered and edges may become satisfied. At this point, vertices with the same label $l$ are grouped together into contiguous subarrays, and the groups are sorted by label. Finally, recurse on each group.

Figure 3.1 illustrates an example of a single level of recursion. In the figure, the distance used is $d = 2$. The three subfigures illustrate (3.1a) the initial graph and
Figure 3.1: An example of a single level of recursion for Algorithm 1, with $d = 2$. Each of the three subfigures shows two equivalent images of the same graph, with the bottom image displaying the current vertex ordering from left to right. Vertices are labeled alphabetically by their initial ordering. Solid arrows represent satisfied edges (i.e., those edges directed to the right in the vertex-ordered graph) and dashed arrows represent violated edges (those edges directed to the left). The number over each vertex is either its random priority (in 3.1a) or its label (in 3.1b and 3.1c).
vertex ordering, (3.1b) the labels assigned to vertices after propagating priorities to a distance of \( d = 2 \), and (3.1c) the new ordering on vertices and the recursive subproblems. After reordering vertices here according to label, two previously violated edges, namely \((D, C)\) and \((J, I)\), become satisfied. Notice that only some of the edges crossing subproblem boundaries transition from violated to satisfied, and no edges change from satisfied to violated.

Distance ranges and maximum recursion depth. One component of the analysis (Section 3.3) is that the number of \( d \)-hop predecessors of \( v \) decreases with each level of recursion. This progress argument, however, is with respect to the specific distance, and it seems difficult to argue anything about the number of \( d' \)-hop predecessors for \( d' > d \). On the other hand, to argue that edges are unlikely to be broken, distances need to be selected randomly from \( K \) possibilities. To reconcile these two issues, the range of distances depends on the level of recursion, decreasing with each level. Moreover, since distances should always be positive, the distance used at recursion depth 0 places a limit on the number of levels of recursion that the algorithm can support.

Putting these ideas together, we have the following. If a call is made with recursion depth \( depth \geq \lambda \), then the algorithm simply returns. Otherwise, the distance \( d \) is selected uniformly at random from the range \([d_{\min}, d_{\max})\), where \( d_{\min} = d_{\max} - K \) and \( d_{\max} = (\lambda - depth) \cdot K \).

3.2.2 Achieving I/O Efficiency

This section describes how to make RecurTS I/O efficient, all of which is fairly straightforward. We first describe the implementation with respect to the initial call to RecurTS on the entire graph. We later describe how to implement the recursion.
Each vertex and edge record is augmented with a constant amount of additional information, so that the total space of the vertex and edge arrays is still $O(V)$ and $O(E)$, respectively. The standard technique for transferring information along edges is by performing sorts and scans of the vertex and edge arrays. These sorts should be viewed as transient, unlike the sort explicitly given in Algorithm 1 whose goal is to produce the topological sort.

First, tag each vertex $v$ with its index $\text{index}(v)$, which can be achieved by a single scan (that is, iterating in order) of the vertex array. Next tag each edge by the indices of its endpoints. This edge-tagging step can be accomplished by sorting both the vertex array by vertex ID and sorting the edge array by the ID of one endpoint. Then perform simultaneous scans of the vertex array and edge array, synchronizing the scans on ID, and copying the index of the vertex to the edges with matching endpoint ID. To store the index of the other endpoint, sort by the other endpoint. The cost of these steps is $\Theta(\text{sort}(V) + \text{sort}(E))$ for sorting the arrays and $O(\text{scan}(V) + \text{scan}(E))$ for iterating over them. The sort bound dominates.

To assign a permutation of priorities, simply select random numbers in the range $1, 2, \ldots, |V|^c$ for each vertex, where $c \geq 2$ is a constant that controls failure probability. Sort the vertices by priority and perform a scan to verify that all priorities are distinct. Repeat this process until the priorities are distinct.

Propagating priorities. The most difficult aspect is implementing the label $l(v) = \max \{ \rho(u) : u \succeq_d v \}$. This is achieved incrementally through a sequence of propagation steps. Initially set $l(v) = \rho(v)$ and perform an update called satisfied-edge propagation. Next, perform $d$ rounds, each including violated-edge propagation followed by satisfied-edge propagation. There are thus $2d + 1$ propagation steps in total.
Before describing how to implement the two types of propagation steps, let us first discuss the goal of each type of update. Let \( l(v) \) and \( l'(v) \) denote \( v \)'s labels at the start and end, respectively, of a single propagation step (satisfied or violated). The goal of satisfied-edge propagation is to update the label to \( l'(v) = \max \{ l(u) : u \preceq_0 v \} \), i.e., propagate the label arbitrarily far but along satisfied edges only. The goal of violated-edge propagation is to update the label to \( l'(v) = \max \{ l(v), \max \{ l(u) : (u, v) \in E \} \} \), i.e., propagate the label along a single hop that is allowed to be violated.

We now argue that the sequence of propagation steps gives each vertex the intended label.

**Lemma 3.2.2.** After \( d + 1 \) satisfied-edge propagation steps interleaved with \( d \) violated-edge propagation steps, we have \( l(v) = \max \{ \rho(u) : u \preceq_d v \} \) for all \( v \in V \).

**Proof.** The proof is by induction on \( d \). The base case is \( d = 0 \), meaning that we are considering the result of the first satisfied-edge propagation. Initially, \( l(u) = \rho(u) \) for all \( u \in V \). The propagation step then updates the labels to \( l'(v) = \max \{ l(u) : u \preceq_0 v \} = \max \{ \rho(u) : u \preceq_0 v \} \), which satisfies the claim for \( d = 0 \).

For the inductive step, assume that \( l(x) = \max \{ \rho(u) : u \preceq_{d-1} x \} \) for all \( x \in V \) after the first \( d - 1 \) rounds, and consider the effect of performing one more round consisting of a violated-edge propagation step followed by a satisfied-edge propagation step. The violated-edge propagation updates the labels to \( l'(y) = \max \{ l(x) : (x, y) \in E \} \). By the inductive assumption on \( l(x) \), this reduces to \( l'(y) = \max \{ \rho(u) : u \preceq_{d-1} x, (x, y) \in E \} \). The subsequent satisfied-edge propagation step updates the labels once more to \( l''(v) = \max \{ l'(y) : y \preceq_0 v \} \), which when substituting in \( l'(y) \) gives \( l''(v) = \max \{ \rho(u) : u \preceq_{d-1} x, (x, y) \in E, y \preceq_0 v \} \). Observing that any path containing \( d \) violated hops can be broken down into a path of \( d - 1 \)
violated hops followed by a single violated edge followed by a satisfied path, as in the preceding expression, completes the proof.

Implementing satisfied-edge propagation. The satisfied edges $E_{sat}$ can be identified by scanning through the edge set and identifying those edges $(u, v)$ with $\text{index}(u) < \text{index}(v)$. Note that when $V$ is sorted by index, the graph $G_{sat} = (V, E_{sat})$ is topologically sorted, which is important as we shall apply time-forward processing.

In more detail, performing the update $l'(v) = \max\{l(u) : u \preceq_0 v \text{ in } G\}$ is equivalent to computing $l'(v) = \max\{l(u) : u \preceq_\infty v \text{ in } G_{sat}\}$. The following is a simple sequential algorithm for computing the updated label with regards to $G_{sat}$. Consider the vertices $v$ in $G_{sat}$ in topological-sort order; update $v$'s label to $\max\{l(v), \max\{l(u) : (u, v) \in E_{sat}\}\}$, i.e., the maximum of its old value and the value on all immediate predecessors. This local-update rule is exactly the kind that can be implemented I/O-efficiently using time-forward processing [6, 25], assuming $G_{sat}$ is topologically sorted.

Time forward processing. This section describes the algorithm for time forward processing from [6, 25]. The algorithm takes as input a directed acyclic graph with the vertices in topologically sorted order. We assume the edges are given in a list in no particular order to start. Each vertex $v$ also has a priority $\rho(v)$. The goal is to compute $f(v)$ for each vertex where $f(v) = \max(p(v), f(u))$ for each $(u, v)$ in $E$, where $f(u)$ is defined recursively. We use an external memory priority queue, which has amortized cost $O(1/B \log_{M/B} N/B)$ I/Os for insert, delete and extract min [6].

The vertices are processed in topological order. To process a vertex $v$, first compute $f(v)$, which will be described later. Then for each outgoing edge from $v$, $(v, w) \in E$, insert $(v, w)$ with key $w$, i.e. the position of $w$ in topologically sorted order, into the
priority queue augmented with \( f(v) \). Notice that each edge is inserted into the priority queue once, and each vertex has an element in the priority queue for each incoming edge it has. To compute \( f(v) \), we must extract min once for each incoming edge that \( v \) has. Then set \( f(v) \) to be the maximum of all \( f(w) \) augmented to each incoming edge \((w, v)\), and \( \rho(v) \).

The edges are inserted in topological order of the source vertex. By sorting the edge list by source vertex, this allows for one scan of the edge list to insert all the edges into the priority queue. The number of inserts to the priority queue is the number of edges \( O(E) \). The number of extract mins is also the number of edges \( O(E) \). In total this is \( O(E/B \log_{M/B} E/B) \) I/Os.

Implementing violated-edge propagation. This step can be accomplished by sorting and scanning. In particular, first sort the edges \((u, v)\) by \( \text{index}(u) \) so that all outgoing edges for a particular vertex \( u \) are consecutive. Then scan through the vertices and edges simultaneously, synchronizing on the vertex index. Attach to the edge \((u, v)\) the priority \( l(u, v) = l(u) \). Next, sort the edges \((u, v)\) by \( \text{index}(v) \). Now the incoming edges for each vertex \( v \) are consecutive. Finally, scan through the edges and vertices simultaneously, and for each \( v \) update \( l'(v) = \max \{ l(v), \max \{ l(u) : (u, v) \in E \} \} \).

Implementing the recursion. To slightly simplify the analysis of the I/O cost, it is convenient to reason about the algorithm as performing the recursion level by level.\(^1\) That is, do not actually make multiple recursive calls. Instead, perform the algorithm as described for the entire level at once. The only additional bookkeeping necessary is to delimit the boundaries between each recursive subproblem in the vertex array.

\(^1\)The issue is that there are no bounds on the relative sizes of a problem and its recursive subproblems — a graph that fits in cache may be partitioned into subgraphs that are much smaller than a block. If considering each recursive subproblem one at a time, the analysis would have to be careful about the accounting of these small subproblems.
Each level of recursion can then be implemented by first scanning through the vertex array and tagging each vertex with a subproblem ID (increasing by one when crossing each subproblem boundary) and similarly tagging the edges with the subproblem IDs of its endpoints. All edges whose endpoints have different subproblems should be ignored in all steps. Whenever sorting vertices by label or priority, the subproblem ID should also be taken into account as the most significant feature in the sort. (That is, sort lexicographically by subproblem ID, then label/priority.) The other details are unchanged.

We can now analyze the I/O cost of the recursive algorithm. Assuming a minimum constant size on the cache is necessary to implement a constant number of synchronized scans in \(O(\text{scan}(N))\) I/Os. There are also similar cache-size assumptions in time-forward processing [6] (which are not highlighted in those theorem statements) that would carry-over to this setting. The following theorem comes from Theorem 3 and Section 4.1 in [6].

**Theorem 3.2.3** (From [6]). There exists a constant \(\delta\) such that if there is a cache with at least \(\delta\) blocks, then given a directed acyclic graph in topologically sorted order, time forward processing can be performed in \(O(\text{sort}(E))\) I/Os.

**Lemma 3.2.4.** There exists a constant \(\delta\) such that the following holds: if the cache contains at least \(\delta\) blocks, then a single execution of \textsc{RecurTS} has I/O cost \(O(K\lambda^2\text{sort}(E))\), with high probability. The \(K \geq 1\) and \(\lambda \geq 1\) here are the global parameters of the algorithm.

**Proof.** Each of the \(\lambda\) levels of recursion performs \(d\) rounds of label propagation, where \(d\) is at most \(\lambda K\). Each round of label propagation can be implemented in \(O(\text{sort}(E))\) I/Os. All other steps of the algorithm can also be accomplished in a constant number of scans and sorts, except assigning priorities performs a single attempt with high
probability. (The failure probability depends on the range of priorities.) The cost of each level of the $\lambda$ levels of recursion is thus $O(\lambda \text{sort}(E))$. \hfill $\square$

### 3.3 Topological Sort Analysis

The section analyzes the topological-sort algorithm given in Section 3.2. The goal is to show that, with high probability, the main algorithm completes after $O(\log V)$ executions of `RecurTS`. The key component toward achieving this goal is to show that in each execution, each violated edge has a constant probability of becoming satisfied. The bulk of this section is devoted to proving this claim. Given the claim, it is simple to show that $O(\log V)$ executions suffice.

Consider any violated edge $(u, v)$ and an execution of `RecurTS`. The most important point of the execution is the moment, if any, that $u$ and $v$ receive different priorities and are hence placed in different recursive subproblems. This step is the only time during the execution that the relative order of $u$ and $v$ may change. If $u$ is ordered before $v$, then the edge becomes satisfied. If $u$ remains ordered after $v$, however, then the edge $(u, v)$ cannot become satisfied for the remainder of the execution (i.e., until the next execution of `RecurTS`).) The following definition captures this bad outcome:

**Definition 3.3.1.** An edge $(u, v)$ is **broken** if (i) $\text{index}(v) < \text{index}(u)$, i.e., $v$ precedes $u$ in the ordering of vertices, and (ii) $u$ and $v$ are in different recursive subproblems.

In Figure 3.1c, the broken edges are $(L, F)$, $(I, H)$, $(O, A)$, and $(O, N)$.

As outlined in Section 3.1.4, our analysis consists of two main components. First, we argue that, for large enough $K$, an edge $(u, v)$ has at most constant probability of becoming broken during an execution of `RecurTS`. Second, we argue that for large enough $\lambda$, the execution is likely to terminate with $u$ and $v$ in different subproblems.
If \( u \) and \( v \) are in different subproblems, and the edge is not broken, then it must be satisfied. The remainder of the section focuses on proving each of these claims.

Predecessors. Throughout the analysis, it is useful to refer to the set of predecessors of a particular vertex. Let \( G = (V, E) \) be a graph, let \( v \in V \) be a vertex in the graph, and let \( d \) be a distance. We define the \( d \)-hop predecessors of \( v \), denoted by \( A(G, v, d) \), as \( A(G, v, d) = \{ x : x \preceq_d v \text{ in } G \} \).

3.3.1 Bounding the Probability of an Edge Becoming Broken

We argue that in any level of recursion, a particular violated edge \((u, v)\) has probability at most \( O(\log V/K) \) of becoming broken. Taking a union bound across all \( \lambda \) levels of recursion gives a probability of at most \( O(\lambda \log V/K) \) that the edge becomes broken across the entire execution of \( \text{RecurTS} \). Setting \( K = \Omega(\lambda \log V) \) and tuning constants appropriately, the probability that \((u, v)\) becomes broken is upper bounded by a constant.

We begin by considering how an edge can become broken. The following lemma implies that an edge \((u, v)\) can become broken only if \( u \)'s highest-priority \( d \)-hop predecessor is located exactly \( d \) violated hops away.

**Lemma 3.3.2.** Consider a call \( \text{RecurTS}(G, i, j, \text{depth}) \). Let \( G' = G[i..j] \) denote the induced subgraph, let \((u, v)\) be an edge in \( G' \), and let \( d \) be the random distance chosen. Finally, let \( x \) denote the vertex in \( A(G', u, d) \) with the highest priority \( \rho(x) \). If \( x \in A(G', u, d - 1) \), then \( l(u) \leq l(v) \).

**Proof.** Suppose \( x \in A(G', u, d - 1) \). Then we have \( x \preceq_{d-1} u \) and \( u \preceq_1 v \), giving \( x \preceq_d v \). It follows that \( l(v) \geq \rho(x) \). Moreover, since \( x \) is the vertex with highest priority among \( u \)'s \( d \)-hop predecessors, \( l(u) = \rho(x) \). We thus have \( l(v) \geq \rho(x) = l(u) \). \( \square \)
We next bound the probability that an edge \((u, v)\) becomes broken in a particular recursive call. Consider the random process as follows. First choose a random distance. Then identify which vertex, from among the \(d\)-hop predecessors, has highest priority. Specifically, determine if the highest-priority predecessor is also a \((d - 1)\)-hop predecessor; if so, by the previous lemma the edge does not break. The probability of the edge breaking thus depends on the relative sizes of the \(A(G', u, d)\) and \(A(G', u, d - 1)\).

The main idea is therefore to characterize distances by relative neighborhood sizes. The argument is roughly as follows, but the following lemma provides a tighter bound. A distance \(d\) is “bad” if at least a \(1/\log V\)-fraction of the \(d\)-hop predecessors are at distance exactly \(d\), i.e., not also \((d - 1)\)-hop predecessors. If a bad distance is selected, the probability of the edge breaking may be high. Fortunately, due to the expansion implied by bad distances, there cannot be too many bad distances — specifically only \(O(\log^2 V)\) of them. If a good distance is selected, the probability that the edge breaks is at most \(O(1/\log V)\). Putting these together, the probability that the edge breaks is \(O(\log^2 V/K + 1/\log V)\). The next lemma improves this by more carefully accounting for how bad each distance is.

Recall that at level of recursion \(\text{depth}\), \(d_{\text{max}} = (\lambda - \text{depth})K\) and \(d_{\text{min}} = d_{\text{max}} - K\).

**Lemma 3.3.3.** Consider a call \(\text{RecurTS}(G, i, j, \text{depth})\). Let \(G' = G[i\ldots j]\) denote the induced subgraph, let \((u, v)\) be a violated edge in \(G'\), and let \(s_d = |A(G', u, d)|\) denote the number of \(d\)-hop predecessors of \(u\) in \(G'\), for each possible \(d \in [(\lambda - \text{depth} - 1)K - 1, (\lambda - \text{depth})K]\). Then the probability that the edge becomes broken during this call is at most \((1/K) \cdot \log(s_{(\lambda - \text{depth})K - 1}/s_{(\lambda - \text{depth} - 1)K - 1})\).

**Proof.** Let \(B\) denote the event that the edge \((u, v)\) is broken. Let \(d\) denote the random distance chosen, and let \(x\) be the vertex in \(A(G', u, d)\) with highest priority. By Lemma 3.3.2, \(\Pr [B] \leq \Pr [x \notin A(G', u, d - 1)]\), so it suffices to bound the latter.
Define $\gamma_d = s_{d-1}/s_d$ to be the fraction of $u$’s $d$-hop predecessors that are also $(d-1)$-hop predecessors.

Let $E_d$ denote the event that distance $d$ is chosen. Once $d$ is fixed, we trivially have $\Pr[B|E_d] \leq 1 - \gamma_d$. Since the distance is chosen uniformly at random from $K$ possibilities, we have

$$\Pr[B] = \sum_{d=d_{\min}}^{d_{\max}-1} (\Pr[B|E_d] \cdot \Pr[E_d]) \leq \sum_{d=d_{\min}}^{d_{\max}-1} (1 - \gamma_d)/K.$$ 

Since $\gamma_d \in (0, 1]$, and for this range $\lg(1/\gamma_d) \geq 1 - \gamma_d$,

$$\Pr[B] \leq \frac{1}{K} \sum_{d=d_{\min}}^{d_{\max}-1} (1 - \gamma_d) \leq \frac{1}{K} \sum_{d=d_{\min}}^{d_{\max}-1} \lg(1/\gamma_d).$$ 

Next we have

$$\sum_{d=d_{\min}}^{d_{\max}-1} \lg(1/\gamma_d) = \lg\left(\frac{s_{d_{\max}-1}}{s_{d_{\min}-1}}\right).$$ 

Therefore,

$$\Pr[B] \leq \frac{1}{K} \sum_{d=d_{\min}}^{d_{\max}-1} \lg(1/\gamma_d) \leq \frac{1}{K} \lg\left(\frac{s_{d_{\max}-1}}{s_{d_{\min}-1}}\right)$$ 

and substituting in the values for $d_{\min}$ and $d_{\max}$ gives the proof. 

3.3.2 Bounding the Probability that An Edge Crosses Subproblems

The second key component of the analysis is to argue that at the end of an execution, the edge $(u, v)$ is likely to cross subproblem boundaries. To achieve this goal, we argue that with each level of recursion, $v$ is likely to lose a constant fraction of its nearby predecessors. Thus, with $\Omega(\log V)$ levels of recursion, it is very likely that $v$ has no predecessors. If $v$ has no predecessors, then $u$ must be in a different subproblem.

Definitions. More formally, consider the calls $\text{RECURTS}(G, i, j, \ell)$ arising during the execution of the recursive algorithm, where $\ell$ here denotes level or depth of the recursion. If $v \in G[i..j]$, we call $G_v^\ell = G[i..j]$ the level-$\ell$ graph of $v$. Notice that $v$ belongs
to at most one subproblem at each level of recursion. If \( v \) does not belong to any level-\( \ell \) subproblems (i.e., if the base case was reached early), then \( G_v^\ell \) is the empty graph. Thus, \( v \) has a corresponding sequence \( G_v^0, G_v^1, \ldots, G_v^\lambda \) of level-0, 1, \ldots, \( \lambda \) graphs, where \( G_v^0 \supseteq G_v^1 \supseteq \cdots \supseteq G_v^\lambda \).

For this subsection, the important feature is the number of nearby, proper predecessors \( v \) has at each level of recursion. A vertex \( x \) is a level-\( \ell \) active predecessor of \( v \) if \( x \neq v \) and \( x \preceq d_{\text{max}} \) in \( G_v^\ell \), where \( d_{\text{max}} = K(\lambda - \ell) \) is the maximum distance for this level of recursion. Notice that \( v \) is not an active predecessor of itself.

### 3.3.3 Reducing the Number of Active Predecessors.

We start with a simple observation, captured by the first lemma: no new relationships are created between vertices as the algorithm recurses. Thus, we need not worry about the set of active predecessors growing — the only challenge is to show that a significant fraction of the predecessors are likely to be knocked out.

**Lemma 3.3.4.** Consider any vertex \( v \) and its level-(\( \ell - 1 \)) and level-(\( \ell \)) subgraphs \( G_v^{\ell - 1} \) and \( G_v^\ell \), respectively. For any vertex \( x \) and distance \( d \), if \( x \npreceq_d v \) in \( G_v^{\ell - 1} \), then \( x \npreceq_d v \) in \( G_v^\ell \).

**Proof.** At first glance, this statement sounds obvious given that edges are never created. There is, however, one concern — satisfying edges can decrease distances between vertices. Since the relative order of vertices only changes when those vertices are placed in different recursive subproblems, an edge can only be satisfied in \( G_v^\ell \) if it is already satisfied in \( G_v^{\ell - 1} \).

We are now ready to argue that the number of active predecessors is likely to decrease at each level of recursion. This proof leverages only the random priorities —
the fact that distances are chosen randomly is not important. The proof (notably the second claim therein) lifts some ideas from [37, Lemma 3.4].

**Lemma 3.3.5.** Consider any vertex $v$ and level $\ell$ of recursion. Let $\alpha$ and $\alpha'$ denote the number of level-$\ell$ and level-$(\ell + 1)$, respectively, active predecessors of $v$. Then $\Pr[\alpha' \leq (3/4)\alpha] \geq 1/3$.

**Proof.** If $\alpha = 0$, the claim is trivial. Otherwise, consider the level-$\ell$ call to RECURTS on graph $G^\ell_v$. Let $d$ be the distance selected (which need not be random for the purpose of the proof). Let $A_d = A(G^\ell_v, v, d)\setminus \{v\}$ denote the set of $d$-hop predecessors of $v$ in $G^\ell_v$, excluding $v$ itself, and let $a_d = |A_d|$. Notice by Lemma 3.3.4 and the decreasing distance ranges, the level-$(\ell + 1)$ active predecessors are a subset of $A_d$. Moreover, $A_d$ is a subset of the level-$\ell$ active predecessors. It is thus sufficient to argue that with probability at least $1/3$, at most $3a_d/4 \leq 3\alpha/4$ of the vertices in $A_d$ are also in $G^\ell_{v+1}$.

Let $x$ be a random variable denoting the highest-priority vertex in $A_d \cup \{v\}$. For any other vertex $y \in A_d$, we say that $x$ knocks out $y$ if $x \not\preceq_d y$. The remainder of the proof amounts to proving the following two claims: (1) If $x$ knocks out $y$, then $y \not\in G^\ell_{v+1}$, and (2) With probability at least $1/3$, $x$ knocks out at least $a_d/4$ vertices from $A_d$.

Claim 1. Recall that $x$ is the highest-priority vertex from $A_d \cup \{v\}$. Thus, $v$ inherits the label $l(v) = \rho(x)$, which defines its subproblem. If $x \not\preceq_d y$, then $l(y) \neq \rho(x)$, and $y$ is in a different subproblem from $v$.

Claim 2. Because the graph is acyclic, for any pair $x \neq y$ of vertices, at least one of the following must be true: $x \not\preceq_d y$ or $y \not\preceq_d x$. Moreover for all $y \in A_d$, $y \preceq_d v$ by definition, so $v \not\preceq_d y$. Thus the total number of pairs $x \in A_d \cup \{v\}$ and $y \neq x \in A_d$ for which $x$ knocks out $y$ must be at least $\binom{a_d}{2} + a_d = a_d(a_d + 1)/2$. Because $x$ is selected from $a_d + 1$ choices, the expected number of vertices knocked out by $x$ is at
least $a_d/2$. By Markov's inequality, the probability that at least $(3/4)a_d$ vertices are not knocked out is therefore at most $2/3$.

Lemma 3.3.5 indicates that with each level of recursion, the number of active predecessors is likely to decrease by a constant factor. The following lemma says that after enough levels of recursion, $v$ is likely to have no remaining active predecessors. The implication is that all of its incoming edges cross subproblem boundaries.

Lemma 3.3.6. Consider any vertex $v$ and a complete execution of the recursive algorithm. With probability at least $1 - 8 \log |V| / \lambda$, $v$ has no active predecessors at the $\lambda$-th level of recursion.

Proof. Let $X_j$ be a random variable denoting the number of levels $\ell$ for which the number of active predecessors of $v$ falls in the range $[(3/4)^j + 1, (3/4)^j \cdot |V|]$, for integer $j$. When $j \geq \log_{4/3} |V|$, the high end of the range is strictly less than $1/|V| \cdot |V| = 1$, meaning that $v$ has no active predecessors remaining. Let $X = \sum_{j < \log_{4/3} |V|} X_j$. If $X < \lambda$, then at or before the the $\lambda$-th level of recursion, $v$ has no active predecessors. Our goal is thus to argue that this event is likely to occur.

Lemma 3.3.5 implies that the number of rounds necessary to get a $(3/4)$ reduction is at most 3 in expectation. Thus $E[X_j] \leq E[X_j | X_j \geq 1] \leq 3$. By linearity of expectation, $E[X] \leq 3 \log_{4/3}(|V|) < 8 \log(|V|)$. By Markov's inequality, $\Pr [X \geq \lambda] \leq 8 \log(|V|)/\lambda$.

3.3.4 Edges are Likely to Become Satisfied

Thus far, we have argued that edges are unlikely to become broken at any particular level of recursion, and that edges are likely to cross subproblem boundaries by the time the recursive algorithm terminates. This section combines those pieces to conclude
that in a single execution of the recursive algorithm, a violated edge is likely to become satisfied.

Before getting to the main claim, we first observe that satisfied edges stay satisfied. This fact is important both to argue that a violated edge is likely to become satisfied in a single execution, and to argue that multiple executions lead to monotonic progress.

**Lemma 3.3.7.** Consider an execution of the recursive algorithm \textsc{RecurTS}. If an edge \((u, v)\) is satisfied at the \(\ell\)-th level of recursion, then it is satisfied at all subsequent levels of recursion.

**Proof.** Proof by induction on the level of recursion. Consider the call at the \(\ell\)th level of recursion, and suppose that the edge \((u, v)\) is satisfied at the start of the call. The goal is to show that it remains satisfied in the next recursive subproblem. Note that if \((u, v)\) is satisfied at the start of the call, then \(u \preceq_0 v\) in the current graph. Let \(l(u)\) be the final label on vertex \(u\), and let \(x\) be the vertex such that \(l(u) = \rho(x)\). Then \(x \preceq_d u\), which coupled with \(u \preceq_0 v\) implies that \(x \preceq_d v\). It follows that \(l(v) \geq l(u)\).

If \(l(v) = l(u)\), then \(u\) and \(v\) maintain their current ordering. If \(l(v) > l(u)\), then \(v\) is placed in an even later subproblem. Either way, \((u, v)\) remains satisfied. \(\square\)

**Lemma 3.3.8.** Let \((u, v)\) be an edge that is initially violated, and consider a complete execution of \textsc{RecurTS}(\(G, 1, |V|, \text{depth}\)). Then the probability that \((u, v)\) becomes broken in this execution is at most \(\lg(|V|)/K\).

**Proof.** Let \(s_d^{(i)}\) be the number of \(d\)-hop predecessors of \(u\) in the \(i\)th level of recursion.

By Lemma 3.3.3 the probability that an edge gets broken in the \(i\)th level of recursion is \((1/K) \cdot \lg(s_d^{(i)}_{\lambda-iK-1}/s_d^{(i)}_{\lambda-iK-K-1})\). Let \(B\) be the event that \((u, v)\) gets broken in any level of recursion. Summing across all the levels we have,
\[ \Pr [ B ] = \sum_{i=0}^{\lambda-1} \left( \frac{1}{K} \lg \left( \frac{s^{(i)}_{(\lambda-i)K-1}}{s^{(i)}_{(\lambda-i-1)K-1}} \right) \right) \]

\[ = \left( \frac{1}{K} \right) \sum_{i=0}^{\lambda-1} \left[ \lg \left( s^{(i)}_{(\lambda-i)K-1} \right) - \lg \left( s^{(i)}_{(\lambda-i-1)K-1} \right) \right]. \]

By Lemma 3.3.4, the number of related nodes at a distance \( d \) cannot increase across levels of recursion \( s^{(i+1)}_d \leq s^{(i)}_d \), and therefore,

\[ \Pr [ B ] \leq \left( \frac{1}{K} \right) \lg \left( \frac{s^{(0)}_{K-1}}{s^{(0)}_0} \right) \leq \left( \frac{1}{K} \right) \lg |V|. \]

Since \( s^{(0)}_{K-1} \leq |V| \), \( \Pr [ B ] \leq \left( \frac{1}{K} \right) \lg |V| \).

**Lemma 3.3.9.** Let \((u, v)\) be any edge, and consider a complete execution of RECURTS with parameters \( \lambda \geq 32 \lg V \) and \( K \geq 4 \lg V \). If \((u, v)\) is violated initially, then with probability at least \( 1/2 \), \((u, v)\) is satisfied at the end of the execution. If \((u, v)\) is satisfied initially, then with probability \( 1 \) it is still satisfied at the end.

**Proof.** By Lemma 3.3.7, a satisfied edge always remains satisfied. The remainder focuses on the case that \((u, v)\) is initially violated.

Let \( A \) be the event that \((u, v)\) is violated at the end of the execution. Let \( B \) be the event that the edge breaks at some level of recursion, and let \( C \) be the event that the two endpoints \( u \) and \( v \) are in the same level-\( \lambda \) subproblem. If neither \( B \) nor \( C \) occurs, then the edge crosses properly ordered subproblems and the edge is satisfied.

We thus have \( \Pr [ A ] \leq \Pr [ B ] + \Pr [ C ] \) by a union bound.

By Lemma 3.3.6, \( \Pr [ C ] \leq 8 \lg V/\lambda \leq 1/4 \) for the specified choice of \( \lambda \). By Lemma 3.3.8, \( \Pr [ B ] \leq \lg V/K \leq 1/4 \), for the specified choice of \( K \). Adding these together gives total failure probability of at most \( 1/2 \). \( \square \)
3.3.5 Bounds on the Main Algorithm

Finally we analyze the main algorithm, which repeatedly executes RECURTS until the graph is topologically sorted.

**Theorem 3.3.10.** Let \( G = (V, E) \) be any directed acyclic graph, and choose \( \lambda \geq 32 \lg V \) and \( K \geq 4 \lg V \). Then for any \( c > 0 \), with failure probability at most \( 1/|V|^c \), the graph is topologically sorted after at most \( \lceil (c + 2) \lg V \rceil \) iterations of the recursive algorithm.

**Proof.** Consider any initially violated edge \((u, v)\). By Lemma 3.3.9, each execution of RECURTS satisfies the edge with probability at least \( \frac{1}{2} \). Moreover, if the edge is satisfied in any iteration, it remains satisfied for all subsequent iterations. The probability that the edge is still violated after \( \lceil (c + 2) \lg V \rceil \) iterations is therefore at most \( (1/2)^{(c+2)\lg V} = 1/|V|^{c+2} \). Taking a union bound across less than \(|V|^2\) possible edges completes the proof. \( \square \)

**Theorem 3.3.11.** For any directed acyclic graph \( G = (V, E) \), there exist settings of \( K \) and \( \lambda \) such that the algorithm topologically sorts the graph in \( O(\text{sort}(E) \cdot \log^4 V) \) I/Os, with high probability.

**Proof.** From Lemma 3.2.4, a single execution of RECURTS has I/O cost \( O(K\lambda^2 \cdot \text{sort}(E)) \), with high probability. Theorem 3.3.10 states that \( O(\log V) \) executions suffice, with high probability, for \( \lambda = \Theta(\log V) \) and \( K = \Theta(\log V) \). Multiplying the \( O(\log V) \) executions by the cost per execution gives the theorem. \( \square \)

3.4 Strongly Connected Components

This section describes our algorithm for strongly connected components. Given a graph \( G = (V, E) \), vertices \( u, v \in V \) are *strongly connected* if there exist directed
paths both from $u$ to $v$ and from $v$ to $u$. A strongly connected component is a maximal set of vertices such that every pair of vertices therein is strongly connected. The condensation $H$ of a graph $G$ is the DAG of strongly connected components, i.e., the graph formed if each strongly connected component is contracted. The goal is to identify for each vertex the strongly connected component to which it belongs and to topologically sort the condensation.

At a high level, the main intent of the algorithm is similar to Algorithm 1 — reorder vertices to satisfy more edges. But it would, of course, be impossible to simultaneously satisfy all edges on a cycle. Our algorithm for strongly connected components therefore performs a little extra work to identify strongly connected vertices, notably those falling on short cycles, and contract them into a single supervertex. The graph is thus gradually transformed into its condensation; with each iteration, the number of violated edges may reduce both by removing contracted edges from the graph and by reordering any remaining supervertices.

Aside from the contraction, component maintenance, and extra bookkeeping, the main difference between the algorithms for topological sort and strongly connected components is that the former propagates priorities in only the forward direction, whereas the latter propagates priorities both forwards and backwards. This two-directional propagation facilitates the discovery of cycles.

3.4.1 Algorithm

Algorithm 2 presents a conceptual version of the algorithm for topologically sorting the condensation $H$ of the graph $G = (V, E)$. Section 3.4.3 provides implementation details for mapping this algorithm to the I/O model.

The algorithm maintains three types of information: (1) A mapping from vertices in the original graph to (partial) components, where each partial component is a
subset of vertices in a strongly connected component; (2) A graph \( H = (V_H, E_H) \) on the partial components, corresponding to the graph formed by contracting each partial component in \( G \); and (3) An ordering of the vertices \( V_H \) in the component graph. As the algorithm progresses, components are merged together through contraction steps. When the algorithm terminates, \( H \) is the condensation, and the vertex ordering represents a topological sort of the condensation.

As before, the top-level algorithm consists of multiple iterations. But now each iteration consists of not only an execution of \texttt{RecurSCC}, but also a contraction step following the execution. Each execution of \texttt{RecurSCC} is analogous to \texttt{RecurTS}, except that some vertices are flagged for contraction. Specifically, the output of \texttt{RecurSCC} is an updated ordering of the vertices \( V_H \) in the component graph \( H \) as before, but unlike \texttt{RecurTS} some contiguous sets of vertices flagged for contraction. Any initially satisfied edges between unflagged vertices remain satisfied, as before, and ideally some violated edges become satisfied. During the contraction step, sets of vertices identified as being strongly connected are contracted, removing any edges between them.

### 3.4.2 The Recursive Subroutine

The recursive subroutine \texttt{RecurSCC} is parameterized by global values \( \lambda \) and \( K \), denoting the maximum recursion depth and range of distances to choose from, respectively. \texttt{RecurSCC} takes as input an induced subgraph \( H[i..j] \) of the graph on partial components, and the current recursion depth \( \text{depth} \).

\texttt{RecurSCC} proceeds as follows. Much of the algorithm is similar to \texttt{RecurTS} of Section 3.2. Firstly, check if the recursion depth is exceeded (i.e., \( \text{depth} \geq \lambda \)), and if so simply return. Otherwise, choose a distance \( d \) uniformly at random from the
range \([d_{\min}, d_{\max}]\), where \(d_{\min} = d_{\max} - K\). As in Section 3.2, the offset for the range is chosen according to the recursion depth, with \(d_{\max} = (\lambda - \text{depth}) \cdot K\).

Next, assign a uniformly random permutation of priorities \(\rho(v)\) to each vertex. Unlike \textsc{RecurTS}, \textsc{RecurSCC} propagates the priorities in both the forward direction and the backward direction. Specifically, define \(f(v) = \max \{\rho(u) : u \trianglelefteq_d v\}\) and \(b(v) = \max \{\rho(w) : v \trianglelefteq_d w\}\). Assign a label \(l(v)\) to each vertex \(v\) based on the results of the forward and backward searches. There are three cases. If the priority from the forward search dominates, i.e., \(f(v) > b(v)\), then \(l(v) = f(v)\) as in \textsc{RecurTS}. If the priority from the backward search dominates, i.e., \(b(v) > f(v)\), then \(l(v) = -b(v)\). These two cases are symmetric — vertices dominated by larger priorities in the forward direction are pushed later in the ordering, and vertices dominated by larger priorities in the backward direction are pushed earlier in the ordering. The third case is if the priorities are equal in both direction. In this case, set \(l(v) = -b(v) + 1/2\).

Finally, sort the vertices by \(l(v)\), with ties broken according to the current ordering. After sorting, partition the vertices into groups of vertices having the same label, as in \textsc{RecurTS}. Recurse on those groups with integer labels, i.e., \(f(v) \neq b(v)\). The groups with non-integer labels are instead flagged for contraction.

Note that the specific choice of label \(-b(v) + 1/2\) for the third case is not particularly important. The only truly important aspect is that \(-b(v) < l(v) < f(v)\) to ensure that satisfied edges remain satisfied. In fact, the ordering across groups of vertices that fall in this third case, for different priorities, does not matter. It is, however, easier to implement the subsequent contraction if each such group of vertices be contiguous. To achieve that, we include the dominating priority in the label, e.g., choosing \(l(v) = -b(v) + 1/2\); many other choices would also suffice.

The main theorem, proved in Section 3.4.5, is the following:
Theorem 3.4.1. Let $G = (V, E)$ be any directed graph. There exist settings of constants $c_1$ and $c_2$ such that for $\lambda \geq c_1 \lg V$ and $K \geq c_2 \lg V$, the following holds. For any $c > 0$, with failure probability at most $1/V^c$, the algorithm terminates within $\lceil (c + 2) \lg V \rceil$ iterations of the main loop.

3.4.3 I/O-Efficient Details

This section provides details on making the algorithm I/O efficient. The original vertices of graph $G$ are stored in an array $V$. A second array $V_H$ stores the vertices of graph $H$. Each vertex in $H$ corresponds to a partial component in $G$ that has been contracted, identified by the ID of a representative vertex. The edges between components are stored in an array $E_H$, with each edge storing the component IDs of its endpoints.

All vertex records $u \in V$ for the original graph are tagged with the ID $c(u)$ of their component’s representative, which corresponds to the ID of a vertex in $V_H$. For convenience, the vertex records $v \in V_H$ are also tagged with $c(v)$. Initially, $c(u) = u$ for all $u \in V$, and $V_H = V$. In general between iterations, $c(v) = v$ if and only if $v \in V_H$. When the algorithm terminates, the vertices representing each strongly connected component are topologically ordered in $V_H$, and for each vertex $u \in V$, $c(u)$ specifies the representative of $u$’s strongly connected component.

The details for the recursive algorithm are similar to those for RECURTS in Section 3.2.2. There are minor differences in that the priorities must be propagated in two directions, now computing $b(v)$ in addition to the $f(v)$ already computed in RECURTS. But steps for $b(v)$ are symmetric, i.e., operating on the transpose graph, which can be computed in $O(sort(E))$ I/Os.

The only significant difference is implementing the main loop, namely in the contraction step.
Contraction. By design, when RECURSCC returns, groups of vertices that are to be contracted are contiguous in the vertex array \( V_H \). When a group is flagged for contraction, the boundaries of the group should also be marked.

The first step of the contraction is to update \( c(v) \) for all vertices \( v \in V_H \) to be contracted. Specifically, the first vertex \( x \) (in array order) in each group is the representative for the group. All other vertices in the group update \( c(u) = x \). This step can be accomplished by a scan of the array \( V_H \).

The next step is to update the components for vertices stored in \( V \). Specifically, each vertex \( u \in V \) has some component ID \( c(u) = v \), where \( v \in V_H \). The goal is to update \( c(u) = c(v) \). To do so, sort \( V \) by component IDs \( c(u) \) and sort \( V_H \) by vertex IDs \( v \). Thus, both \( V \) and \( V_H \) are sorted according to their original components. Moreover, the vertices \( v \in V_H \) already know their new component \( c(v) \). Next perform synchronized scans of \( V \) and \( V_H \), and for each vertex \( u \in V \), update \( c(u) = c(v) \).

Any vertices in \( V \) with \( c(v) \neq v \) can now be removed from \( V_H \). This step can be accomplished with a scan. At this point, all vertices have the correct component IDs, and only representatives are stored in \( V_H \).

The final step is to update the edges \( E_H \). Specifically, any edge \( (u, v) \) should be updated to reflect the component IDs of its endpoints, i.e., \( (c(u), c(v)) \). To do so, sort \( V \) by ID, and sort the edges \( (u, v) \in E_H \) by the ID of \( u \). Next perform synchronized scans of \( E_H \) and \( V \), updating \( u \) to \( c(u) \) for each edge \( (u, v) \). Then sort the edges by the ID of the other endpoint \( v \) and perform a similar update. Finally, self loops can be removed by scanning through all the edges one last time and checking for any edges of the form \( (u, u) \). Optionally, duplicate edges can also be removed by sorting the edges one last time (by both endpoints) and scanning through to remove duplicates.
3.4.4 I/O Complexity of Strongly Connected Components

Assuming Theorem 3.4.1, we now bound the I/O cost of the algorithm.

**Lemma 3.4.2.** A single execution of \textsc{RecurSCC} has I/O cost $O(K\lambda^2\text{sort}(E))$ I/Os, with high probability.

*Proof.* Proof is identical to proof of Lemma 3.2.4, except that the specific constants change because priorities must be propagated in two directions. \hfill \Box

**Theorem 3.4.3.** For any directed graph $G = (V, E)$, there exist settings of $K$ and $\lambda$ such that the algorithm topologically sorts the condensation of the graph in $O(\text{sort}(E) \cdot \log^4 V)$ I/Os, with high probability.

*Proof.* Theorem 3.4.1 states that for $\lambda = \Theta(\log V)$ and $K = \Theta(\log V)$, after $O(\log V)$ executions of \textsc{RecurSCC}, the algorithm is successful with high probability. From Lemma 3.4.2, the cost of each execution of \textsc{RecurSCC} is $O(K\lambda^2\text{sort}(E))$. The cost of each contraction step is $O(\text{sort}(E))$, which is dominated by the cost of executing the recursive algorithm. Therefore the total I/O cost of the full algorithm is $O(\text{sort}(E) \cdot \log^4 V)$.

\hfill \Box

3.4.5 Strongly Connected Components Analysis

The goal of this section is to prove Theorem 3.4.1, i.e., that $O(\log V)$ executions of the main loop suffice, with high probability. The analysis follows a similar structure to the analysis of topological sort in Section 3.3. The main goal is to show that any violated edge $(u, v) \in E_H$ has a constant probability of either becoming satisfied in an execution of \textsc{RecurSCC} or being contracted away thereafter. Since any cycle in the graph must have at least one violated edge, satisfying all remaining edges implies that all cycles have been contracted, and the condensation of the graph is topologically sorted. Given that claim, it is easy to show that $O(\log V)$ iterations suffice.
As before, the analysis consists of two main components applied to graph $H$. A minor difference is that groups of vertices to be contracted are technically not part of a recursive subproblem. Insofar as definitions are concerned (e.g., being broken), when we say “subproblem” we mean each group of vertices produced by the partitioning step in the algorithm, either corresponding to a group to be contracted or a recursive call.

The first component of the analysis is to show that an edge is unlikely to break. This component is largely similar to the corresponding component in Section 3.3, except that edges may break due to searches in either direction. Note that, conveniently, edges within a group to be contracted are never broken as these edges do not cross subproblem boundaries.

The goal of the second component is now to show that for any edge $(u, v) \in E_H$, the execution of RECURSCC is likely to end either with $u$ and $v$ in different subproblems, or with $u$ and $v$ marked for contraction with each other. If the edge is not broken, and $u$ and $v$ are in different subproblems, then the edge becomes satisfied. If $u$ and $v$ are contracted, then the edge is removed from the graph entirely because the contraction step removes self loops.

Successors. The main differences in the analysis arise from the fact that priorities are propagated in two directions. It is thus no longer sufficient to focus just on the $d$-hop predecessors. We must also consider the successors. We define the $d$-hop successors of $v$, denoted by $D(G, v, d)$, as $D(G, v, d) = \{x : v \preceq_d x \text{ in } G\}$.

The presence of backward propagation impacts that analysis in various places, most of which are minor. The most difficult is in arguing progress with respect to the number of nearby vertices. Rather than argue progress on just the number of nearby
(active) predecessors, Lemma 3.4.8 argues progress on both nearby predecessors and successors.

3.4.6 Bounding the Probability of an Edge Becoming Broken

The following lemmas are analogous to Lemmas 3.3.2 and 3.3.3. Notably, an edge cannot be broken unless either its $d$-hop predecessor or successor is exactly $d$ hops away, which is unlikely to occur.

**Lemma 3.4.4.** Consider call RecurSCC($H, i, j, \text{depth}$). Let $H' = H[i \ldots j]$ denote the induced subgraph, let $(u, v)$ be an edge in $G'$, and let $d$ be the random distance chosen. Let $x$ denote the vertex with highest priority in the set $A(H', u, d)$. If $x \in A(H', u, d - 1)$, then $f(u) \leq f(v)$. Similarly, let $y$ denote the vertex with highest priority in the set $D(H', v, d)$. If $y \in D(H', v, d - 1)$, then $b(v) \leq b(u)$.

**Proof.** Suppose $x \in A(H', u, d - 1)$. Then we have $x \leq_d u$ and $u \leq_1 v$, giving $x \leq_d v$. It follows that $f(v) \geq \rho(x)$. Moreover, since $x$ has the highest priority among $u$'s $d$-hop predecessors, $f(u) = \rho(x)$. We thus have $f(v) \geq \rho(x) = f(u)$.

Suppose $y \in D(H', v, d - 1)$. Then we have $u \leq_1 v$ and $v \leq_{d-1} y$, giving $u \leq_d y$. It follows that $b(u) \geq \rho(y)$. Moreover, since $y$ has the highest priority among $v$'s $d$-hop predecessors, $b(v) = \rho(y)$. We thus have $b(u) \geq \rho(y) = b(v)$. \hfill \Box

**Lemma 3.4.5.** Consider call RecurSCC($H, i, j, \text{depth}$). Let $H' = H[i \ldots j]$ denote the induced subgraph and let $(u, v)$ be a violated edge in $H'$. Let $s_d = |A(H', u, d)|$ and $t_d = |D(H', u, d)|$ denote the number of $d$-hop predecessors and $d$-hop successors, respectively, of $u$ in $H'$, for each possible $d \in [(\lambda - \text{depth} - 1)K - 1, (\lambda - \text{depth})K)$. Then the probability that the edge becomes broken during this call is at most $(1/K) \cdot \lg(s_{(\lambda - \text{depth})K-1}/s_{(\lambda - \text{depth} - 1)K-1}) + (1/K) \cdot \lg(t_{(\lambda - \text{depth})K-1}/t_{(\lambda - \text{depth} - 1)K-1})$. 48
Proof. The violated edge \((u, v)\) becomes broken if and only if \(l(u) > l(v)\), in which case \(v\) is ordered before \(u\), and \(u\) and \(v\) are placed into different subproblems. We have \(l(u) > l(v)\) only if \(f(u) > f(v)\) or \(b(v) > b(u)\). The probability that \(f(u) > f(v)\) is the same as the proof of Lemma 3.3.3, but this time using Lemma 3.4.4. The case that \(b(v) > b(u)\) is symmetric, which gives the total probability of \((u, v)\) becoming broken during this call to be \((1/K) \cdot \lg(s(\lambda - \text{depth})K_1/s(\lambda - \text{depth} - 1)K_1) + (1/K) \cdot \lg(t(\lambda - \text{depth})K_1/t(\lambda - \text{depth} - 1)K_1)\). 

3.4.7 Bounding the Probability that an Edge Crosses Subproblems

The analysis here is analogous to Section 3.3.2. The main difference is that here we consider the number of active vertices in both directions, not just predecessors. For this section, we adopt the same notion of level-\(\ell\) graphs as in Section 3.3.2, except applied to the contracted graph \(H\) instead of the original graph \(G\). Note that \(H_v^{\ell}\) is the empty graph if \(v\) is no longer part of a recursive subproblem, which can now also occur if \(v\) is marked for contraction before the \(\ell\)-th level of recursion.

This first lemma says that vertices do not get closer together, i.e., no new relationships are created, when recursing. Consequently, it is sufficient to argue that a constant fraction of related vertices are likely to be knocked out.

Lemma 3.4.6. Consider any vertex \(v\) and its level-(\(\ell - 1\)) and level-(\(\ell\)) subgraphs \(H_v^{\ell-1}\) and \(H_v^{\ell}\), respectively. For any vertex \(x\) and distance \(d\), if \(x \not\preceq_d v \) in \(H_v^{\ell-1}\), then \(x \not\preceq_d v \) in \(H_v^{\ell}\). Similarly, if \(v \not\preceq_d x \) in \(H_v^{\ell-1}\), then \(v \not\preceq_d x \) in \(H_v^{\ell}\).

Proof. The distances in the graph can only decrease if edges become satisfied, new edges are incorporated, or if vertices are contracted. None of these occurs within the scope of recursive subproblems — the relative ordering within each subproblem is unchanged, and contraction only occurs between iterations. 

49
The remainder of the section focuses on the number of active vertices, except we now consider both predecessors and successors.

**Definition 3.4.7.** A vertex $x$ is a level-$\ell$ active successor of $v$ if $x \neq v$ and $v \preceq d_{\text{max}} x$ in $H_v^{\ell}$, where $d_{\text{max}} = K(\lambda - \ell)$ is the maximum distance for this level of recursion. Notice this definition is symmetric to the level-$\ell$ active predecessors of $v$ and similarly, $v$ is not an active successor of itself.

Consider any edge $(u, v)$ that is violated at the start of the recursive algorithm. Observe that if $v$ has no level-$\ell$ active predecessors, then either $(u, v)$ falls within a group marked for contraction, or $(u, v)$ crosses a subproblem boundary.

The analysis differs here from topological sort because the subproblem derives from both predecessors and successors. In particular, the label $l(v)$, for a vertex $v$, is based on whether the forwards or backwards search dominates. Our claim here is that the sum of the number of active predecessors and active successors decreases by a constant factor in each level of recursion with constant probability.

**Lemma 3.4.8.** Consider any vertex $v$ and level $\ell$ of recursion. Let $A$ and $A'$ be the set of level-$\ell$ and level-$(\ell + 1)$, respectively, active predecessors of $v$. Let $D$ and $D'$ be the set of level-$\ell$ and level-$(\ell + 1)$, respectively, active successors of $v$. Let $\eta = |A \cup D|$, and $\eta' = |A' \cup D'|$. Then $\Pr[\eta' \leq (7/8)\eta] \geq 1/6$.

**Proof.** Consider the level-$\ell$ call to RECURSCC on graph $H_v^\ell$. Let $d$ be the distance selected. Let $A_d = A(H_v^\ell, v, d) \setminus \{v\}$ be the set of $d$-hop predecessors of $v$ in $H_v^\ell$, excluding $v$ itself, and let $a_d = |A_d|$. Similarly, let $D_d = D(H_v^\ell, v, d) \setminus \{v\}$ be $v$’s $d$-hop successors. Let $r_d = |A_d \cup D_d|$ be the total number of $d$-hop related vertices, in both directions, excluding $v$. For the remainder of the proof, assume without loss of generality (by symmetry) that $|A_d| \geq |D_d|$, and hence $a_d \geq r_d/2$. 

50
Since \( d \leq d_{\text{max}} \), we have \( A_d \subseteq A \) and \( D_d \subseteq D \) and hence \( r_d \leq \eta \). By Lemma 3.4.6 and the fact that distances decrease with each level of recursion, \( A' \subseteq A_d \) and \( D' \subseteq D_d \). It suffices to show that, with probability at least \( 1/6 \), at least \( a_d/4 \) of the vertices in \( A_d \) are not in \( H_v^{\ell+1} \). Assuming this outcome occurs, we have \( \eta' = |A' \cup D'| \leq |A_d \cup D_d| - a_d/4 = r_d - a_d/4 \leq (7/8)r_d \leq (7/8)\eta \) as desired.

Let \( x \) be a random variable denoting the vertex in \( A_d \cup D_d \cup \{v\} \) with highest priority. Consider the random process as follows: first toss a weighted coin to determine if \( x \) is in \( A_d \cup \{v\} \) or \( D_d \setminus A_d \), then select a vertex uniformly at random from the appropriate set. Since \( |A_d| \geq |D_d| \), the former occurs with probability at least \( 1/2 \).

The remainder of the proof thus conditions on the assumption that \( x \in A_d \cup \{v\} \), with the final success probability multiplied by \( 1/2 \).

The remainder of the proof is similar in setup to Lemma 3.3.5, but the knocks-out relation and specific cases are more complicated. We say that \( x \) knocks out \( y \) if \( x \not\leq_d y \) or if both \( x \leq_d y \) and \( y \leq_d x \). Once more, it suffices to prove the following two claims:

1. If \( x \) knocks out \( y \), then \( y \not\in H_v^{\ell+1} \), and
2. With probability at least \( 1/3 \), \( x \) knocks out at least \( a_d/4 \) vertices from \( A_d \).

Claim 1. We start by noting that by assumptions on choice of \( x \), \( f(v) = \rho(x) \). Moreover, since \( \rho(x) \) is the highest priority \( v \) observes in either direction, \( b(v) \leq \rho(x) \).

A necessary condition for \( y \in H_v^{\ell+1} \) is thus that \( f(y) = \rho(x) \) and \( b(y) \leq \rho(x) \). If \( b(y) = f(y) = \rho(x) \), however, then \( y \) is marked for contraction and not part of \( H_v^{\ell+1} \). So \( y \in H_v^{\ell+1} \) also requires \( b(y) < \rho(x) \).

By definition, if \( x \) knocks out \( y \) then either \( x \not\leq_d y \), or both \( x \leq_d y \) and \( y \leq_d x \). If \( x \not\leq_d y \), then \( f(y) \neq \rho(x) \), implying that \( y \not\in H_v^{\ell+1} \) as discussed above. Suppose instead that \( x \leq_d y \) and \( y \leq_d x \). Then \( f(y) \geq \rho(x) \) and \( b(y) \geq \rho(x) \). Again, this implies that \( y \not\in H_v^{\ell+1} \) because \( b(y) \neq \rho(x) \).
Claim 2. If \( x \not\preceq_d y \), then \( x \) knocks out \( y \); likewise if \( y \not\preceq_d x \), then \( y \) knocks out \( x \). If \( x \preceq_d y \) and \( y \preceq_d x \), then \( x \) and \( y \) knock out each other. Therefore, either \( x \) knocks out \( y \), \( y \) knocks out \( x \), or both. Moreover, \( x = v \) knocks out every vertex as in this case \( f(v) = b(v) = \rho(v) \), and hence \( H_v^{l+1} \) is the empty graph. The total number of pairs \( x \in A_d \cup v \) and \( (y \neq x) \in A_d \) for which \( x \) knocks out \( y \) is at least \( \binom{a_d}{2} + a_d(a_d + 1)/2 \). The rest of the claim is that same as Lemma 3.3.5, which completes the proof.

The preceding lemma indicates that each level of recursion is likely to reduce the total number of active vertices by a constant factor. The following lemma applies this lemma across \( \lambda \) levels of recursion to conclude that \( v \) is likely to be in its own subproblem, or marked for contraction, before the recursion bottoms out.

**Lemma 3.4.9.** Consider any vertex \( v \) and a complete execution of the recursive algorithm. With probability at least \( 1 - 32 \lg |V| / \lambda \), \( v \) has no active predecessors at the \( \lambda \)-th level of recursion.

**Proof.** Let \( X_j \) be a random variable denoting the number of levels \( \ell \) for which the number of active predecessors and active successors of \( v \) falls in the range \( [(7/8)^j, (7/8)^j) \cdot |V| \), for integer \( j \). When \( j \geq \log_{8/7} |V| \), the high end of the range is strictly less than \( 1/|V| \cdot |V| = 1 \), meaning that \( v \) has no active predecessors or active successors remaining. Let \( X = \sum_{j < \log_{8/7}|V|} X_j \). If \( X < \lambda \), then at or before the the \( \lambda \)-th level of recursion, \( v \) has no active predecessors. Our goal is thus to argue that this event is likely to occur. Lemma 3.4.8 says that the number of rounds necessary to get a \((7/8)\) reduction is at most 6 in expectation. Thus \( E[X_j] \leq E[X_j | X_j \geq 1] \leq 6 \). By linearity of expectation, \( E[X] \leq 6 \log_{8/7}(|V|) < 32 \lg(|V|) \). By Markov’s inequality, \( \Pr \{ X \geq \lambda \} \leq 32 \lg(|V|)/(\lambda) \). 

\( \square \)
3.4.8 Edges are Likely to Become Satisfied

We have argued that in each execution of the recursive algorithm, a particular edge $(u, v)$ is unlikely to become broken, and moreover that $v$ is likely to either be in its own subproblem or marked for contraction. The implication is that if both favorable outcomes occur, the edge $(u, v)$ either cross subproblem boundaries and becomes satisfied, or both $u$ and $v$ are contracted with each other and the edge disappears. Completing this argument again requires monotonic progress on satisfied edges. The following lemma says that satisfied edges never become violated later.

**Lemma 3.4.10.** Consider an execution of the recursive algorithm RecurSCC. If an edge $(u, v)$ is satisfied at the $\ell$-th level of recursion, then it remains satisfied at all subsequent levels of recursion.

**Proof.** Proof by induction on the level of recursion. Consider the call at the $\ell$th level of recursion, and suppose that the edge $(u, v)$ is satisfied at the start of the call. The goal is to show that it remains satisfied in the next recursive subproblem. Note that once $u$ and $v$ fall in different subproblems (or the same subproblem marked for contraction), their relative order never changes.

We first claim that if $l(v) \geq l(u)$ and $u$ and $v$ are in the same subproblem, then $(u, v)$ remains satisfied at the next level of recursion. To see that, if $l(v) > l(u)$, then sorting by label keeps the edge satisfied. If $l(v) = l(u)$ but $(u, v)$ is satisfied initially, then $\text{index}(v) > \text{index}(u)$. Thus, breaking ties by index keeps the edge satisfied. The remainder of the proof thus focuses on showing that $l(v) \geq l(u)$.

Since $(u, v)$ is satisfied, $u \preceq_0 v$. More importantly, for all $x$, $x \preceq_d u$ implies $x \preceq_d v$. Similarly, $v \preceq_d x$ implies $u \preceq_d x$. It follows that

$$f(u) \leq f(v) \text{ and } b(u) \geq b(v) . \quad (3.1)$$
To show \( l(v) \geq l(u) \), we consider several cases.

Case 1: \( f(u) > b(u) \). Then \( l(u) = f(u) \). We have \( f(v) \geq f(u) > b(u) \geq b(v) \), where the first and last inequality follow Equation 3.1 and the middle one is by assumption. Both vertices are assigned their forwards label, and \( f(v) \geq f(u) \), implying \( l(v) \geq l(u) \).

Case 2: \( f(u) = b(u) \). Then \( u \) is assigned label \( l(u) = -b(u) + 1/2 \). Similarly to the first case, we have \( f(v) \geq f(u) = b(u) \geq b(v) \), implying \( f(v) \geq b(v) \). If \( f(v) > b(v) \), then \( l(v) = f(v) \), which is nonnegative and hence larger than \( l(u) = -b(u) + 1/2 \). If \( f(v) = b(v) \), then \( l(v) = -b(v) + 1/2 \). Since \( b(u) \geq b(v) \), \( l(v) = -b(v) + 1/2 \geq -b(u) + 1/2 = l(u) \).

Case 3: \( f(u) < b(u) \). Then \( l(u) = -b(u) \). If \( f(v) > b(v) \), then \( l(v) = f(v) > -b(u) = l(u) \), so \( l(v) > l(u) \). If \( f(v) = b(v) \), then \( l(v) = -b(v) + 1/2 \). Since \( b(u) \geq b(v) \) from Equation 3.1, \( -b(v) \geq -b(u) \), implying \( l(v) = -b(v) + 1/2 \geq -b(u) = l(u) \). If \( f(v) < b(v) \), then \( l(v) = -b(v) \). Since \( b(u) \geq b(v) \), \( l(v) \geq l(u) \).

\[ \square \]

**Lemma 3.4.11.** Let \((u, v)\) be an edge that is initially violated, and consider a complete execution of \textsc{RecurSCC}(\(G, 1, |V|, \text{depth}\)). Then the probability that \((u, v)\) becomes broken in this execution is at most \(2 \log(|V|)/K\).

**Proof.** This proof is the same as Lemma 3.3.8 except the probability is twice as high since we include the successors.

Let \( s_d^{(i)} \) and \( t_d^{(i)} \) be the number of \( d\)-hop predecessors and successors, respectively, of \( u \) in the \( i \)th level of recursion. By Lemma 3.4.5 the probability that the edge gets broken in the \( i \)th level of recursion is \((1/K) \cdot (\log(s_{\lambda-i}^{(i)}K_{-1}/s_{\lambda-i}^{(i)}K_{-1}) + \log(t_{\lambda-i}^{(i)}K_{-1}/t_{\lambda-i}^{(i)}K_{-1}))\). Let \( B \) be the event that \((u, v)\) gets broken in any level of recursion. Summing across all the levels we have,

\[
\Pr [B] = (1/K) \sum_{i=0}^{\lambda-1} [\log(s_{\lambda-i}^{(i)}K_{-1}) - \log(s_{\lambda-i-1}^{(i)}K_{-1}) + \log(t_{\lambda-i}^{(i)}K_{-1}) - \log(t_{\lambda-i-1}^{(i)}K_{-1})].
\]

54
This time using Lemma 3.4.6, the number of related nodes at a distance $d$ cannot increase across levels of recursion $s_d^{(i+1)} \leq s_d^{(i)}$, and $t_d^{(i+1)} \leq t_d^{(i)}$, and therefore,

$$\Pr [B] \leq (1/K)(\lg s_{K-1}^{(0)} - \lg s_0^{(\lambda-1)} + \lg t_{K-1}^{(0)} - \lg t_0^{(\lambda-1)}) \leq (1/K)(\lg s_{K-1}^{(0)} + \lg t_{K-1}^{(0)}).$$

Since $s_{K-1}^{(0)} \leq |V|$, and $t_{K-1}^{(0)} \leq |V|$, $\Pr [B] \leq (1/K)(\lg |V| + \lg |V|) = (2/K) \lg |V|$.

\[\square\]

**Lemma 3.4.12.** Let $(u, v)$ be any edge, and consider a complete execution of RecurSCC followed by the contraction step with parameters $\lambda \geq 128 \lg V$ and $K \geq 8 \lg V$. If $(u, v)$ is violated initially, then with probability at least $1/2$, $(u, v)$ is not violated at the end (either because it is satisfied or removed from the graph). If $(u, v)$ is not violated initially, then with probability 1 it is also not violated at the end.

**Proof.** By Lemma 3.4.10, a satisfied edge always remains satisfied. It may, however be removed from the graph in the contraction step. Nevertheless, it can never become violated. The remainder instead focuses on the case that an edge $(u, v)$ is initially violated.

Let $A$ be the event that $(u, v)$ is violated at the end of the execution. Let $B$ be the event that the edge breaks at some level of recursion, and let $C$ be the event that the two endpoints $u$ and $v$ are in the same level-$\lambda$ subproblem. If neither $B$ nor $C$ occurs, then either the edge crosses properly ordered subproblems at some level, or $u$ and $v$ are marked for contraction. In either case, $(u, v)$ is not violated anymore. We thus have $\Pr [A] \leq \Pr [B] + \Pr [C]$ by a union bound.

By Lemma 3.4.9, $\Pr [C] \leq 32 \lg V/\lambda \leq 1/4$ for the specified choice of $\lambda$. By Lemma 3.4.11, the probability of breaking at any level is at most $2 \lg V/K$, and so for the specified choice of $k$, $\Pr [B] \leq 1/4$. Adding these together gives total failure probability of at most $1/2$. \[\square\]
3.4.9 Bounds on the Main Algorithm

Lemma 3.4.13. Only vertices that are strongly connected are contracted.

Proof. Given vertices $u$ and $v$ that are contracted, we will show that $u$ and $v$ are strongly connected. Since $u$ and $v$ are contracted, it must be the case that $f(u) = b(u) = f(v) = b(v)$. There must be some vertex $x$ such that $\rho(x) = f(u)$. It could be the case that either $u$ or $v$ is vertex $x$. Since $\rho(x) = f(u) = b(u) = f(v) = b(v)$, $x \leq_d u$, $x \leq_d v$, $u \leq_d x$ and $v \leq_d x$. Therefore $x$ is strongly connected to both $u$ and $v$, which implies that $u$ and $v$ are strongly connected.

We next prove Theorem 3.4.1, which states that the algorithm topologically sorts the condensation of the graph after $\lceil (c + 2) \lg V \rceil$ executions with failure probability at most $1/V^c$, for any $c > 0$.

Proof of Theorem 3.4.1. By Lemma 3.4.13, the algorithm never performs any erroneous contractions. If the algorithm terminates, it must therefore be the case that the graph is topologically sorted, which is only possible there are no cycles, i.e., if all strongly connected components have been contracted.

Lemma 3.4.10 says that violated edges are never introduced. Moreover, by Lemma 3.4.12, each violated edge has a constant probability of being removed or becoming satisfied. The rest of the proof is the same as Theorem 3.3.10.

3.5 Conclusions

This work has shown the first algorithm for topological sort and related problems that is I/O efficient even for sparse graphs.

The main question remaining is whether the algorithm can be improved to achieve an I/O cost $O(sort(E) \cdot \log^x V)$, for $x < 4$. One of the logarithmic factors arises from
the fact that distance ranges do not overlap at each level of recursion. We suspect that this logarithmic factor can be removed, yielding $x = 3$. Achieving $x < 3$, however, seems difficult. Inherent in the approach are at least two logarithmic factors: the number of iterations and the number of levels of recursion. Moreover, reducing the number of distances to $K = O(1)$, which would be necessary to get $x < 3$, would require some significant new ideas.

Another interesting question is whether randomization is necessary for these problems. Randomization plays a key role in our algorithm, but there may be alternative approaches.
Algorithm 2: Conceptual algorithm for strongly connected components

1: function SCC($G = (V, E)$)
2:  $H = (V_H, E_H)$, where $V_H = V$ and $E_H = E$ initially
3: repeat until $H$ is topologically sorted
4:  RECURSCC($H, 1, |V_H|, 0$)
5:  Perform contraction step and update $H$
6: function RECURSCC($H, i, j, depth$) ▷ Reorders the subarray $V_H[i..j]$ of vertices
7:  $H' = H[i..j]$
8:  if $depth \geq \lambda$ or $i = j$ then return
9:  $d_{max} = (\lambda - depth) \cdot K$ and $d_{min} = d_{max} - K$
10: Choose $d$ uniformly at random from $[d_{min}, d_{max}]$
11: Choose a uniformly random permutation of priorities $\{\rho(v)\}$
12: For all $v$, compute $f(v) = \max \{\rho(u) : u \preceq_d v \text{ in } H'\}$
13: For all $v$, compute $b(v) = \max \{\rho(w) : v \preceq_d w \text{ in } H'\}$
14: for each $v \in V_H[i..j]$
15:  $l(v) = \begin{cases} -b(v) & \text{if } b(v) > f(v) \text{ (backward search dominates)} \\ -b(v) + 1/2 & \text{if } b(v) = f(v) \text{ (strongly connected)} \\ f(v) & \text{if } b(v) < f(v) \text{ (forward search dominates)} \end{cases}$
16: Sort vertices $V_H[i..j]$ lexicographically by $(l(v), \text{index}(v))$
17: Partition $V_H[i..j]$ into maximal groups $[i_1, j_1], [i_2, j_2], \ldots, [i_t, j_t]$ of a single label ($i_r = j_{r-1} + 1$)
18: for $r = 1$ to $t$
19:  if $f(V_H[i_r]) = b(V_H[i_r])$ then ▷ do not recurse on strongly connected groups
20:    mark the vertices in the group for contraction
21:  else RECURSCC($H, i_r, j_r, depth + 1$)
The work in this chapter is published in *Efficient Construction of Directed Hopsets and Parallel Approximate Shortest Paths* [22] in *Symposium on Theory of Computing 2020*.

4.1 Introduction

The single source shortest path problem is given a graph $G$ and source node $s$ to compute the shortest path distance from $s$ to each node in the graph. In the sequential setting this problem can be solved by Dijkstra’s algorithm [31] in $(m + n \log n)$ time for a $n$-node, $m$-edge graph with non-negative integer edge weights.

In this chapter we turn to approximate single source shortest paths for directed graphs. For a graph $G$ and source node $s$, the $(1 + \epsilon)$ approximate single-source shortest paths problem is to output a distance estimate $d(v)$, for each node such that, $\text{dist}(s, v) \leq d(v) \leq (1 + \epsilon) \text{dist}(s, v)$. Recall that the work of an algorithm is the total number of operations, and the span is the longest chain of sequentially dependant operations. Our result is the first algorithm to simultaneously achieve nearly linear work and sublinear span for directed $(1+\epsilon)$-approximate shortest paths. Our algorithm runs in $\tilde{O}(m \log(nW)/\epsilon^4)$ work and $n^{1/2+O(1/\log\log n)}/\epsilon$ span, where $W$ is the ratio of the maximum edge weight to the minimum strictly positive edge weight.

One technique for solving parallel shortest paths is to construct hopsets. For a graph $G = (V, E)$, a $(\beta, \epsilon)$-hopset is a set of weighted edges $E'$ such that, for each
pair of nodes $u, v$, there exists a path $p$ from $u$ to $v$ where $p$ contains at most $\beta$ edges and $\text{dist}_G(u, v) \leq \text{dist}^{\beta}_{G'}(u, v) \leq (1 + \epsilon)\text{dist}_G(u, v)$, where $G' = (V, E \cup E')$.

The size of the hopset is $|E'|$, and $\beta$ is called the hopbound. After adding the hopset edges to the graph, the shortest paths are at most $\beta$ hops. We can then use Klein and Subramanian’s distance limited shortest paths algorithm to solve approximate shortest paths in $\tilde{O}(E + E')$ work and $\beta$ span. The problem that remains is efficiently constructing a hopset with the desired hopbound and approximation.

There has been lots of work on hopsets for undirected graphs [27, 34, 35, 54]. Elkin and Neiman show how to construct a linear size hopset with subpolynomial hopbound [35].

For directed hopsets the is a simple algorithm for constructing an exact hopset which achieves $\tilde{O}(\sqrt{n})$ hopbound, has size $O(n)$, however the work is high, $\tilde{O}(m\sqrt{n})$. The algorithm works by first sampling each node with probability $O(1/\sqrt{n})$. Then each sampled node computes the shortest path distance to each other sampled node. For a sampled node $s$, an edge $(s, s')$ is added to the hopset all the other sampled nodes $s'$ with weight equal to the distance from $s$ to $s'$. Since each sampled node adds one edge for each other sampled node, the size of the hopset is $O(n)$ with high probability. To see the hopbound, consider a shortest path from $u$ to $v$. With high probability there is a sampled node in the first $\tilde{O}(\sqrt{n})$ nodes on the path, and similarly for the last $\tilde{O}(\sqrt{n})$ nodes. The path between the sampled nodes is one hop with the hopset edge. The work for the algorithm comes from computing the shortest paths from each sampled node, which is $\tilde{O}(m\sqrt{n})$

Ullman and Yannakakis parallelize this hopset algorithm for unweighted directed graphs. Their algorithm has $O(mn^{1/2})$ work with $O(n^{1/2})$ hopbound [59]. Klein and Subramanian extend this algorithm to integer weights and achieve the same bounds [44].
Our result is the first efficient construction of directed hopsets with sublinear hop-bound. Our sequential algorithm for unweighted graphs runs in $\tilde{O}(m/\epsilon^2)$ time and produces a $\tilde{O}(n/\epsilon^2)$-size $(\beta = n^{1/2+O(1/\log \log n)}, \epsilon)$-hopset, with high probability. We extend this result to weighted graphs and achieve the same hop-bound, but with size $\tilde{O}(n \log(nW) / \epsilon)$ and $\tilde{O}(m \log(nW) / \epsilon^2)$ runtime where $W$ is the ratio of the maximum edge weight to the minimum strictly positive edge weight. Finally our parallel algorithm for weighted graphs also gets the same hopbound and approximation and runs in $\tilde{O}(m \log(nW) / \epsilon^4)$ work and $n^{1/2+O(1/\log \log n)} / \epsilon$ span. The construction of the hopset is with high probability in that there is a small probability that the hopset produced fails to achieve the $(1 + \epsilon)$ approximation. We then apply Klein and Subramanian’s distance limited shortest paths algorithm [44] to solve approximate single source shortest paths in the same work and span as constructing the hopset.

In follow up work, we extended this result so that the algorithm can exhibit a trade off between the work and span. Specifically, our follow up result is an algorithm for single source reachability and $(1 + \epsilon)$-approximate shortest paths that runs in $\tilde{O}(m\rho^2 + n\rho^4)$ work and $n^{1/2+o(1)}/\rho$ span, for all $\rho \in [1, \sqrt{n}]$ [23].

4.1.1 Relation to Parallel Reachability and Diameter Reduction

The main idea for our algorithm is based on two parallel algorithms for directed diameter reduction by Fineman [37] and Jumalapati et. al. [41], which we call JLS throughout this work. The reachability problem is given a directed graph $G$ and source node $s$, to compute the set of nodes that are reachable from $s$, i.e. the set of nodes $v$ where there exists a path from $s$ to $v$ in $G$.

The two aforementioned algorithms use diameter reduction and then parallel BFS to solve parallel reachability. The diameter of a graph $G$ is the longest shortest path in $G$ among any $u, v$ pair of nodes. If a graph has diameter $k$, then the problem
can be solved in $O(m)$ work and $O(k)$ span by running parallel BFS. The diameter reduction problem is given a directed graph $G$, to add a set of shortcuts to the graph to reduce the diameter to $o(n)$. Parallel BFS runs the same as BFS except explores all the edges in one layer in parallel. The work is the same as sequential BFS, $O(m + n)$, however the span is proportional to the diameter. The cost of parallel reachability is the work and span for reducing the diameter to $O(k)$, plus $O(m)$ work and $O(k)$ span for parallel BFS.

Fineman’s algorithm adds a size $\tilde{O}(n)$ set of shortcuts to the graph which reduces the diameter to $\tilde{O}(n^{2/3})$ [37]. The algorithm has $\tilde{O}(m)$ work and adds a set of shortcuts of size $\tilde{O}(n)$. Note that since the shortcuts are added to the graph, it is important that the size of the set of shortcuts is not too big. The runtime of parallel BFS is proportional on the number of edges in the graph after the shortcuts are added. By adding $\tilde{O}(n)$ shortcuts, the work of parallel BFS does not increase too much. JLS extends Fineman’s algorithm to perform diameter reduction to $n^{1/2+o(1)}$ diameter with the same work, and number of shortcuts [41].

Lower bounds and further diameter reduction. There is a lower bound for shortcuts that for a linear size number of shortcuts there exists a graph where the diameter cannot be reduced below $\Omega(n^{1/6})$ [39]. This lower bound applies to hopsets meaning that for a linear size hopset the hopbound cannot be better than $\Omega(n^{1/6})$. Subsequent work has shown that it is possible to reduce the diameter of a graph to $O(n^{1/3})$ with a $\tilde{O}(n)$ size set of shortcuts however the best algorithm to construct such a set of shortcuts has work as high as matrix multiplication [45].
4.1.2 Roadmap for this Chapter

In the next section we will give some preliminaries and notation used throughout the chapter. Next in Section 4.3 we will show the sequential algorithm for unweighted hopsets. This algorithm gets at many of the main ideas and difficulties for constructing a hopset. In the following section we show how to make the algorithm work weighted graphs, and then finally in Section 4.6 we show how to make the algorithm parallel.

4.2 Preliminaries

For a weighted directed graph $G$, we denote $W$ as the maximum edge of an edge. We assume that the minimum non-zero edge weight is one. If a graph does not have this property, then we can perform the following scaling. Let $y < 1$ be the minimum non-zero edge weight. We scale all the edge weights by $1/y$, i.e. for each edge $e \in E$, the new scaled edge weight $w'(e)$ is $w(e)/y$.

Recall that a path $P = \langle v_0, v_1, ... v_\ell \rangle$ is a sequence of vertices that are joined by edges. The length or size of a path $P = \langle v_0, v_1, ... v_\ell \rangle$ is the number of edges on the path and is denoted $|P| = \ell$. The weight of a path is the sum of all the edge lengths on the path, $w(P = \sum_{i=1}^{\ell} w(v_{i-1}, v_i)$. The first and last nodes on the path are called the head, $\text{head}(P) = v_0$ and the tail, $\text{tail}(P) = v_\ell$, respectively.

Recall that $\text{dist}_G(s, t)$ is length of the shortest path from $s$ to $t$ in $G$. Define $\text{dist}^\beta_G(s, t)$ to be the shortest path from $s$ to $t$ containing at most $\beta$ edges. If there is no path from $s$ to $t$ that contains $\beta$ or less edges, then $\text{dist}^\beta_G(s, t) = \infty$.

For a node $v$ define the $R^{-}_d(G, v)$ to be the set of nodes $u$ where $\text{dist}(u, v) \leq d$. Symmetrically, define $R^{+}_d(G, v)$ to be the set of nodes $u$ where $\text{dist}(v, u) \leq d$. For a node $v$, $R(G, v) = R^{-}_d(G, v) \cup R^{+}_d(G, v)$. Next we extend the related nodes to
paths. For a path \( P = \langle v_1, v_1, ... v_\ell \rangle \), \( R_d^-(G, P) = \{ u \mid u \in R_d^-(G, v_i), v_i \in P \} \), and \( R_d^+(G, P) = \{ u \mid u \in R_d^+(G, v_i), v_i \in P \} \).

We denote variables that follow the binomial distribution with \( n \) independent trials and probability \( p \) as \( B(n, p) \). For a binomial random variable \( X \sim B(n, p) \), \( E\left[\frac{1}{X+1}\right] \leq \frac{1}{E[X]} \). If \( X \) is a binomial random variable with parameters \( n \) and \( p \), then by applying a Chernoff Bound the following holds for \( \delta \geq 0 \),

\[
\Pr [ X \geq (1 + \delta)np ] \leq e^{-\frac{\delta^2 np}{2+\delta}}.
\]

4.3 Unweighted Sequential Hopsets Algorithm

In this section we will describe the sequential algorithm for hopsets. The algorithm, called \( \text{Hopset}(G) \), takes as input a graph \( G \), has parameters \( k, \lambda \) and \( L \), and outputs a \( (\beta = n^{1/2+o(1)}, \epsilon) \)-hopset.

The main idea of the algorithm is to perform distance limited graph searches from nodes and add edges to the hopset to reached nodes. The algorithm is recursive and partitions the graph into subgraphs to be recursed on. We have two special types of nodes. First, shortcutters search both forwards and backwards to add edges to the hopset. Second, shortcutters (which are a subset of the pivots) search forwards and backwards to label nodes, and the labels are used to partition into subgraphs for recursion. The final output is the set of edges added to the hopset in each level of recursion.

4.3.1 Parameters

The three parameters of the algorithm are \( k, \lambda, \) and \( L \). The parameter \( k \) tunes the probability a node becoming a pivot at a particular level of recursion. The parameter \( \lambda \) is set to be a constant and controls the probability of success for the whole algorithm.
$L$ sets the level of recursion at which a node becomes a shortcutter. This parameter affects the approximation and the runtime. Increasing $L$ gives a better approximation but also increases the runtime.

4.3.2 Pivots and Shortcutters

In each level of recursion, some nodes are assigned the roles of pivots or shortcutters. The label $\ell(v)$ is assigned to each node $v$ and sets the level of recursion at which the node becomes a pivot. Nodes become a shortcutter $L$ levels earlier (or maximum of level 0). A node $v$ is assigned $\ell(v) = i$ with probability $(\lambda k^{i+1} \log n)/n$, where the minimum successful setting is selected.

The roles of pivots and shortcutters are as follows. Pivots search forwards and backwards in the graph to a limited distance. For each reached node, the pivot adds a label indicating that it was reach in the forwards, backwards or both directions. These labels are later used for partitioning the graph into subgraphs for recursion. Shortcutters also search forwards and backwards in the graph to a limited distance. This time, for each reached node the algorithm adds an edge from the shortcutter to the reached node with weight equal to the distance from the shortcutter to the reached node. Likewise for nodes reached in the backwards direction, an edge is added from the reached node to the shortcutter with weight equal to the distance from the reached node to the shortcutter.

4.3.3 Fringe Nodes

Since the searches are limited in distance there are some nodes that are on the boundary of the searches. We call these nodes fringe nodes. In order to not break up the graph too much, we duplicate fringe nodes and add them to multiple subproblems.
4.3.4 Search Distances

The pivots and shortcutters both search to a limited distance. The search distances decrease with each level of recursion. Within each level of recursion, there is a range of distances a pivot may search from. The ranges are disjoint and decrease with each level of recursion. The actual search distance is chosen from the range in such a way as to minimize the number of fringe nodes. Note that each pivot may search from a different distance to minimize fringe nodes with respect to that pivot. Shortcutters also search to a limited distance, however the distances are all the same within each level of recursion.

4.3.5 Description of the Hopset Algorithm

The algorithm, \textsc{Hopset}(G), shown in Algorithms 3 and 4, takes as input a unweighted graph \( G \) and will output a set of edges \( H \). The set \( H \) is initially set to empty. The whole algorithm is repeated \( \lambda \log n \) times to increase the probability of success. In Line 4, the algorithm attempts to shortcut paths of length \( 2^{j-1} \) to \( 2^j \) for \( j \in [(\log n)/2, \log n] \).

Next the algorithm assigns a level of recursion to each vertex at which the vertex will become a pivot. Each node \( v \) sets \( \ell(v) \) to \( i \) with probability \( (\lambda k^{i+1} \log n)/n \), taking the minimum of successful settings.

For each node \( v \) where \( \ell(v) \leq L \) the algorithm searches forwards and backwards to find the sets \( R_{2j+1}^+(G, v) \) and \( R_{2j+1}^-(G, v) \). For each reached node \( u \) in the forwards direction, the edge \( (v, u) \) is added to \( H \) with weight \( \text{dist}(v, u) \). Symmetrically, for nodes \( u \) reached in the backwards direction, the edge \( (u, v) \) is added to \( H \) with weight \( \text{dist}(u, v) \).
Next the algorithm calls the recursive subroutine HSRecurse, $D = 2^j k^{-c}, r = 0$, shown in Algorithm 4. After the algorithm finishes the recursion all of the hopset edges are unioned and returned as the set $H$.

4.3.6 HSRecurse Subroutine

The recursive subroutine HSRecurse takes as input a graph $G$, a distance $D$, and a level of recursion $r$. It returns a set of edges $H$, which is initially set to empty. Each vertex $v$ where $\ell(v) \leq r$ is a pivot for this level of recursion. Each pivot $v$ chooses $\sigma_v$ uniformly at random from $[1, 4\lambda^2 k \log^2 n]$. Next $\rho_v$ is chosen to minimize the number of nodes exactly $\rho_v * D_r$ distance away, for $\rho_v \in [16\lambda^2 k \log^2 n + 4k(\sigma_v - 1), 16\lambda^2 k^2 \log^2 n + 4k\sigma_v]$. In particular, $\rho_v$ is chosen from the specified range to minimize $|R_{(\rho_v + 1)D_r}(G, v) \setminus R_{(\rho_v - 1)D_r}(G, v)|$. Next the pivot searches forwards and backwards to distance $\rho_v D_r$. The labels $v^{Des}$ and $v^{Anc}$ are added to nodes reached in the forwards and backwards directions, respectively. Any node that is reached in both directions gets the label $X$. Fringe nodes are defined to be nodes that are distance $(\rho_v - 1)D_r$ to $(\rho_v + 1)D_r$ away from $v$. The algorithm recurses on the set of fringe nodes with level of recursion $r + 1$ and the same distance $D$. Note that nodes added to the fringe subproblem are nodes at the specified distance that are reached in either the forwards or backwards direction.

Each node $v$ where $\ell(v) \leq r + L$ is a shortcutter in this level of recursion. Each shortcutter searches forwards and backwards to distance $32\lambda^2 k^2 D_r \log^2 n$. For each reached ancestor $u$, the algorithm adds the edge $(u, v)$ to $H$ with weight $dist_G(u, v)$. Symmetrically an edge $(v, u)$ is added to $H$ for each reached descendant $u$ with weight $dist_G(v, u)$.

Any nodes that are reached in both directions by a pivot $v$ are removed from the graph. Nodes that are reached in both directions have an $X$ label, so any nodes
with an $X$ label are removed. Note that each pivot $v$ itself is removed because $v$ is a descendant and an ancestor of itself. Next the algorithm partitions the nodes into sets $V_1, V_2, ... V_t$ based on their labels. This step is described below. For each set of nodes $V_i$, the algorithm recurses on the induced subgraph $G[V_i]$ with level of recursion $r + 1$, and distance $D$. Finally the set of edges $H$ is returned as output to the subroutine.

4.3.7 Partition Based on Labels

Line 15 of Algorithm 4 partitions the nodes into sets of vertices. Specifically, partition the nodes in to maximal subgroups such that each pair of nodes has exactly the same set of labels. For example, two nodes are in the same group if they have $v^{Anc}$, $w^{Des}$ and no labels from $u$ or $x$. The ordering of the sets $V_1, V_2, ...$ does not matter.

Algorithm 3 Hopset algorithm for unweighted directed graphs. $L, \lambda$ and $k$ are parameters.

1: function $\text{Hopset}(G = (V, E))$
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 $i$, break if setting successful.
8: if $\ell(v) \leq L$ then
9: for each $u \in R_{2j+1}^+(G, v)$ add $(v, u)$ to $H$ with weight $dist_G(v, u)$
10: for each $u \in R_{2j+1}^-(G, v)$ add $(u, v)$ to $H$ with weight $dist_G(u, v)$
11: $H \leftarrow H \cup \text{HSRecurse}(G, D = 2^j k^{-c}, r = 0)$
12: return $H$

4.4 Hopsets Analysis

In this section we will analyze the algorithm presented in Section 4.4.1. Our goal is to show the following theorem.
Algorithm 4 Recursive subroutine for Hopset Algorithm. $L$, $\lambda$ and $k$ are parameters.

1: function HSRECURSE($G, D, r$)
2: \[ D_r \leftarrow D / (\lambda^r k^{r/2}) \], \quad H \leftarrow \emptyset
3: \textbf{for each} \ v \in V \text{ with } \ell(v) = r \\textbf{do}
4: \quad \text{Choose } \sigma_v \text{ uniformly at random from } [1, 4\lambda^2 k \log^2 n]
5: \quad \text{Minimize } |R(\rho_v+1)_{D_r}(G, v) \backslash R(\rho_v-1)_{D_r}(G, v)| \text{ 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: \quad \textbf{for each} \ u \in R^+_{\rho_v D_r}(G, v) \text{ add label } v^{\text{Des}} \text{ to vertex } u
7: \quad \textbf{for each} \ u \in R^-_{\rho_v D_r}(G, v) \text{ add label } v^{\text{Anc}} \text{ to vertex } u
8: \quad \textbf{for each} \ u \in R^+_{\rho_v D_r}(G, v) \cap R^-_{\rho_v D_r}(G, v) \text{ add label } X \text{ to vertex } u
9: \quad V^{\text{fringe}}_v \leftarrow R(\rho_v+1)_{D_r}(G, v) \backslash R(\rho_v-1)_{D_r}(G, v)
10: \quad \text{add label } X \text{ to vertex } u
11: \quad \textbf{for each} \ v \in V \text{ with } \ell(v) = r + L \\textbf{do}
12: \quad \textbf{for each} \ u \in R^+_{32\lambda^2 k^2 D_r \log^2 n}(G, v) \text{ add } (v, u) \text{ to } H \text{ with weight } \text{dist}_G(v, u)
13: \quad \textbf{for each} \ u \in R^-_{32\lambda^2 k^2 D_r \log^2 n}(G, v) \text{ add } (u, v) \text{ to } H \text{ with weight } \text{dist}_G(u, v)
14: \quad \textbf{for each} \ u \in V \text{ that has a } X \text{ label, remove } u
15: \quad V_1, V_2, ..., V_t \leftarrow \text{partition based on labels}
16: \quad \textbf{for each} \ i \in [1, t] \\textbf{do}
17: \quad \quad H \leftarrow H \cup \text{HSRECURSE}(G[V_i], D, r + 1)
18: \quad \textbf{return } H

69
**Theorem 4.4.1.** There exists a randomized algorithm which given a n-node, m-edge directed graph $G$ outputs a $(n^{1/2+o(1)}, \epsilon)$-hopset with high probability. The algorithm runs in in $\tilde{O}(m/\epsilon^2)$ time, and the size of the hopset is $\tilde{O}(n/\epsilon^2)$.

First we will show the runtime and hopset size. Then in Section 4.4.2 we will show the hopbound, and finally in Section 4.4.3 we will show the approximation, and the proof of Theorem 4.4.1.

### 4.4.1 Runtime and Hopset Size

Here we will show the runtime of the algorithm and size of the hopset, which are captured by the following theorem.

**Theorem 4.4.2.** For a unweighted directed graph $G$ with $n$ nodes and $m$ edges, $\text{HOPSET}(G)$ outputs a hopset of size $O(nk^{L+1} \log^4 n)$ and runs in $O(mk^{L+1} \log^4 n)$.

Both the runtime and the hopset size rely on some key claims about how many nodes are explored in the searches. For the runtime, the cost of the searches dominates and can be bounded by the number of nodes reached. For the size of the hopset, an edge is added to the hopset when a node is reached in the search. Therefore bounding the nodes reached in the searches also bounds the number of edges added to the hopset.

The proof follows the same idea as JLS [41]. They are able to show that for each node the number of related ancestors and descendants is bounded and decreases with each level of recursion. Lemma 4.4.3 shows the same result as JLS. Next we bound the total number of fringe nodes. Fringe nodes are duplicated so this is necessary for the next claim which is a bound on the total number of nodes in all recursive subproblems. Following JLS, we then show the number of related pivots and shortcutters each node has. This part of the proof is almost the same as JLS except that we have more
shortcutters than they do. Once we have the total related shortcutters and pivots as well as the total number of nodes and edges in all recursive subproblems we are able to show the main theorem.

We begin by showing the same lemma and a similar proof as JLS on the number of related ancestors and descendants in each level of recursion. The main idea of the proof is as follows. There are two types of recursive subproblems to consider, and we will show that in each, the number of related ancestors and descendants decreases with each level of recursion. First there are fringe subproblems, which are called in Line 10 and only contain fringe nodes. The algorithm chooses a distance that minimizes the number of nodes in the fringe problem which allows us to bound the number of nodes added to fringe subproblems thus upper bounding the number of related nodes in the subproblem. Second, there are the main recursive calls which are called in Line 17. This part of the proof leverages the fact that the probability of being a pivot increases with each level of recursion. In particular, if a node has many ancestors then it is likely that some of the ancestors will be pivots. These pivots break up the graph leaving less related ancestors in the next level of recursion. The proof gets quite a bit complicated due to possible cycles in the graph, but we are able to show the number of ancestors is reduced in each level of recursion. A symmetric argument applies for descendants.

**Lemma 4.4.3.** For a graph $G = (V, E)$ with $n$ nodes and $m$ edges, consider a full execution of HSRecurse($G, d, 0$). The following holds for each recursive call of HSRecurse($G, D, r$) for all $v \in V$ with probability $1 - n^{-7\lambda + 3}$,

$$|R^+_{p_{max}D_r}(G, v)| \leq nk^{-r}, |R^-_{p_{max}D_r}(G, v)| \leq nk^{-r}.$$
Proof. We prove this lemma by induction on level of recursion \( r \). The base case is \( r = 0 \), and trivially, there are at most \( n \) related ancestors and descendants for each \( v \in V \).

When \( r > 0 \), we distinguish between two types of recursive subproblems. First, there are core problems which are called in Line 17, \( \text{HSRecurse}(G[V_i], D, r + 1) \). Second, there are recursive subproblems called on fringe nodes, in Line 10, which are \( \text{HSRecurse}(G[V_u^{\text{fringe}}], D, r + 1) \). We will show these two cases separately, beginning with the fringe problems.

Each fringe subproblem \( \text{HSRecurse}(G[V_u^{\text{fringe}}], D, r + 1) \) at level of recursion \( r + 1 \) is called by some node \( u \). By the inductive assumption, \( |R_{\rho_{\max}} D_r (G, u)| \leq 2nk^{-r} \). The pivot \( u \) selects a distance \( \rho_u \) from a range of size \( 4k \) that minimizes the number of fringe nodes, i.e., the set \( |R_{(\rho_u+1)D_r} (G, v) \setminus R_{(\rho_u-1)D_r} (G, v)| \). There are a total of \( 4kD_r \) distances and the fringe nodes will be nodes at some distances in range of size \( 2D_r \). Since the distance chosen minimizes the number of nodes at this range, the total number of nodes is reduced by at least \( 4kD_r/(2D_r) = 2k \). In total the size of the fringe subproblem is at most \( 2nk^{-r}/(2k) = nk^{-r-1} \), and so each node has at most \( nk^{-r-1} \) related ancestors.

Next we turn to the core problems. We will show the the number of related ancestors and the claim holds for descendants by a symmetric argument. First we will show the claim for acyclic graphs. Adding cycles complicates the proof and we will show that case after. Let \( A = R_{\rho_{\min}} D_r (G, v) \) be the set of ancestors of \( v \) after

Assume \( |A| > nk^{-r-1} \), otherwise the lemma holds immediately. Consider a topological ordering of the nodes in \( A \), \( \langle u_1, u_2, ... u_{|A|} \rangle \). If \( u_i \) is a pivot then any \( u_j \) with \( j \leq i \) either gets no label from \( u_i \), or gets the label \( u_i^{\text{Anc}} \), while \( v \) will only get the label \( u_i^{\text{Des}} \) from \( u_i \) since the graph is acyclic. Therefore, if \( u_i \) is a pivot then any node \( u_j \), for \( j \leq i \), is in a different subproblem than \( v \). Next, if some node \( u_i \) is a pivot in
\[ \langle u_1, u_2, \ldots, u_{|A|} \rangle \text{ where } i \geq |A| - nk^{-r-1}, \text{ then there will be at most } nk^{-r-1} \text{ ancestors of } v \text{ in the next level of recursion.} \]

The probability that no node \( u_i \text{ for } i \geq |A| - nk^{-r-1} \) is a pivot is,

\[
(1 - \frac{\lambda k^{r+1} \log n}{n})^{nk^{-r-1}} \leq e^{-\lambda \log n} \leq n^{-1.4\lambda}.
\]

Next we show the case where there are cycles in the graph. The proof gets considerably more complicated but the main idea is the same from the acyclic case. Again, let \( A = R_{\rho_{\min}D_r}(G, v) \), and let \( \langle A_1, A_2, \ldots \rangle \) be the strongly connected components of \( A \) in topologically sorted order. This case becomes more challenging because there is no order between nodes that are strongly connected. Moreover it possible to order \( A_i = \langle u_1, u_2, \ldots \rangle \) such that there exists a node \( u_j \) where if \( u_j \) is a pivot at level \( r \), \( u_j' \) gets label \( u_j^{Des} \) and \( j' < j \). This implies that the argument for acyclic graphs does not hold for graphs with cycles.

For the cyclic case, we want to count the number of nodes in a strongly connected component \( A_i \) could be in \( R_{\rho_{\min}D_r}(G', v) \). We claim that for any \( A_i \), there can be at most \( n^{-k-1}/2 \) nodes in \( R_{\rho_{\max}D_r}(G', v) \) with probability \( n^{-0.7\lambda+1} \). We will show this claim later in the proof.

Now consider the latest index \( j \) of the strongly connected component where at least half of the nodes are ordered at or after \( A_j \). Specifically, \( j \) is the index, where \( |\bigcup_{i \geq j} A_i| \leq nk^{-r-1}/2 \) and \( |\bigcup_{i \geq j} A_i| > nk^{-r-1}/2 \). If \( |\bigcup_{i \geq j} A_j| \leq nk^{-r-1}, \) then by the same reasoning as the acyclic case, \( R_{\rho_{\min}D_r}(G', v) \leq nk^{-r-1} \) with probability \( 1 - n^{-0.7\lambda} \). The only difference in this case we use \( n^{-k-1}/2 \) instead of \( n^{-k-1} \). Otherwise \( |\bigcup_{i \geq j} A_j| > nk^{-r-1}, \) and so \( |A_j| \geq n^{-k-1}/2 \). Given our claim, there will be at most \( n^{-k-1}/2 \) nodes from \( A_j \) that are in \( R_{\rho_{\min}D_r}(G', v) \) and any \( A_j', \) where \( j' < j, \) will be not in \( R_{\rho_{\min}D_r}(G', v) \). Therefore with probability \( 1 - 2n^{-0.7\lambda+1}, R_{\rho_{\min}D_r}(G', v) \leq nk^{-r-1} \).

We can take a union bound over nodes \( v \in V \) and all levels of recursion \( r, \) and the lemma holds for the core problems with probability \( 1 - n^{-0.7\lambda+3} \).
It remains to show our claim that for any any strongly connected component $A_i$, there will be at most $n^{−k−1}/2$ nodes in $R_{ρ\max \mathcal{D}_r}(G', v)$ with probability $n^{−0.7λ+1}$.

Consider two nodes $u_j, u_{j'} \in A_i$ in a strongly connected component $A_i$. Define the relation $\mathcal{R}$ as $\mathcal{R}(u_j, u_{j'}) = 1$ if $dist_{G'}(u_j, u_{j'}) \geq dist_{G'}(u_{j'}, u_j)$, and $\mathcal{R}(u_j, u_{j'}) = 0$ if $dist_{G'}(u_j, u_{j'}) < dist_{G'}(u_{j'}, u_j)$. Based on the definition, if $\mathcal{R}(u_j, u_{j'}) = 0$ then $\mathcal{R}(u_{j'}, u_j) = 1$. If $\mathcal{R}(u_{j'}, u_j) = 1$ and $u_{j'}$ is a pivot at level $r$, then $u_j$ gets $u_{j'}^{\text{Des}}$ label only if $u_j$ gets $u_{j'}^{\text{Anc}}$ label. Also if $u_{j'}$ is chosen as a pivot, $v$ gets only $u_{j'}^{\text{Des}}$ label. Thus, if $\mathcal{R}(u_{j'}, u_j) = 1$ and $u_{j'}$ is a pivot, then $u_j$ is not in the same subproblem as $v$.

Next consider the set $T(u_j) = \{u_{j'} \in A_i \mid \mathcal{R}(u_{j'}, u_j) = 1\}$ for a node $u_j \in A_i$, which is the set of nodes where $u_{j'} \in T(u_j)$ being a pivot implies that $u_j$ is not in the same subproblem as $v$. By construction, if $|T(u_j)| > nk^{−r−1}/2$, then $u_j$ will be in $R_{ρ\min \mathcal{D}_r}(G', v)$ with probability,

$$(1 - \frac{λk^{r+1} \log n}{n})n^{−r−1}/2 \leq n^{−0.7λ}.$$ 

Define the set $S = \{u_j \mid |T(u_j)| \leq nk^{−r−1}/2\}$. By the reasoning above, for a node $u \in A_i$, if $u \notin S$, then $u \notin R_{ρ\min \mathcal{D}_r}(G', v)$ with probability $1 − n^{−0.7λ}$. If $u_j \in S$ is a pivot, then there will be at most $nk^{−r−1}/2$ nodes $u_{j'}$ where $\mathcal{R}(u_{j'}, u_j) = 1$, which implies that there can be at most $nk^{−r−1}/2$ nodes where $\mathcal{R}(u_j, u_{j'}) = 0$. Thus, if $u_j$ is a pivot, there will be at most $nk^{−r−1}/2$ nodes that get the $u_{j'}^{\text{Des}}$ label and the rest of the nodes in $A_i$ will either get $X$ or no label from $u_j$.

Now we consider two cases of sizes for $S$. First, if $|S| \leq nk^{−r−1}/2$, then at most $nk^{−r−1}/2$ nodes in $A_i$ will be in $R_{ρ\min \mathcal{D}_r}(G', v)$. Otherwise $|S| \geq nk^{−r−1}/2$, in which case a node in $S$ is a pivot with probability

$$(1 - (1 - \frac{λk^{r+1} \log n}{n})n^{−r−1}/2) \geq 1 - n^{−0.7λ}.$$ 

If there is a node in $S$ that is a pivot then there will be at most $nk^{−r−1}/2$ nodes in $R_{ρ\min \mathcal{D}_r}(G', v)$. Therefore, in either case of the size of $S$, there will be at most
most $nk^{-r-1}/2$ nodes of a strongly connected component $A_i$ in $R_{\rho_{\text{min}}D_r}^-(G', v)$ with probability $1 - n^{-0.7\lambda + 1}$.

The next part of the analysis shows the expected number of nodes added to fringe problems for any vertex. The proof leverages the previous lemma and that the algorithm chooses the distance from a range that minimizes the number of fringe nodes, where the range is chosen uniformly at random from a larger range. JLS has a similar lemma however the proof differs because we choose the fringe distance to minimize nodes added to the fringe subproblems. Both of our analyses rely on the large distance range to argue that the fringe subproblems are small.

**Lemma 4.4.4.** Consider a call to HSRecurse($G', D, r$) at some level of recursion $r$ and any vertex $v \in G$. For $v$ the nodes in the fringe subproblem are nodes in $|R_{(\rho_v+1)D_r}(G', v)\setminus R_{(\rho_v-1)D_r}(G', v)|$ where $\rho_v$ is chosen as described in the algorithm. The expected number of nodes in $v$’s fringe subproblem is $1/(4\lambda k \log n)$.

**Proof.** If node $v$ is not a pivot then define the size of the fringe subproblem to be zero. The range for $\rho_v$ is $4k$, which is chosen uniformly at random from a larger range of size $4\lambda^2k \log^2 n$. The algorithm chooses $\rho_v$ to minimize the nodes that at distance in the range $((\rho_v - 1)D_r, (\rho_v + 1)D_r]$. By Lemma 4.4.3, the number of related nodes at level $r$ is $2nk^{-r}$. In total the expected number of nodes in $v$’s fringe problem at level $r$ is $2nk^{-r} \cdot 1/(4\lambda^2k \log^2 n) \cdot 2/(4k)$. By multiplying by the probability that $v$ becomes a pivot at level $r$, we get the expected number of nodes added to the fringe problem is $1/(4\lambda k \log n)$. \qed

Now we are ready to bound the total number of nodes in all recursive subproblems. The core problems form a partition of the nodes in the previous level of recursion. The fringe nodes are duplicated which is a problem, but the previous lemma showed
that the number of nodes in the fringe problems is small so we are able to bound them. This lemma is based off of Lemma 5.3 and Corollary 5.5 from JLS [41].

Lemma 4.4.5. Consider a full execution of \textsc{Hopset}(G). The expected number of nodes and edges in all recursive executions of \textsc{Hsrecurse}(G', D, r) are $2n \log n$ and $2m \log n$, respectively.

Proof. We will prove the total number of nodes, and the edges is the same. In a single call of \textsc{Hsrecurse}(G' = (V', E'), D, r), the vertices called in recursive subproblems are those called in the fringe problems and those in the core problems. By construction, the vertices called in core problems form a partition of the nodes in $G'$ (with some possibly removed by getting label $X$). By Lemma 4.4.4, the expected size of each pivot’s fringe problem is $1/(4\lambda k \log n)$. In total, the number of nodes added to recursive subproblems is $|V'|(1 + 1/(4\lambda k \log n))$. By summing over all levels of recursion, and if $k \geq \log n$, then the total expected number of nodes in all recursive subproblems is,

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

In the next two lemmas we will show the number of related pivots and shortcutters each node has. The first lemma is the same as JLS [41], while the second is slightly different due our algorithm having more shortcutters than theirs. The main idea of the proofs are the same. With each level of recursion the number of pivots and shortcutters increases, however the graph is broken up into smaller subgraphs and each node has less total related ancestors and descendants. By considering the probability of a node becoming a shortcutter or pivot, and the related ancestors in each level of recursion we are able to show the bounds.
Lemma 4.4.6. Consider a full execution of $\text{Hopset}(G)$, and all the recursive calls to $\text{HSRecurse}(G', D, r)$. For each $v \in V$, there at most $6\lambda k \log n$ pivots $u$ where $v \in R(G', (\rho_u + 1)D_r, u)$, with probability at least $1 - n^{-0.7\lambda+4}$.

Proof. Consider a call to $\text{HSRecurse}(G', D, r)$ that contains $v$ at level $r$. If $v$ is related to a pivot $u$ at distance $(\rho_u + 1)D_r$, then $u \in R(G', \rho_{\max}D_r, v)$. By Lemma 4.4.3, $|R(G', \rho_{\max}D_r, v)| \leq 2nk^{-r}$ with probability $1 - n^{-0.7\lambda+3}$. The number of pivots follows a binomial distribution $B(|R(G', \rho_{\max}D_r, v)|, \frac{\lambda k^{r+1} \log n}{n})$. Therefore we have,

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

Taking a union bound over all $v \in V$, and all levels of recursion $r$ completes the proof.

Next we will show the number of related shortcutters. Here we add in the shortcutters that are added to the graph in $\text{Hopset}(G)$ before any calls to $\text{HSRecurse}$. The proof for the shortcutters in recursive calls to $\text{HSRecurse}(G', D, r)$ the same as the previous lemma except the probability of being a shortcutter is different that a pivot. This higher probability leads to there being more shortcutters than pivots, and thus more shortcutters to related to each node.

Lemma 4.4.7. Consider a call to $\text{Hopset}(G)$. For all $v \in V$, the number of shortcutters $u$ where $v \in v \in R(G, n, u)$ is $6\lambda k^{L+1} \log n$ with probability at least $1 - n^{-0.7\lambda+4}$.

Next consider the recursive calls to $\text{HSRecurse}(G', D, r)$. For all $v \in V$, the number of shortcutters $u$ where $v \in v \in R(G', \rho_{\max}D_r, u)$ is $6\lambda k^{L+1} \log n$ with probability at least $1 - n^{-0.7\lambda+4}$.

Proof. First we’ll show the lemma for the shortcutters in $\text{HSRecurse}(G', D, r)$, and then for shortcutters in $\text{Hopset}(G)$. This first part of the proof is the same as the previous lemma except that the probability of being a shortcutter at level $r$ is
\( \lambda k^{L+r+1} \log(n)/n \) which is higher than it was for being a pivot. By the same argument as the previous lemma, the number of shorteners \( u \) where \( v \in R(G', \rho_{\max} D_r, u) \) is 
\[ 6\lambda k^{L+1} \log n \]
with probability at least \( 1 - n^{-0.7+4} \).

For shorteners in \( \text{Hopset}(G) \), nodes \( v \in V \) are shorteners if \( \ell(v) \leq L \). The probability of being a shortener for each \( v \in V \) is 
\[ \sum_{i=0}^{L} \lambda k^{i+1} \log(n)/n \leq 2\lambda k^{L+1} \log(n)/n, \] for \( k \geq 2 \). The number of shorteners follows a binomial distribution \( B(n, 2\lambda k^{L+1} \log(n)/n) \). Finally we get,
\[
\Pr[B(n, 2\lambda k^{L+1} \log(n)/n) > 6\lambda k^{L+1} \log(n)] \leq e^{-2\lambda k \log(n)} \leq n^{-2\lambda}
\]
by applying a Chernoff bound.

Now we are ready to show the proof of Theorem 4.4.2 which bounds the size of the hopset and the runtime of the algorithm.

**Proof of Theorem 4.4.2.** The cost of assigning levels \( \ell(v) \) to nodes is linear. To perform the searches from pivots and shorteners we can run BFS which has a cost of the total number of edges explored. For the pivots, the number of edges explored is the number of pivots times the number of edges each pivot explores. By Lemmas 4.4.5 and 4.4.6, the total edges explored is \( 2m \log n \cdot 6\lambda k \log n = O(m \lambda k \log^2 n) \). Recall that \( \lambda \) is a constant, which will be set later. Likewise for the shorteners, the total edges explored is \( O(m \lambda k^{L+1} \log^2 n) \), by Lemmas 4.4.5 and 4.4.7. The partitioning step can be done by sorting the nodes based on their label. The algorithm is run once for \( \Theta(\log n) \) distance settings, and is repeated \( \lambda \log n \) times. In total the runtime of the algorithm is \( O(m \lambda k^{L+1} \log^4 n) \).

The size of the hopset is the number of edges added, which is the number of nodes reached in the searches by shorteners. Lemma 4.4.5 shows the number of nodes in all recursive subproblems is \( 2n \log n \), and Lemma 4.4.7 shows that each node is related
to $6\lambda k^{L+1} \log n$ pivots. Multiplying these together and by the number of repetitions of the algorithm gives $O(nk^{L+1} \log^4 n)$ total hopset edges.

4.4.2 Hopbound

The goal of this section is to show the algorithm in Section 4.3, Hopset($G$), outputs a hopset with hopbound $\beta = n^{1/2+O(1/\log k)}k^{c+1/2-L/2} \log^2 n$. We consider a shortest path $P$ with $|P| > \beta$ throughout the full execution of the algorithm, and show this path is shortcutted to a new path with at most $\beta$ hops. In the next section, we’ll show this new path is a good approximation of $P$.

The high level idea of the argument follows Fineman [37] and JLS [41]. We consider the set of subproblems that contain nodes on the path throughout the execution of the algorithm. Define a path-relevant subproblem to be a subproblem HSRecurse($G'$, $D$, $r$) where $G'$ contains any nodes of path $P$.

In a path-relevant subproblem we characterize nodes by their relation to the path with respect to a distance $D$. An ancestor is a node $v$ such that, $v \in R^-_d(G, P) \setminus R^+_d(G, P)$, a descendant is a node $v$ such that $v \in R^+_d(G, P) \setminus R^-_d(G, P)$, and a bridge is a node $v$ such that $v \in R^-_d(G, P) \cap R^+_d(G, P)$. Note that nodes on the path are bridges. Any other nodes in the subproblem are unrelated to the path.

When a bridge is selected as a shortcutter, the bridge shortsctuts the path to two hops. If an ancestor or a descendant is selected as a pivot it breaks the path into two subpaths which are in different subproblems in the next level of recursion. This is problematic because in the next level, we now need a shortcutter in both subproblems to shortcut the path, and can only shortcut the path to four hops. While we don’t want to break up the path too much, Fineman and JLS show that we still make progress. When an ancestor or descendant is selected as a shortcutter, the number of ancestors and descendants related to the path decreases in the next level of recursion.
This means that as we recurse the number of ancestors and descendants decrease making us more likely to select a bridge shortcutter. Nodes unrelated to the path don’t affect the path in the next level of recursion.

We will follow the path relevant subproblems as we go through the algorithm in the following way. Define a *path-relevant subproblems tree* to be a tree of subproblems for a path $P$. The root of the tree is the whole path $P$ which is contained in level-0 subproblem. For each node in the tree, if a bridge is selected as a pivot then the node is a leaf and has no children. If no bridge is selected, then the node has a child node for each path-relevant subproblem containing subpaths of $P$. After the full execution of the algorithm, the leaves of the tree contain the full path $P$. In Lemma 4.4.8 we construct the path-relevant subproblems tree. Lemma 4.4.8 leverages Lemma 4.4.9 which shows a similar claim to one that JLS and Fineman have. Namely, that as we recurse the number of ancestors and descendants decrease with each level of recursion.

The construction of the path-relevant subproblems tree is different than Fineman’s and JLS’s because in our algorithm, the search distance decreases with each level of recursion. A pivot may stop its search in the middle of the path which splits the path into an additional subproblem. Also, the algorithm duplicates the fringe nodes and calls fringe and core subproblems. In order to analyze the final length of the path, we choose which subproblems to count, and which to ignore. We can ignore some as long as the nodes were copied to another subproblem.

In order to handle the decreasing search distance, we divide certain path-relevant subproblems into *logical path-relevant subproblems*. Even though the algorithm searches past a certain point, we will ignore shortcuts added past this point. In a way, this hurts our analysis because we are shortcutting shorter paths, however this makes it easier to cope with the decreasing search distance. The search distance decreases by $O(\sqrt{k})$ with each level of recursion. The logical subproblems are split
such that the length of each subpath is less than the length of the next subproblems
basic search distance, \(D_r\). Note that the real search distance is a scalar times \(D_r\),
but subpaths of length \(D_r\) are sufficient. Splitting the path into \(D_r\) length subpaths
creates \(O(\sqrt{k})\) new subproblems. Since they are length \(D_r\), it is ensured that any
bridge shortcutters will reach the ends of the subpath in the next level of recursion.
For non-path nodes in the problem, we copy them and add them to the appropriate
logical subproblem.

Consider a path relevant subproblem at level \(R\) and the subpath \(P = \langle v_i, v_{i+1}, \ldots v_j \rangle\)
contained within. If \(j - 1 > D_r\) then we partition \(P\) into \(q = \lceil j - i/D_r \rceil\) subpaths
\(P_1, P_2, \ldots P_q\), each of which has length \(D_r\), except for possibly the last one which may
be shorter. Since \(D_r = \frac{D_r}{\lambda k^{r/2}}\), this partition divides the path into at most \(\lambda k^{r/2}\)
subpaths.

**Lemma 4.4.8.** Consider a logical path-relevant subproblem \((G', P = \langle v_0, v_1, \ldots, v_\ell \rangle, r)\)
corresponding to a call to \(\text{HSRECURSE}(G', D, r)\). Let \(p_r = (\lambda k^{r+1} \log n)/n\) be the prob-
ability a vertex is a pivot at level \(r\). Let \(S = \{v \mid \ell(v) = r, v \in R_{v \rho v D_r}(G', P)\}\) be the set
of pivots at level \(r\) that are \(\rho v D_r\)-related to \(P\). There exists subpaths \(P_0, P_1, P_2, \ldots, P_2|S|\)
such that the following hold.

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 union of all nodes of \(P_i\) for
   \(0 \leq i \leq 2|S|\) is \(P\).

3. Each subpath, \(P_0, P_1, P_2, \ldots, P|S|+1\) are in core problems and each \(P_i\) is contained
   in some \(V_{a_i}\).

4. Each subpath, \(P|S|+1, \ldots, P_2|S|\) are called in fringe problems and each \(P_i\) is con-
   tained in some \(V_u^\text{Fringe}\), where \(u \in S\).
Additionally, with probability $1 - n^{-0.7\lambda+4}$, we have that

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

and

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

**Proof.** Let $u$ be a node and suppose $u \preceq_{\rho_{u}D_r} P$ or $P \preceq_{\rho_{u}D_r} u$. In this case $u$ will add a $u^{Anc}$ or $u^{Dec}$ label to at least one node on $P$. If $u \preceq_{\rho_{u}D_r} P$ and $P \preceq_{\rho_{u}D_r} u$, then $u$ is a bridge and so there are no path-relevant subproblems. Otherwise, $u$ does not add a label to any vertices on $P$.

First we will show the case where all the pivots are ancestors, and then we will extend this case to where there are also descendant pivots. Let $u$ be an ancestor pivot and we will show the partition of the path into three subpaths, which are contained in path-relevant subproblems. There are two core subpaths, one of which has label $u^{Des}$, and another which has no label from $u$. The third subpath is in a fringe subproblem.

Consider the following two nodes on $P$ which are indexed as follows. The first node which we call $Fringe(P, u)$ is the earliest $\rho_{u}D_r$-descendant of $u$ on $P$, $Fringe(P, u) = \min\{i \mid u \preceq_{\rho_{u}D_r} v_i\}$. The second node, called $Core(P, u)$, is the earliest $(\rho_{u} - 1)D_r$-descendant of $u$ on $P$, $Core(P, u) = \min[min\{j \mid u \preceq_{(\rho_{u} - 1)D_r} v_j\}, \ell + 1]$. If there is no node on $P$ which is a $(\rho_{u} - 1)D_r$-descendant of $u$, then set $Core(P, u) = \ell + 1$. For a node $v_j$ on $P$, if it is a $(\rho_{u} - 1)D_r$-descendant of $u$, then $v_j$ is also a $\rho_{u}D_r$-descendant of $u$ so $Fringe(P, u) \leq Core(P, u)$. Now divide $P$ into three subpaths, $P^{Unrelated} = \langle v_0, ..., v_{Fringe(P, u) - 1} \rangle$, $P^{Fringe} = \langle v_{Fringe(P, u)}, ..., v_{Core(P, u) - 1} \rangle$ and $P^{Core} = \langle v_{Core(P, u)}, ..., v_{\ell} \rangle$. For each of these subpaths, its possible that the subpath is empty. If $Fringe(P, u) = 0$, then define the unrelated subpath as empty. If $Fringe(P, u) = Core(P, u)$, then define the fringe subpath as empty. If $Core(P, u) = \ell + 1$, then define the core subpath as empty.

82
Next we will show that the subpaths $P^{Unrelated}$, $P^{Fringe}$ and $P^{Core}$ are contained in path-relevant subproblems. For $P^{Unrelated}$, the subpath gets no labels from $u$ so it is unaffected. Consider a node $v_{i'} \in P^{Fringe}$. Since $i' < Core(P,u)$, $u \not\leq_{(\rho_u-1)D_r} v_{i'}$, because otherwise $u \leq_{(\rho_u-1)D_r} v_{i'}$, which would mean $i' \geq Core(P,u)$. We also have that, $u \not\preceq_{\rho_uD_r} v_{Fringe}(P,u)$ by definition, and $v_{Fringe}(P,u) \not\preceq_{D_r} v_{i'}$ since $P$ is at most length $D_r$, which means $u \not\leq_{(\rho_u+1)D_r} v_{i'}$. Since $u \leq_{(\rho_u+1)D_r} v_{i'}$ and $u \not\preceq_{(\rho_u-1)D_r} v_{i'}$, we have $v_{i'} \not\in R_{(\rho+1)D_r}(G,u) \setminus R_{(\rho-1)D_r}(G,u)$, and so $P^{Fringe}$ is in $u$’s fringe subproblem. For the last subpath $P^{Core}$, we will show that any node $v_{j'} \in P^{Core}$ is a $\rho_uD_r$-descendant of $u$ and therefore gets the $u^{Des}$ label. By definition, $u \leq_{(\rho_u-1)D_r} v_{Core}(P,u)$, and $v_{Core}(P,u) \preceq_{D_r} v_{j'}$ since $P$ is at most length $D_r$, and therefore, $u \leq_{\rho_uD_r} v_{j'}$. Now we have shown that an ancestor pivot $u$ divides the path into up to three subpaths. The subpaths are the unlabeled subpath, the fringe problem and the core problem.

Next we will show the case that there are $t$ ancestor pivots. In particular, we will show that the subpaths defined by the pivots are contained in unrelated, fringe and core problems, and the union of all the subpaths is the original path $P$. For each pivot $u$, define $P^{Fringe}_u = \langle v_{Fringe}(P,u), ..., v_{Core}(P,u) \rangle$. As we showed above, $P^{Fringe}_u$ is in $u$’s fringe problem. Let the $Core(P,u)$ values for all $t$ ancestor pivots in non-decreasing order be $c_1, c_2, ..., c_t$, and for convenience, let $u_1, ..., u_i, ..., u_t$ be the corresponding ancestor pivots. Set $c_0 = 0$ and $c_{t+1} = \ell + 1$. Each vertex on a subpath $\langle v_{c_i}, ..., v_{c_{i+1}-1} \rangle$ gets a label from each pivot $u_1, ..., u_i$. We will change the subpaths to make the analysis easier. Specifically, if a node is in a fringe problem then we will not consider it in the core problem. The $i$-th core subpath is $P_i = \langle v_{c_i}, ..., v_{f_i-1} \rangle$, where $f_i$ is $f_i = min(min\{Fringe(P,u_j) \mid j > i\}, c_{i+1})$. Since $c_i \geq Core(P,u_j)$ for all $j \leq i$, $P_i$ gets a label from each pivot $u_1, ..., u_i$. The last node on $P_i$ is indexed at most at $f_i - 1$ which means that nodes on $P_i$ do not get labels from any pivots after $u_i$. Therefore,
we have that each $P_i$ is in a core problem. Define $P_0 = \langle v_0, \ldots, v_{c_1-1} \rangle$ and $P_i = P_{u_i}^{Core}$. In total there are at most $t + 1$ core problems.

We have shown that the nodes on $P_0$ are in the unrelated subpath, the nodes on $P_i$, for $i \in [1, t]$ are in the core problems. The remaining nodes on $P$ are nodes in in $\langle v_f, \ldots, v_{c_i+1-1} \rangle$, which by definition are in fringe problems. Therefore, the union of all the subpaths $P_0, P_u^{Fringe}$, and $P_{u_i}^{Core}$ for $i \in [1, t]$ is the entire path $P$.

Now that we have shown the case where all the pivots are ancestors, we will consider the case where some pivots are descendants. While adding descendant pivots makes the analysis more complicated, the main idea is the same. First we will define the core and fringe subpaths and show that nodes on these subpaths get the correct labels, and then we will show that the union of the subpaths is the whole path $P$.

For a descendant pivot $u$ we will define $Fringe(P, u)$ to be the index of the last node on $P$ that is a $\rho_uD_r$-ancestor of $u$ i.e. $Fringe(P, u) = \max\{i \mid v_i \preceq_{\rho_uD_r} u\} + 1$. Second, define $Core(P, u)$ to be the latest node on $P$ that is a $(\rho_u - 1)D_r$-ancestor of $u$, $Core(P, u) = \max(\max\{j \mid v_j \preceq_{(\rho_u - 1)D_r} u\}, -1) + 1$. For convenience, both $Fringe(P, u)$ and $Core(P, u)$ are shifted by one index. If there is no node on path $P$ that is a $(\rho_u - 1)D_r$-ancestor of $u$, then set $Core(P, u) = 0$. Similar to the ancestor pivots case, we define all the $P_u^{Fringe}$ subpaths to be 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, $P_u^{Fringe} = \langle v_{Fringe(P, u)}, \ldots, v_{Core(P, u) - 1} \rangle$. The same idea can be applied from the ancestor case to show that $P_u^{Fringe}$ is in $u$’s fringe subproblem.

For the core problems, we again consider the $|S|$ $Core(P, u)$ values in non-decreasing order, $c_1, c_2, \ldots, c_{|S|}$, and let $u_1, \ldots, u_i, \ldots, u_{|S|}$ be the corresponding pivots, which can be ancestor or descendant pivots. Set $c_0 = 0$ and $c_{|S|+1} = \ell$. A path node $v_i$, that is not in a fringe problem, 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 $c_j' \geq i$. 

84
Next we will describe give the subpaths explicitly. First define $f_i$ and $g_i$ as $f_i = \min(\min\{\text{Fringe}(P, u_j) \mid j > i\}, c_{i+1})$ and $g_i = \max(\max\{\text{Fringe}(P, u_j) \mid j \leq i\}, c_i)$. We can now define path $P_i = \langle v_{g_i}, \ldots, v_{f_i-1} \rangle$. It can be 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 to 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 to the “right” of $u_i$. We will show that each node on $P_i$ gets a $\text{Des}_j$ label from each pivot $u_j \in A_i$ and does not get any labels from pivots in $\bar{A}_i$. Since the core value of the nodes in $A_i$ is less than or equal to $c_i$, $P_i$ gets a label from each node in $A_i$, which gives us the first claim. For the second part of the claim, suppose a node $u'$ on $P_i$ gets a label from an 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 contradicts the definition of $f_i$. Thus, we know $P_i$ does not get any labels from $u_j \in \bar{A}_i$.

Now we turn to the descendants case, which is almost the same as the ancestor case except that our definitions of $\text{Fringe}(P, u)$ and $\text{Core}(P, u)$ are shifted by one index. Similarly to the ancestor case, define $D_i = \{u_j \mid j \geq i + 1, u_j \text{ is a descendant pivot}\}$ and $\bar{D}_i = \{u_j \mid j < i + 1, u_j \text{ is a descendant pivot}\}$. The same reasoning from the ancestor case applies to show that all nodes in $P_i = \langle v_{g_i}, \ldots, v_{f_i-1} \rangle$ get an $\text{Anc}$ label from each vertex in $D_i$, and get no labels from any vertex in $\bar{D}_i$. Each subpath $P_i$ is in a core problem, and there are $P_0, P_1, \ldots, P_{|S|}$ subpaths which is at most $|S| + 1$ total subpaths.

The last part of the claim for ancestor and descendant pivots is that the union of the subpaths $P_i$, for $i \in [1, |S|]$, and $P_u^{\text{Fringe}}$, for all $u \in S$, is the whole path $P$. Since $\langle v_{g_i}, \ldots, v_{f_i} \rangle$, for $i \in [1, |S|]$, are in core problems, and by definition, $\langle v_{c_i}, \ldots, v_{g_i-1} \rangle$ and $\langle v_{f_i}, \ldots, v_{c_{i+1}-1} \rangle$ are in fringe problems, we get this part of the claim.

The last part of the lemma is showing the number of path-related nodes at the next level of recursion, $r + 1$. To bound of the number of nodes we will start by
counting related nodes in the core problem and then in the fringe problem. For the
core problem, consider the core subproblems created by $\rho_{\text{min}}D_r$-pivots. All $\rho_{\text{min}}D_r$-
pivots will be in $S$ because the search distance is at least $\rho_{\text{min}} + 1$. Notice that leaving
out some pivots in $S$ only increases the number of related nodes at level $r + 1$, so we
can leave out some pivots in $S$ in our analysis.

Consider the path-related nodes in the core problem at level $r + 1$. Any ancestor
pivot that reaches the path in $\rho_{\text{min}}D_r$, also reaches $v_\ell$ in $\rho_{\text{max}}D_r$ because the length of
$P$ is at most $D_r$. Combining the above with Lemma 4.4.3, with probability $1 - n^{-0.7\lambda + 3}$
we have,

$$|R^-_{\rho_{\text{min}}D_r}(G, P)| \leq |R^-_{\rho_{\text{max}}D_r}(G, v_\ell)| \leq nk^{-r}.$$  

Assume $|R^-_{\rho_{\text{min}}D_r}(G, P)| > 1.5/p_r$, otherwise the claim holds. Let $Y_s$ be the number
of ancestors at level $r + 1$ in a logical path-relevant subproblem $s$, and let $X_s$ be the
$\rho_{\text{min}}D_r$ ancestors pivots in $s$. By the random choice of nodes to be pivots, $X_s$ is a bino-
mial random variable, $B(|R^-_{\rho_{\text{min}}D_r}(G, P)|, p_r)$. Therefore, $E[X_s] = |R^-_{\rho_{\text{min}}D_r}(G, P)| \cdot p_r \leq \lambda k \log n$ with probability $1 - n^{-0.7\lambda + 3}$, where the probability comes from Lemma 4.4.3. By applying a Chernoff bound,

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

and so the number of ancestor pivots is least $4\lambda k \log n$ with high probability. Next
we take a union bound over all nodes $v \in V$ and all levels of recursion $r$ so that the
claim holds with probability $1 - n^{-0.7\lambda + 4}$.

In the algorithm, the scalar interval is $|I| = 4\lambda^2 k \log^2 n$, and if $\lambda \geq 4$, then
$|I| \geq 4X_s$. Lemma 4.4.9 will show that if $|I| \geq 4X_s$ then,

$$E[Y_s \mid X_s] \leq \frac{1.5}{X_s + 1} \cdot |R^-_{\rho_{\text{min}}D_r}(G, P)|.$$
Now we can show the expectation of $Y_s$.

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

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

$$\leq 1.5 |R_{\rho_{\text{min}}D_r}(G, P)| \cdot E[\frac{1}{X_s + 1}]$$

$$\leq 1.5 |R_{\rho_{\text{min}}D_r}(G, P)| \cdot \frac{1}{|R_{\rho_{\text{min}}D_r}(G, P)| \cdot p_r} = \frac{1.5}{p_r}$$

The first equation is the definition of expectation, and we get the second line by Lemma 4.4.9. The third line is again by the definition of expectation, and the last line is follows from $X_s$ being a binomial random variable. We have now shown the expected number of path-related ancestor nodes. The case for the descendants is symmetric, and so in total we get the number of path-related nodes is

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

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

The last part of the lemma to show is the related nodes in the fringe problems. For all pivots $u, u \in R_{(\rho_{\max} - 1)D_r}(G, P)$, and from above we have $|R_{(\rho_{\max} - 1)D_r}(G, P)| \leq |R_{\rho_{\max}D_r}(G, v_0)| + |R_{\rho_{\max}D_r}(G, v_\ell)| \leq 2nk^{-r}$. By Lemma 4.4.4, in expectation, each node adds $1/(4\lambda k \log n)$ nodes to its fringe problem. Therefore the following holds,

$$\sum_{i=|s|+1}^{2|s|} E[|R_{\rho_{\text{min}}D_r}(G'[V_{u^{\text{Fr}inge}}, P_i)|]$$

$$\leq |R_{(\rho_{\max} - 1)D_r}(G, P)| \cdot \frac{1}{4\lambda k \log n} \leq \frac{2}{4\lambda k^{r+1} \log n/n} \leq \frac{1}{p_r}.$$

Now we will show the helper lemma for the previous lemma. Fineman [37] shows a similar lemma, which is that for one ancestor pivot the expected number of ancestors decreases by half in the next level of recursion. JLS [41] extends the lemma to
handle multiple pivots that can be ancestors or descendants. The lemma gets more complicated for our algorithm because each pivot searches for a different distance, however we are still able to get the claim.

**Lemma 4.4.9.** Consider the path \( P = \langle v_0, v_1, ..., v_\ell \rangle \), where \( \ell \leq D_r \), and the set of \( \rho_{minD_r} \)-distance ancestors \( R_{\rho_{minD_r}}(G, P) \) 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 \( R_{\rho_{minD_r}}(G, P) \). Let \( P_i \) be the path defined in Lemma 4.4.8. If the chosen interval \(|I| \geq 4t\), then

\[
\sum_{i=0}^{\left|S'\right|} E\left[ |R_{\rho_{minD_r}}^{-}(G'[V_a], P_i)| \right] \leq \frac{3}{2(t+1)} \left| R_{\rho_{minD_r}}^{-}(G', P) \right|.
\]

**Proof.** We will begin by defining a relationship for two nodes \( u, v \). We say \( u \) knocks out \( v \) to mean that if \( u \) is a pivot and \( w \) is a path-related node at level \( r \), then \( w \) is not path-related in a path-related subproblem at level \( r + 1 \) meaning that \( w \notin R_{\rho_{minD_r}}^{-}(G'[V_a], P_i) \) for all \( i \in [1, \left|S'\right|] \). Note that if \( u \) does not knock out \( v \) then it is possible that \( v \) is not path-related in level \( r + 1 \) because another pivot may knock out \( v \).

Recall that each pivot chooses a scalar for the interval that it searches from. We will show that for most scalars, either \( u \) knocks out \( v \) or \( v \) knocks out \( u \). Based on this claim, we will argue that for any fixed set \( S' \in R_{\rho_{minD_r}}^{-}(G', P) \), with \( |S'| = t + 1 \), if \( t \) pivots are selected from \( S' \), the remaining node will be a \( \rho_{minD_r} \)-ancestor in next level of recursion with probability \( \frac{3}{2(t+1)} \). Finally, we will show that choosing \( t \) pivots from \( R_{\rho_{minD_r}}^{-}(G', P) \) then the claim from the lemma statement holds.

We will first show that for two nodes \( u \) and \( v \), and most scalars, either \( u \) knocks out \( v \) or \( v \) knocks out \( u \). For nodes \( u, v, \) and interval scalar \( \sigma \in I \), let \( \chi(u, v, \sigma) \) be an indicator that when \( \sigma \) is \( u \)'s interval scalar, \( u \) does not knock out \( v \). We will argue that there is at most one scalar \( \sigma \in I \) where both \( \chi(u, v, \sigma) = 1 \) and \( \chi(v, u, \sigma) = 1 \). Assume
that \( \text{dist}_{G'}(u, v) \leq \text{dist}_{G'}(v, u) \), otherwise a symmetric argument can be applied. Let 
\( h = \text{dist}_{G'}(u, v) \) be the distance from \( u \) to \( v \). In the algorithm, each pivot chooses 
a interval scalar \( \sigma \), and a second scalar from within the interval, which is chosen so 
that it minimizes the number of nodes in the fringe problem/ Let the interval defined 
by \( \sigma \) be \( [\sigma_{\min}, \sigma_{\max}] \). The second scalar \( \rho \) is \( \rho \in [\sigma_{\min}, \sigma_{\max}] \). Next we will show 
when \( \sigma_{\min}D_r > h \) and \( \sigma_{\max}D_r \leq h \), at least one of \( \chi(u, v, \sigma) \) and \( \chi(v, u, \sigma) \) equals 
0. This implies that only the interval \( [\sigma_{\min}, \sigma_{\max}] \) can have both \( \chi(u, v, \sigma) = 1 \) and 
\( \chi(v, u, \sigma) = 1 \).

The first case is \( h < \sigma_{\min}D_r \). Fineman [37] has a similar proof that either \( u \) knocks 
out \( v \) or \( v \) knocks out \( u \). If \( v \) is a pivot, then \( u \) gets a \( v^{Anc} \) label because 
\( u \preceq \rho_uD_r v \), where \( \rho_u \) is the scalar for \( u \) and \( \rho_u \geq \sigma_{\min} \). In the proof of Lemma 4.4.8, we showed 
that none of the subpath \( P_i \) in the core problem gets a \( v^{Anc} \) label. Therefore, \( u \) is not 
path-related in the next level of recursion, and so \( v \) knocks out \( u \).

The second case is \( h \geq \sigma_{\max}D_r \). Let \( c_u = \text{Core}(P, u) \) and \( c_v = \text{Core}(P, v) \), as 
defined in the proof of Lemma 4.4.8. Since the search distance is at least \((\rho_{\min} + 1)D_r \) 
for each \( \rho_{\min}D_r \)-ancestor pivot, both \( c_u \) and \( c_v \) are at most \( \ell \). Let \( c'_u = \min\{j \mid \) 
\( u \preceq \rho_{\min}D_r v_j \} \) and \( c'_v = \min\{j \mid v \preceq \rho_{\min}D_r v_j \} \) be indices. Since \( u, v \in R_{\rho_{\min}D_r}(G', P), \) 
c\( c'_u \) and \( c'_v \) are always valid. If \( u \) is not pivot, \( u \) can only be a \( \rho_{\min}D_r \)-ancestor for a 
subpath \( P_i \) where \( \text{tail}(P_i) \geq c'_u \), since the shortest paths never decrease during the 
exection of the algorithm. If \( u \) is not a \( \rho_{\min}D_r \)-ancestor of any subpath \( P' \), then it 
will not be in any subsequent levels of recursion, i.e. \( u \notin R_{\rho_{\min}D_r}(G'[V'], P') \). Since 
\( \rho_u \geq \rho_{\min} + 1 \), and by definition of \( c'_u \), \( u \preceq_{(\rho_u-1)D_r} v_{c'_u} \), it follows that \( c_u \leq c'_u \) 
and \( c_v \leq c'_v \). First assume that \( c_u \leq c'_v \). If \( u \) is a pivot with subpath \( P_{u,i} \), then 
we have already shown in Lemma 4.4.8 that all nodes on paths \( P_i \), where \( i \geq i_u \), 
get the \( u^{Des} \) label. The node \( v \) however never gets a descendant label from \( u \) since 
\( h \geq \sigma_{\max}D_r > \rho_uD_r \). This implies that \( v \) is not be a \( \rho_{\min}D_r \)-ancestor for all paths \( P_j \),
where \( j \geq i \). We also know that \( v \) is not a \( \rho_{\min}D_r \)-ancestor for all paths \( P_j \) where \( j < i \) because \( \text{tail}(P_j) < c_u \leq c'_v \). Lastly, \( v \) is not a \( \rho_{\min}D_r \)-ancestor for subpath \( P_j \) if \( \text{tail}(P_j) < c'_v \). Therefore, if \( c_u \leq c'_v \), then \( \chi(u, v, \sigma) = 0 \), and the argument is symmetric for \( c_v \leq c'_u \). For both \( \chi(u, v, \sigma) = 1 \) and \( \chi(v, u, \sigma) = 1 \) to hold at the same time for \( h \geq \sigma_{\max}D_r \), it has to be the case that \( c_u > c'_v \) and \( c_v > c'_u \). This is impossible however since \( c_u \leq c'_u \) and \( c_v \leq c'_v \). Therefore, when \( h > \sigma_{\max}D_r \), one of \( \chi(u, v, \sigma) \) and \( \chi(v, u, \sigma) \) must be 0. Now we have shown the first claim which is that either \( u \) knocks out \( v \) or \( v \) knocks out \( u \).

Next we turn to our second claim which is any \( t + 1 \)-sized ancestors set \( S' \in R^-_{\rho_{\min}D_r}(G', P) \), if \( t \) pivots are chosen uniformly at random from \( S' \), the remaining node \( u_i \) will be path-relevant in the next level of recursion with probability at most \( \frac{3}{2(t+1)} \). Let \( R^-_{\rho_{\min}D_r}(G', P) \) be denoted as \( R^- \) for convenience. Let \( E_{\text{left}} \) be the event that \( u_i \) is path-relevant in the next level. If \( u_j \)'s scalar is \( \sigma_{u_j} \), where \( j \neq i \), then \( u_i \) is path-relevant in the next level if and only if all of the pivots \( u_j \) do not knock out \( u_i \) for \( j \neq i \), i.e., \( \chi(u_j, u_i, \sigma_{u_j}) = 1 \) holds for all \( j \neq i \). Therefore the probability of \( E_{\text{left}} \) is

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

Let \( A_{ji} = \{|\sigma | \chi(u_j, u_i, \sigma) = 1, \sigma \in I\}| \) be the number of interval scalars where \( u_j \) does not knock out \( u_i \). The probability of \( E_{\text{left}} \) can be rewritten as

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

For each \( u_j \) and \( u_i \), there is at most one \( \sigma \) where both \( \chi(u_j, u_i, \sigma) = 1 \) and \( \chi(u_i, u_j, \sigma) = 1 \), and so \( A_{ij} + A_{ji} \leq |I| + 1 \). The sum \( \sum_{i=1}^{t} \prod_{j \neq i} A_{ji} \) is maximized
when one of $A_{ij}$ and $A_{ji}$ equals $|I| + 1$, and therefore,
\[
\Pr [E_{\text{left}}] \leq \frac{(|I| + 1)^t}{(t + 1)|I|^t} \leq \frac{1}{t + 1} \cdot (1 + \frac{1}{|I|})^{|I|/4} \leq \frac{3}{2(t + 1)}.
\]

The last part of the proof is to show the expected number of path-related ancestors in the next level of recursion when $t$ pivots are chosen from $R^-$. Let $\mathbb{I}(S, u)$ be an indicator that a node $u$ is path-relevant in the next level of recursion when $S$ is the set of pivots. For any fixed set $S'$ we have,
\[
\sum_{S \subset S', |S| = t} \mathbb{I}(S, S' \setminus S) = (t + 1) \Pr [E_{\text{left}}] \leq 3/2.
\]

Now we take the sum over all possible sets $S' \in R^-$,
\[
\sum_{S' \subset R^-, |S'| = t + 1} \mathbb{I}(S, S' \setminus S) \leq \frac{3}{2} \left( \frac{|R^-|}{t + 1} \right).
\]

The above formula first chooses $S'$ then chooses a vertex $u \in S'$. This formula can be rewritten as first choosing $S$ and then choosing a vertex $u \notin S$, which is,
\[
\sum_{S \subset R^-, |S| = t} \mathbb{I}(S, S' \setminus S) \leq \frac{3}{2} \left( \frac{|R^-|}{t + 1} \right).
\]

If $u \in S$, then $\mathbb{I}(S, u) = 0$ and so,
\[
\sum_{S \subset R^-, |S| = t} \mathbb{I}(S, u) \leq \sum_{S \subset R^-, |S| = t, u \in R^-, S \neq S'} \mathbb{I}(S, S' \setminus S) \leq \frac{3}{2} \left( \frac{|R^-|}{t + 1} \right).
\]

In all, if the algorithm chooses $t$ ancestor pivots, then the expected number of ancestors in the next level is
\[
\sum_{i=1}^{|S'|+1} E_{|S'|=t}[|R_{\rho_{\text{min}}^+}(G'[V_{a_i}], P_i)|] = E_{|S'|=t} \left[ \sum_{u \in R^-} \mathbb{I}(S, u) \right]
\]
\[
= \sum_{S \subset R^-, |S| = t, u \in R^-} \mathbb{I}(S, u) \Pr [S] \leq \frac{3}{2} \left( \frac{|R^-|}{t + 1} \right) / \left( \frac{|R^-|}{t} \right)
\]
\[
\leq \frac{3}{2(t + 1)} |R_{\rho_{\text{min}}^+}(G', P)|.
\]
In the next lemma, we bound the total number of subproblems in each level of recursion. The main idea of the proof is to count the number of path-related nodes and the number of subproblems at each level of recursion. By bounding the number of path-related nodes, we know roughly the number of pivots which determines the maximum number of subproblems in the next level of recursion.

**Lemma 4.4.10.** Consider a full execution of HopsetG and the path-relevant subproblems tree. Let $Z_r$ be the number of subproblems in the $r$th level of recursion. Then, for all $r \geq 0$ the following holds,

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

**Proof.** We will show

$$
E[Z_r] \leq 15\lambda^r k^{c+r/2+1/2} \log n \tag{4.1}
$$

Assuming Equation 4.1 holds, then, \( \Pr \left[ Z_r \geq 30\lambda^r k^{c+r/2+1/2} \log^2 n \right] \leq \frac{1}{2\log n} \) by Markov’s inequality. Taking a union bound over all levels of recursion and if $\lambda \geq 8$ then,

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

The rest of the proof will show Equation 4.1. Let $Y_r$ be the number of path related vertices in the $r$th level of recursion. We will show \( E[Y_r] \leq 4\lambda^r nk^{c-r/2} \), and the expectation of $Z_r$ (Equation 4.1) by induction on level of recursion. The base case is $r = 0$ holds trivially because there are at most $n$ path related nodes, and one subproblem.

For inductive step we will start by constructing the logical subproblems. Let $Z'_r$ be the number of logical subproblems at level $r$, and let $Y'_r$ be the number of path-related nodes in the logical subproblems.
A subproblem is divided if the subpath length is greater than \( \ell/(\lambda^r k^{c+r/2}) \). This split creates at most \( \lambda^r k^{c+r/2} \) subproblems, and so \( Z'_r = Z_r + \lambda^r k^{c+r/2} \). By Lemma 4.4.3, the number of related nodes in each subproblem at level \( r \) is at most \( 2nk^{-r} \), with probability \( 1 - n^{0.7\lambda+3} \). Therefore the number of path-related nodes in the logical subproblems is

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

Next we will find \( Y_{r+1} \). By Lemma 4.4.8, the expected number of path-related nodes in all subproblems at level \( r \) is \( 4/p_r \) with probability \( 1 - n^{-0.7\lambda+4} \), where \( p_r \) is the probability a node is a pivot at level \( r \). Let \( s \) be a subproblem at level \( r \). At level \( r + 1 \), \( s \) is split among child subproblems. Define \( Y_s \) to be the number of path-related nodes in child subproblems of \( s \) at level \( r + 1 \). The expectation of \( Y_{r+1} \) is the following for \( \lambda \geq 4 \),

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

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

$$\leq 64\lambda^{r-1} nk^{c-r+1/2} \leq 4\lambda^{r+1} nk^{c-r+1/2}.$$

Now we can bound \( Z_{r+1} \). If there are \( t \) pivots then there are at most \( 2t + 1 \) subproblems in the next level of recursion. We divide the subproblems into \( 2t \) and 1 subproblem in order to count \( Z_{r+1} \). The 1 subproblems adds the extra \( Z'_r \) term, while
the 2t are counted by $Y_r'$. If $k \geq 2$, then we have,

$$E[Z_{r+1}] = \sum E[Z_r | Y_r'] \cdot \Pr[Y_r'] = p_r \cdot \sum 2Y_r' \Pr[Y_r'] + E[Z_r']$$

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

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

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

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

Now we can show that a long path gets shortcutted based on the path-relevant subproblems tree.

**Lemma 4.4.11.** Consider an execution of \textsc{Hopset}(G) on a graph $G$ containing a path $P = \langle v_0, \ldots, v_\ell \rangle$ where $|P| > n^{1/2}$. Let $Z_r$ be the number of path-relevant subproblems in the path-relevant subproblems tree at level $r$. Then there is a $v_1$ to $v_\ell$ path in $G' = (V, E \cup E')$ consisting of at most $3 \sum_{r \leq \log n - L} Z_r$ edges, where $E'$ is the hopset produced by \textsc{Hopset}(G).

**Proof.** We will start by considering the first $L$ levels of recursion. If any pivots $u$ are bridges at in the first $L$ levels of recursion, then $u$ was a shortcutter in \textsc{Hopset}(G), which was before any calls to \textsc{HSRecurse}(G', D, r = 0). For $u$ to be a bridge, $u \in R^+(G', \rho_{\max} D_r, P) \cap R^-(G', \rho_{\max} D_r, P)$, for $r \leq L$. When $u$ was a shortcutter, the search distance for $u$ is $2^{j+1}$ for $j \in [\log n/2, \log n]$ which means $u$ reaches both ends of the path. Therefore the new path is two hops, $v_0$ to $u$, and $u$ to $v_\ell$.

Otherwise there are no bridges pivots in the first $L$ levels of recursion. Consider a path-relevant subproblem and the subpath therein at level $r \geq L$ that has a bridge pivot $u$. At level $r - L$, the subpath is contained in a subproblem which contains a larger part of the subpath, $P' = \langle v_1, \ldots, v_j \rangle$. Since $u$ is a bridge at level $r$, $u \in$
\[ R^+(G', \rho_{\max} D_r, P) \cap R^-(G', \rho_{\max} D_r, P), \] and at level \( r - L \), \( u \) was a shortcutter. At level \( r - L \), the distance from \( v_i \) to \( u \) and \( u \) to \( v_j \) is at most \( \rho_{\max} D_r + |P'| \). Lemma 4.4.8 showed that \( |P'| \leq D_r \), which implies that when \( u \) is a shortcutter, it reaches \( v_i \) and \( v_j \). Therefore at level \( r - L \), the path is shortcutted to two hops, \( v_i \) to \( u \) and \( u \) to \( v_j \).

At level \( \log_k n \), all the nodes are pivots, and therefore each subpath must have a bridge pivot, and is shortcutted to two hops. In total there are \( \sum_{r \leq \log_k n - L} Z_r \) hopset edges that shortcut the subpaths in each path-relevant subproblem, and there are \( \sum_{r \leq \log_k n - L} Z_r \) edges that go between subproblems.

Now we are ready to show the hopbound for hopset produced by our algorithm.

**Lemma 4.4.12.** For a graph \( G \), consider the hopset produced by an execution of \( \text{Hopset}(G) \) with parameters \( k \), \( \lambda \), and \( L \). With probability \( 1 - n^{-\lambda + 2} \), the hopset has hopbound \( n^{1/2 + O(1/\log k) k^{c+(1-L)/2}} \log^2 n \).

**Proof.** Let \( P = v_0, v_1, ... v_\ell \) be any shortest path in \( G \) with \( |P| \geq n^{1/2} \). By Lemma 4.4.11, there is a path from \( v_0 \) to \( v_\ell \) 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 subproblems tree at level \( r \). The algorithm is repeated \( \lambda \log n \) times, which combined with Lemma 4.4.10 implies that there is a path-relevant subproblems tree with \( \bigcap_{r \leq \log_k n - L} Z_r \leq 32 \lambda^r k^{c+\frac{r+1}{2}} \log^2 n \) with probability \( 1 - n^{-\lambda} \). The final hopbound is

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

with probability \( 1 - n^{-\lambda + 2} \), which is achieved by taking a union bound over all possible shortest paths. \( \square \)
4.4.3 Approximation

In the last section we showed that the hopset achieves the desired hopbound by showing that a long shortest path is shortcutted to a new path with fewer edges on it. In this section we will show that the new path is a good approximation of the shortest path. Recall that for a path $P$, $w(P)$ is the sum of all the edge weights on $P$.

**Lemma 4.4.13.** Consider the hopset $E'$ produced by an execution of Hopset$(G)$ on a graph $G$ containing a path $P = (v_0, \ldots, v_\ell)$ where $|P| > n^{1/2}$. In $G' = (V, E \cup E')$, there exists a path $P'$ from $v_1$ to $v_\ell$ such that $|P'| \leq n^{1/2+O(1/\log k)}k^{c+(1-L)/2}\log^2 n$ with probability $1 - n^{-\lambda+2}$ and $w(P') \leq 4096\lambda^2-Lk^{(5-L)/2}\log^5 n \cdot w(P)$.

**Proof.** By Lemmas 4.4.11 and 4.4.12 there is a path $P'$ from $v_0$ to $v_\ell$ with $|P'| \leq n^{1/2+O(1/\log k)}k^{c+(1-L)/2}\log^2 n$ with probability $1 - n^{-\lambda+2}$. We will show this path $P'$ achieves the approximation.

Recall that in the path-relevant subproblems tree for $P$ a subproblem has no child subproblems if there is a bridge pivot. If there is a bridge pivot $u$ such that $\ell(u) \leq L$ then subproblems tree stops at level-0. Recall that the search distance at level-0 is $D \in [\ell k^{-c}, 2\ell k^{-c})$. The bridge searches forwards and backwards up to the search distance times the scaling factor which is at most $32\lambda^2k^2\log^2 n$ in each direction. The total error is at most $D \cdot 2 \cdot 32\lambda^2k^2\log^2 n \leq 128\lambda^2k^{2-c}\log^2(n)\ell$.

If there is no bridge pivot in the first $L$ levels then consider a bridge pivot $u$ at level $r + L$ for $r \geq L$. If $u$ is a pivot at level $r + L$ then $u$ was a shortcutter at level $r$, and the path-relevant subproblem will end at level $r$. Similarly to the first case, the error for this subproblem is $2 \cdot 32\lambda^2k^2\log^2 n \cdot D_r$. By summing over all path-relevant
subproblems, \( Z_r \) the total error is

\[
\sum_{r=\log_2 n-L}^{n-L} Z_r \cdot 64\lambda^2 k^2 \log^2 n \cdot D_r \leq 4096\lambda^2 L k^{(5-L)/2} \log^5(n) \ell.
\]

By setting \( c \) such that \( \lambda c = \frac{L k^{(L-1)/2}}{32 \log^4 n} \), the error from the first case equals the second which gives the total error on the path.

Now that we have shown the approximation we can prove Theorem 4.4.1.

**Proof of Theorem 4.4.1.** Set \( L = 15 - 2 \log k \epsilon \), and \( k = \Omega(\log n) \) to get the desired error \( \epsilon \ell \). By Theorem 4.4.2 and the stated setting of \( L \), the size of the hopset is \( O(nk^{16} \log^4 n / \epsilon^2) \) and the runtime is \( O(mk^{16} \log^4 / \epsilon^2) \). Finally, by Lemma 4.4.12 and the stated settings of \( c \) and \( L \), the hopbound is \( n^{1/2+O(1/\log k)} / \log n \) with probability \( 1 - n^{-\lambda+2} \).

4.5 Hopsets for Weighted Graphs

In this section we extend the algorithm from the previous section to work for weighted graphs. The algorithm is almost the same so most of the analysis still holds. We will show that for a weighted graph \( G \) with maximum edge weight \( W \), the algorithm outputs a \( (n^{1/2+o(1)}, \epsilon) \)-hopset of size \( O(nk^{16} \log^3 n \log(nW) / \epsilon^2) \) and runs in \( O(mk^{16} \log^4 n \log(nW) / \epsilon^2) \).

4.5.1 Algorithm for Weighted Graphs

The algorithm for weighted graphs, shown in Algorithm 5, is nearly the same as the algorithm for unweighted graphs. The only difference is Line 4 where the algorithm runs for \( j \in [-1, \log(nW)] \). The unweighted hopset needed to shortcut paths of length \( \sqrt{n} \) to \( n \). Here, the shortest paths can be up to \( nW \), since the maximum edge weight is \( W \), and the \(-1\) is needed to account for zero weighted edges. The rest of the algorithm
is the same, including the recursive subroutine \( HSRecurse(G, D, r) \) which is shown in Section 4.3.

**Algorithm 5** Hopset algorithm for weighted directed graphs. \( L, k \) and \( \lambda \) are parameters.

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

### 4.5.2 Weighted Graph Hopsets Analysis

In this section we will show the following theorem. Much of the analysis from the unweighted hopsets algorithm applies, but there are some differences which we will show.

**Theorem 4.5.1.** For an undirected graph \( G \) with non-negative edge weights, there exists a randomized algorithm that computes a \( (n^{1/2+o(1)}, \epsilon) \)-hopset of size \( O(nk^{16} \log^3 n \log(nW)/\epsilon^2) \) and runs in \( O(mk^{16} \log^4 n \log(nW)/\epsilon^2) \) with probability \( 1 - n^{-\lambda + 2} \).

First we will show the runtime for the algorithm and the size of the hopset. Most of the proof is the same as for the unweighted case with the exception of the cost of the searches, and the number distance the algorithm is run for. Since the graph is
weighted we cannot run BFS, and instead must run Dijkstra’s algorithm [31]. Finally, the algorithm is run for \( \log(nW) \) distances which adds this factor to the cost. The size of the hopset follows by the same argument.

**Lemma 4.5.2.** For a directed graph \( G \) with maximum edge weight \( W \), the algorithm \( \text{WeightedHopset}(G) \) outputs a hopset of size \( \tilde{O}(nk^{L+1} \log(nW)) \) with high probability, and runs in \( \tilde{O}(mk^{L+1} \log(nW)) \), where \( L \) and \( k \) are parameters.

**Proof.** Since the graph is weighted we use Dijkstra’s algorithm which costs \( O(m + n \log n) \) for a \( n \)-node \( m \)-edge graph [31]. Following the proof of Theorem 4.4.2, the number of edges explored in the searches is \( O(mk^{L+1} \log^2 n) \), and the number of nodes reached in the searches is \( O(nk^{L+1} \log^2 n) \). The number of distances is \( O(\log(nW)) \), and so the total runtime using Dijkstra’s algorithm for the searches is \( O(mk^{L+1} \log^4 n \log(nW)) \). By the same reasoning, the size of the hopset is \( O(nk^{L+1} \log^3 \log(nW)) \). \qed

Most of the analysis for the hopbound still holds from the unweighted section. The only change we have to make is splitting the logical path-relevant subproblems because they were based on number of edges but here we care about the distance. We split the path in to subpaths based on the weight of the subpaths. For a path-relevant subproblem \( (G', P, r) \) at level \( r \) contains a subpath \( P = \langle v_i, v_{i+1}, \ldots, v_j \rangle \), with \( w(P) \geq \frac{D}{\lambda^r k^r r^2} \), we partition \( P \) into \( q \) disjoint subpaths \( P_1, P_2, \ldots P_q \). We maximize the weight of each subpath \( P_i \) such that \( w(P_i) \leq D_r \), except for the last subpath which may be shorter. The number of new logical subproblems added is at most \( \lambda^r k^{c+r/2} \), which is the same as the unweighted case. The rest of the analysis for the hopbound is unchanged.

Finally, we will show the approximation for the weighted hopset. For paths \( P \) with \( w(P) \) the analysis is unchanged from the unweighted section. However for a
path \( P \) with \( w(P) = 0 \), we will show the new analysis. The \( \text{WeightedHopsets}(G) \) algorithm runs with \( j = -1 \) to account for paths with length at most \( 1/2 \). Recall that the lightest non-zero edge weight is 1, any path with \( w(P) < 1 \) must have \( w(P) = 0 \). Therefore, the approximate path weight zero satisfies any paths \( P \) with \( w(P) < 1 \).

By a proper setting of \( c \), and \( L \) set as stated in Section 4.4.3, \( \text{WeightedHopset}(G) \) produces a \( (n^{1/2+o(1)}, \epsilon) \)-hopset. By the previous analysis and Lemma 4.5.2, we have Theorem 4.5.1.

### 4.6 Parallel Algorithm for Hopsets

In this section we will show a parallel algorithm for hopsets for weighted graphs. The algorithm is an extension of the algorithm in the previous section with some key differences. Next we’ll describe the challenges of making the algorithm parallel and how we overcome the issues. Afterwards, we’ll show the algorithm, and then the analysis.

The key challenge for the algorithm is that it is too expensive to search long paths in the graph. The algorithm for weighted graphs uses Dijkstra’s algorithm to perform the searches, but Dijkstra’s is too expensive in parallel. Instead we use the rounding technique from Klein and Subramanian [44], and then BFS for the searches. Their rounding technique is as follows. Given a path \( P = \langle v_0, v_1, ... v_\ell \rangle \) with total weight \( w(P) \), and a small number \( \delta \), each edge \( e \) is rounded up to the nearest integer multiple of \( \delta w(P)/\ell \). Each edge will have at most \( \delta w(P)/\ell \) error, and the whole \( \ell \)-edge path has \( \delta w(P) \) error. Now consider the path but treating \( \delta w(P)/\ell \) as one unit. The new weight of the path is \( \tilde{w} = \frac{w(P) + \delta w(P)}{\delta w(P)/\ell} = (1 + \delta)\ell/\delta \). After performing this scaling and rounding, the algorithm can run BFS to depth \( O(\ell/\delta) \) in order to search the whole path \( P \).
Even with the scaling and rounding some of the paths may still be too long in terms of the number of edges. To overcome this problem we run HSRECURSE to a limited distance, add the hopset edges to the graph, and then repeat. Consider a path $P$ with $|P| = 4\beta$, and an execution of HSRECURSE($G, D, r$) that searches to $2\beta$, where $\beta$ is the hopbound. After running HSRECURSE($G, D, r$) and adding the hopset edges to the graph, there is guaranteed to be a path $P'$ that approximates $P$, such that $|P'| = 2\beta$, and $w(P') \leq (1 + \epsilon)(1 + \delta)\hat{w}(P) \leq (1 + \epsilon)w(P)$. By repeating this process, we can approximate long paths. For a path $P$ of any length, we can run HSRECURSE($G, D, r$) $\log(|P|/\beta)$ times and the hopbound will be $\beta$, and the approximation will be $(1 + \delta)\log(|P|/\beta)(1 + \epsilon)\log(|P|/\beta)$.

4.6.1 Algorithm

The parallel algorithm for hopsets is shown in Algorithm 6. Much of the algorithm is the same as the sequential weighted hopsets algorithms, though there are some key modifications. First, the edges are scaled and rounded by $\hat{w} = \delta 2^{i-1} / \beta$ in Lines 6 - 9. Second, like the previous sections we run the algorithm for distance guesses $2^i$ for $i \in [-2, \log(n^2W)]$ in some range. Here we again run the algorithm for distance guesses but after running the algorithm for all the distance guesses we add the hopset edges to the graph and repeat the algorithm. With each iteration of the algorithm we are shortcutting paths that contain more and more edges. The loop over the distances is defined in Line 5, and the repetitions loop is defined in Line 4. The range for the distances starts at $-2$ to account for zero weighted paths. In Line 16 if edge weight is less than 1, then set it to 0. We use the notation $w \cdot H$ to mean that that weight of each edge in $H$ is multiplied by $w$.

Next we’ll show the reasoning behind the rounding scheme. Let $\beta$ be the hopbound of the algorithm in Section 4.3, and $\delta$ be a small number to be set later. Consider a
path \( P = \langle v_0, v_1, \ldots, v_\ell \rangle \) with \( \ell \in [\beta, 2\beta] \), and \( w(P) \in [2^i, 2^{i+1}] \) for some an integer \( i \). Let \( \hat{w} = \delta 2^{i-1}/\beta \). If the edge weights are rounded as follows,

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

then each edge has at most \( \hat{w} \) error. The whole path has at most \( \ell \hat{w} \leq 2\beta \cdot \delta 2^{i-1}/\beta \leq 2^i \delta \). We can treat \( \hat{w} \) as one unit, and if \( k^c D = 4(1+\delta)\beta/\delta \), then the rounded length of \( P \), \( \hat{w}(P) \) is

\[
\hat{w}(\hat{P}) \in \left[ \left\lfloor \frac{w(\hat{P})}{\hat{w}} \right\rfloor \cdot \hat{w}, \left\lfloor \frac{(1+\delta)w(\hat{P})}{\hat{w}} \right\rfloor \cdot \hat{w} \right] \subseteq \left[ \frac{2\beta}{\delta}, \left\lceil \frac{4(1+\delta)\beta}{\delta} \right\rceil \right] \cdot \hat{w} \subseteq \left[ \frac{k^c D}{2+2\delta}, k^c D \right] \cdot \hat{w}.
\]

We can run BFS to depth \( 4(1+\delta)\beta/\delta \) treating \( \hat{w} \) as one unit to search the entire path. This allows the search depth to be independent of \( w(P) \). The algorithm rounds the edges and doesn’t multiple by \( \hat{w} \) to treat \( \hat{w} \) as one unit, and then multiplies each edge by \( \hat{w} \) when the edges are added to the hopset in Line 16.

### 4.6.2 Parallel Algorithm Analysis

In this section we will show the following theorem.

**Theorem 4.6.1.** For a \( n \)-node, \( m \)-edge directed graph \( G \), there exists a randomized parallel algorithm which computes a \( (n^{1/2+o(1)}, \epsilon) \)-hopset in \( O(m \log^{22} n \log(nW)/\epsilon^2 + n \log^{44} n \log^2(nW)/\epsilon^4) \) work and \( n^{1/2+o(1)}/\epsilon \) span with high probability. The size of the hopset is \( O(n \log^{22} n \log(nW)/\epsilon^2) \).

We will start by showing the hopbound and the hopset size, and then analyze the work and span of the algorithm.
Algorithm 6 Parallel hopset algorithm for weighted directed graphs. $L, \delta, k, \lambda$, and $c$ are parameters.

1: function PARALLELHOPSET($G = (V, E)$)
2: $H \leftarrow \emptyset$
3: $\beta \leftarrow 6\lambda^{\log_k n} n^{1/2}/\log n$
4: repeat $\lambda \log^2 n$ times
5: for each $i \in [-2, \log(n^2 W)]$
6: $\hat{w} = \delta \cdot 2^{i-1}/\beta$, $\hat{H}' \leftarrow \emptyset$
7: Construct a new graph $\hat{G} = (\hat{V} = V, \hat{E} = E)$
8: for each $e \in \hat{E}$
9: $\tilde{w}(e) = \begin{cases} +\infty & \text{if } w(e) \geq 2^{i+1} \\ \left\lfloor \frac{w(e)}{\hat{w}} \right\rfloor & \text{if } w(e) < 2^{i+1} \\ 1 & \text{if } w(e) = 0 \end{cases}$
10: for each $v \in \hat{V}$
11: for each $i' \in [0, \log_k n]$
12: With probability $(\lambda k^{i'+1} \log n)/n$, set $\ell(v)$ to $i'$, break if setting successfully.
13: if $\ell(v) \leq L$ then
14: for each $u \in R^+_{8(1+\delta)\beta/\delta}(G, v)$ add $(v, u)$ to $\hat{H}'$ with weight $\text{dist}_{\hat{G}}(v, u)$
15: for each $u \in R^-_{8(1+\delta)\beta/\delta}(G, v)$ add $(u, v)$ to $\hat{H}'$ with weight $\text{dist}_{\hat{G}}(u, v)$
16: $H \leftarrow H \cup (\hat{w} \cdot \hat{H}') \cup (\hat{w} \cdot \text{HSRECURSE}(\hat{G}, D = 4(1+\delta)\beta/(\delta k^c), r = 0))$
17: $E \leftarrow E \cup H$
18: return $H$
In this section we will show the hopbound and the hopset size for the parallel hopsets algorithm.

**Lemma 4.6.2.** Consider an execution of \textsc{ParallelHopset}(G) on a graph \( G = (V, E) \). Suppose for a path \( \hat{P} \) with \( |P| \leq 2\beta \), \textsc{HSRecurse} returns a \((1 + \epsilon')\) approximate path containing at most \( \beta \) edges with probability at least \( 1/2 \). Consider a path \( P \) with \( |P| = 2^j\beta \). If Lines 5-16 in Algorithm 6 are repeated \( j\lambda \log n \) times then there will be an approximate path \( P' \) with probability \( 1 - (2^j - 1)n^\lambda \) in \( G = (V, E \cup H) \) such that \( |P'| \leq \beta \) and \( w(P') = (1 + \delta)^j(1 + \epsilon')^j w(P) \).

**Proof.** Proof by induction on \( j \). For \( j = 1 \), the path \( P \) has \( |P| = 2\beta \). Lines 5-16 run for each possible distance guess. If they are repeated \( \lambda \log n \) times, then with probability \( 1 - \frac{1}{2^{\lambda \log n}} = 1 - n^{-\lambda} \), \textsc{HSRecurse}(G, D, 0) returns an approximate path for \( P \) with approximation \((1 + \epsilon)\). By the rounding technique described above, the error for the rounded path is \( \delta w(P) \), which gives a final approximation of \((1 + \delta)(1 + \epsilon)w(P)\).

Next assume the lemma holds for \( j \) and we will show the case for \( j + 1 \). For a path \( P \) with \( |P| = 2^{j+1}\beta \), divide \( P \) into two continuous subpaths \( P_1 \) and \( P_2 \) so that each subpath has at most \( 2^j\beta \) edges. By the inductive assumption after \( j\lambda \log n \) repetitions, with probability \( 1 - (2^j - 1)n^\lambda \) there are approximate paths \( P'_1 \) and \( P'_2 \) that are \((1 + \delta)^j(1 + \epsilon')^j \) approximations for \( P_1 \) and \( P_2 \), respectively, and \( |P'_1| \leq \beta \), and \( |P'_2| \leq \beta \). Now consider the subpath \( \langle P'_1, P'_2 \rangle \). After another \( \lambda \log n \) repetitions, there will be an approximate path \( P' \), which approximates \( P \), with \( |P'| \leq \beta \) with approximation \((1 + \delta)^{j+1}(1 + \epsilon)^{j+1} \). By taking a union bound over \( P'_1, P'_2 \) and \( P' \), the probability is \( 1 - (2^{j+1} - 1)n^{-\lambda} \).

Next we will give settings of the parameters which will give us a corollary on the hopbound. From Section 4.4.3, the recursive subroutine \textsc{HSRecurse}(G, D, 0)
returns a \((\beta = 6\lambda \log n n^{1/2}/\log n, \epsilon')\)-hopset with \(c\) set such that \(k^c = \frac{\lambda k(L-1/2)}{32 \log^3 n}\) and \(L = 15 - 2 \log_k \epsilon'\). By Lemma 4.6.2 any path \(P\) is shortcutted to \(\beta\) edges. The maximum weight of the path may increase as the successive approximate paths can be longer, but the weight of the path can be at most \((1+\epsilon)^{\log n} nW \leq n^2 W\). The algorithm runs for distances up to \(n^2 W\), which covers all paths. By setting \(\delta = \epsilon/(8 \log n)\) and \(\epsilon' = \epsilon/(8 \log n)\), we get a \((\beta, \epsilon)\)-hopset. By setting \(k = \Omega(\log n)\), then we can set \(L = 17 - 2 \log_k \epsilon\).

Finally we can show the hopset size. By Theorem 4.4.2, the size of the hopset is \(O(nk^{L+1} \log^2 n)\) for each run of \(\text{HSRecurse}(G, D, r)\). The number of runs is the number of iterations defined by the loops in Lines 4 and 5 which is \(\lambda \log^2 n\) and \(\log(n^2 W)\). The total hopset size is \(O(nk^{18} \log^4 n \log(nW)/\epsilon^2)\).

**Corollary 4.6.3.** Let \(G\) be a weighted directed graph, and consider a complete execution of \(\text{ParallelHopset}(G)\) with parameters \(L, k, \delta\) and \(c\) set as stated above. The algorithm 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 \(1 - n^{-\lambda+3}\).

4.6.4 WORK AND SPAN

To analyze the work and span of the algorithm, we will begin by noting that the cost of the searches dominates both the work and the span. For the work, the main difference from the previous analyses is that in the parallel algorithm edges are added to the graph throughout the algorithm. We will show the total work with edges added to the graph. The details for the parallel implementation are the same as Fineman [37] and JLS [41]. The only addition is that we must update the graph and add the hopset edges to the graph. This can be done by using parallel mergesort [29].

As we showed in the proof of Theorem 4.4.2, the total number of edges explored in \(\text{HSRecurse}(G, D, r)\) is \(O(mk^{L+1} \log^4 n)\). The size of the hopset is
\[ O(jnk^{18} \log^2 n \log(nW)/\epsilon^2) \] which means at most that many edges are added to the graph. In total, the work is

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

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

The span is dominated by the searches. The maximum search distance for a call to \( \text{HSRECURSE}(G, D, 0) \) is \( 4(1 + \delta)\beta/\delta \). In each recursive call the search distance halves, and so the total span for a single call to \( \text{HSRECURSE}(G, D, 0) \) is \( O(\beta/\delta) \). The algorithm is repeated \( O(\log^2 n) \) times which gives the total span \( O(\beta/\delta \cdot \log^2 n) = n^{1/2 + o(1)} \log^2 n/\epsilon \).

### 4.6.5 Parallel Approximate Shortest Paths

For a directed graph with non-negative edge weights we can solve \( (1 + \epsilon) \) approximate shortest paths by running the parallel hopset algorithm and then Klein and Subramanian’s \( k \)-limited shortest paths algorithm [44]. By Lemma 4.6.1 and running Klein and Subramanian’s algorithm, we have the following theorem.

**Theorem 4.6.4.** Given a \( n \)-node, \( m \)-edge directed graph with non-negative edge weights, there exists an algorithm which computes \( (1 + \epsilon) \)-approximate 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.
Chapter 5

Parallel Distance Limited Shortest Paths on Acyclic Graphs with \{0,-1\} Edge Weights

The result in this chapter will appear in Parallel Shortest Paths with Negative Edge Weights in Symposium on Parallelism in Algorithms and Architectures 2022.

5.1 Introduction

This chapter gives a parallel algorithm for distance-limited DAG shortest paths. Specifically, given a directed acyclic graph $G = (V,E)$ with edge weights in $\{0,-1\}$, source node $s \in V$, and distance limit $L$, the algorithm computes shortest-path distances to nodes with distance at least $-L$ from $s$. In particular, it outputs a distance value $d(v)$ where $d(v) = \text{dist}(s,v)$ if $d(v) \geq -L$, and $d(v) = -\infty$ otherwise. In addition, the algorithm also outputs for each node $v$ a negative edge $\text{parent}(v) = (x,y)$ that satisfies the following, if one exists: (1) $w(x,y) = -1$, (2) $\text{dist}(s,x) = \text{dist}(s,v) + 1$, and (3) there is a path from $y$ to $v$.

In the following section we will show a sequential algorithm for this problem, and later show how it can be made parallel. We assume throughout this chapter that all vertices in the graph are reachable from $s$, which is without loss of generality as reachability can be solved more efficiently than this problem [41].
Algorithm 7 RAM DAG shortest paths

Input: Graph $G = (V, E)$, whose edge weights $w : E \rightarrow \{0, -1\}$, and a source node $s \in V$.
Output: For all $v$, $d(v)$ is the shortest path distance from $s$ to $v$

parent$(v) = (x, y)$ is a negative ancestor edge with $d(x) = d(v) + 1$.

1: for each $v \in V$, $d(v) \leftarrow \infty$
2: Topologically sort $G$
3: for $u \in V$ in topologically sorted order
   4:     for each outgoing edge $(u, v)$
   5:        if $d(v) > d(u) + w(u, v)$ then
   6:           $d(v) \leftarrow d(u) + w(u, v)$, parent$(v) = u$

Definitions. We call an edge $(x, y)$ a negative ancestor edge of $v$ if the following hold:
(1) $w(x, y) < 0$, and (2) there is a directed path from $y$ to $v$. If $(x, y)$ is a negative ancestor of $v$, then we also call $x$ negative originator for $v$.

5.1.1 RAM Solution

In this section we will show a linear time solution to $\{0, -1\}$ DAG shortest paths. The solution is shown in Algorithm 7. First set each node’s distance to infinity. Then topologically sort the graph. Next loop over the vertices in topologically sorted order. For each node $u$, and each outgoing edge $(u, v)$, if $d(v) > d(u) + w(u, v)$ then set $d(v)$ to be $d(u) + w(u, v)$, and set parent$(v)$ to $u$.

The cost of the algorithm is $O(m + n)$. Topological sort can be done in $O(m + n)$ time by using depth first search [31], and then each node and each edge is considered once.

This algorithm does not translate well to the parallel setting. While the work is low $O(m + n)$, the span is $O(n)$ because it loops over the nodes in topologically
sorted order. It seems hard to parallelize this algorithm because the nodes must be considered in topological order.

5.1.2 Relation to Longest Paths

Given a graph $G$ with $\{0, 1\}$ edge weights and a source node $s$, we want to find the longest paths from $s$ to each node in the graph. This problem is equivalent to the problem we are trying to solve in this section. For each edge $e \in E$, if $w(e) = 1$, set $w'(e) = -1$, and if $w(e) = 0$, set $w'(e) = 0$. Then solve DAG shortest paths with $\{0, -1\}$ edge weights.

5.2 Algorithm

5.2.1 Overview

At a high level, the algorithm is a peeling algorithm proceeding in rounds $0, 1, \ldots, L$. Round $i$ identifies the set of vertices with distance exactly $-i$ and removes them from the graph. The challenge is to identify the set of nodes to peel by an efficient parallel algorithm.

First, consider a natural inefficient algorithm: (1) Identify the set $S \subseteq V$ of negative vertices, i.e., those with incoming negative edges. (2) Run multisource reachability with sources $S$, thereby identifying all vertices having a negative ancestor. (3) All vertices not reached in the reachability step have distance $-i$; remove them, and all of their incident edges, from the graph. An indirect approach like this seems necessary because the weights are negative; the problem is equivalent to finding longest paths in DAGs with weights $\{0, 1\}$.
The main problem with this algorithm is that it is not efficient, as each execution of multisource reachability has $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span [41], and we cannot afford to multiply these by $L$.

Our algorithm follows the same peeling approach, but we do not recompute reachability on the entire graph in each round. Instead, for each vertex $v$ we maintain $\text{label}(v)$, which corresponds to a negative ancestor edge $(u, x)$ if at least one exists. While $u$ remains in the graph, $v$ cannot be peeled. We thus do not include $v$ in any reachability steps until $u$ disappears. Eventually, $u$ is peeled from the graph, rendering $\text{label}(v)$ invalid. Our “propagate” algorithm applies multisource reachability to the subgraph of such invalid vertices and finds new negative ancestors, restoring the invariant.

We now give an overview of the analysis. Imagine for the sake of argument that $v$ always finds a negative ancestor uniformly at random. Then it is fairly easy to see that $v$’s label changes $O(\log n)$ times. (Roughly a constant fraction of the negative ancestors need to be peeled before there is a constant probability that the sampled ancestor is one of the peeled ones.) Each vertex thus belongs to only $O(\log n)$ subgraphs on which reachability/propagation is performed, keeping the total work down to $\tilde{O}(m)$. Moreover, because $n^{1/2+o(1)}$ is concave, the worst-case for the span is that each of the $L$ calls to reachability operate on graphs with $\tilde{O}(n/L)$ vertices. We thus get span $L \cdot (n/L)^{1/2+o(1)} = \sqrt{Ln}^{1/2+o(1)}$.

Priorities. We do not know how to maintain a uniformly random negative ancestor efficiently, but we achieve roughly the effect. We assign each vertex an independently random priority chosen from a geometric distribution with a rounded tail. Specifically, for $1 \leq i < \lceil \log_2 n \rceil$, we set $\text{priority}(v) = i$ with probability $1/2^i$; and with the remaining probability $1/2^{\lceil \log_2 n \rceil}$, we set $\text{priority}(v) = \lceil \log_2 n \rceil$. Priorities never
change throughout the execution. For an edge, we use $\text{priority}(x, y)$ as a shorthand for $\text{priority}(x)$. To approximate uniformly random ancestors, our algorithm ensures that nodes are labeled by a negative ancestor of maximum priority.

5.2.2 Algorithm details

Our algorithm for computing $\{0, -1\}$ DAG shortest paths is shown in Algorithm 8. Our style of pseudocode here reflects higher-level directives, not parallel code. With the exception of the black-box reachability subroutine, it is fairly straightforward and uninteresting to parallelize each of the steps. We thus defer a slightly more detailed discussion to Section 5.3.3.

As noted in the overview, the algorithm proceeds in rounds, and these rounds are ordered sequentially. In each round $i$, we identify the frontier $F$ of nodes at distance $-i$ and logically peel them from the graph. Note that we do not actually remove the nodes from the graph\footnote{This is primarily to avoid having to discuss how to update the graph in parallel.}; instead, we mark these nodes as finalized. We call the nodes that are not finalized live. Initially, all nodes are live.

We initialize a distance $d(v) = -\infty$ for all nodes and update this value when the vertex is eventually peeled. Similarly, we maintain a negative ancestor edge $\text{parent}(v)$ that is only guaranteed to be a correct parent edge when the vertex is finalized.

The crux of the algorithm is to maintain a negative ancestor edge $\text{label}(v)$ for each vertex $v$. Initially, all labels are invalid, indicated by $\bot$. Labels are restored by running \textsc{Propagate}, which occurs first in Line 5. We shall prove that when \textsc{Propagate} returns, a node $v$ has $\text{label}(v) = \bot$ only if all of $v$’s negative originators have been finalized. The set of nodes with no label even after \textsc{Propagate} returns is denoted by $F$. We also maintain a particular inversion of these labels: namely, $\text{SentLabel}(u)$ is
Algorithm 8 Peeling Algorithm

Input: Graph \( G = (V, E) \), whose edge weights \( w : E \rightarrow \{0, -1\} \), and a source node \( s \in V \).
Output: For all \( v \), \( d(v) \) is the shortest path distance from \( s \) to \( v \)
\( \text{parent}(v) = (x, y) \) is a negative ancestor edge with \( d(x) = d(v) + 1 \).

1: \textbf{foreach} \( v \in V \) \hfill \text{▷ Initialization}
2: \hspace{1em} \( d(v) \leftarrow -\infty \)
3: \hspace{1em} \( \text{label}(v) \leftarrow \bot; \text{parent}(v) \leftarrow \bot; \text{SentLabel}(v) = \{\} \)
4: \hspace{1em} assign a (geometric) random \( \text{priority}(v) \in \{1, \ldots, \lceil \log_2 n \rceil\} \)
5: \textsc{Propagate}(G, V)
6: \hspace{1em} let \( F = \{u \in V \mid \text{label}(u) = \bot\} \)
7: \hspace{1em} \textbf{for} \( i = 0 \) to \( L \)
8: \hspace{2em} let \( R = \bigcup_{u \in F} \text{SentLabel}(u) \)
9: \hspace{2em} \textbf{foreach} \( v \in R \)
10: \hspace{3em} \( \text{label}(v) \leftarrow \bot \)
11: \hspace{2em} \textbf{foreach} \( u \in F \)
12: \hspace{3em} \( d(u) = -i \); mark the node as finalized
13: \hspace{2em} \textsc{Propagate}(G, R)
14: \hspace{2em} let \( F = \{u \in R \mid \text{label}(u) = \bot\} \)

the set of all vertices that have any edge \((u, \ast)\) leaving \( u \) as their label. This \text{SentLabel} facilitates invalidating labels when \( u \) is finalized/peeled from the graph.

In each round of the algorithm (Lines 7–14), the current frontier \( F \) of nodes to peel is processed as follows. First, we identify the set of nodes \( R \) that have a label including one of the peeled nodes, and we invalidate their labels. Then we set the distance to all nodes in \( F \) and finalize them, which logically peels them from the graph. Next, we call propagate on the nodes \( R \) (i.e., those with invalid label) to restore their labels. Finally, we let \( F \subseteq R \) be the next frontier of nodes, i.e., those with no label after calling propagate.
Propagate. The \textsc{Propagate}(G, V') function, see Algorithm 9, takes as input the graph $G = (V, E)$ and a subset $V' \subseteq V$ of vertices. As called, $V'$ always corresponds to the set of vertices with invalid label. The goal is to label the nodes in $V'$ with one of their highest-priority negative ancestor edges, or $\bot$ if none exists.

\textsc{Propagate} considers each of the priorities in turn, from high to low. Each iterations starts by initializing the vertices in $V'$ by inheriting any “nearby” labels. Specifically, we define the function \textsc{GetNearbyLabel}(G, V', i, v) to return an edge (or $\bot$) as follows:

- If $v$ has at least one incoming edge $(u, v) \in E$ with: (1) $u$ is live, (2) $w(u, v) = -1$, and (3) $\text{priority}(u) = i$, then choose any one such $u$ and return the edge $(u, v)$.

- If $v$ has at least one incoming edge $(u, v) \in E$ with: (1) $u$ is live, (2) $u \notin V'$, i.e., it already has a valid label, and (3) $\text{priority}(\text{label}(u)) = i$, then choose any one such $u$ and return the edge indicated by $\text{label}(u)$.

- If neither of the above two cases applies, return $\bot$

After initializing the nearby labels for nodes in $V'$, those nodes in $V'$ with labels are the set of sources $S$. We next run multisource reachability on the induced subgraph $G' = G[V']$ with sources $S$. When multisource reachability returns, for each node in $v \in V'$, $\pi(v)$ indicates a node $s \in S$ that can reach $v$, if one exists. If $\pi(v) \neq \bot$, then we simply update $v$ by inheriting $\pi(v)$’s label information.

Finally, each iteration ends by removing all newly labeled nodes from $V'$. In this way, $V'$ reflects the nodes that are still unlabeled.

After all iterations complete, \textsc{Propagate} updates the $\text{SentLabel}$ of all nodes as appropriate. In particular, $\text{SentLabel}(u)$ should be updated to also include all newly label nodes that have a label $(u, \ast)$. Note that this update is applied across the entire graph as appropriate, not just those nodes in $V'$.
Algorithm 9 Propagate Algorithm

1: function Propagate($G = (V, E), V' \subseteq V$)
2: for $i = \lceil \log_2 n \rceil$ downto 1
3: foreach $v \in V'$
4: $e \leftarrow \text{GetNearbyLabel}(G, V', i, v)$
5: $\text{label}(v) \leftarrow e$
6: if $e \neq \bot$ then $\text{parent}(v) \leftarrow e$
7: let $S = \{v \in V' \mid \text{label}(v) \neq \bot\}$
8: $G' \leftarrow G[V']$
9: run multisource reachability on $G'$ with sources $S$
10: for each $v \in V'$
11: if $\pi(v) \neq \bot$ then
12: $\text{label}(v) \leftarrow \text{label}(\pi(v))$
13: $\text{parent}(v) \leftarrow \text{label}(v)$
14: remove all nodes with $\text{label}(v) \neq \bot$ from $V'$
15: update $SentLabel$ sets to include new label assignments

5.3 Analysis

5.3.1 Correctness

This section proves the correctness of our peeling algorithm. We start by focusing on Propagate. For a live node $v$, we use $NA(v)$ to denote the set of live negative ancestors of $v$. That is

$$NA(v) = \{(x, y) \in E \mid x \text{ is live, } w(x, y) = -1, \text{ and } y \in \text{Anc}(v)\}.$$  

We use $\text{maxPri}(v) = \max_{(x, y) \in NA(v)} \text{priority}(x, y)$ to denote the maximum priority over all of $v$’s live negative ancestors. For completeness, we define $\text{maxPri}(v) = 0$ if $NA(v) = \emptyset$. Finally, we say that a live node $v$ is correctly labeled if one of the applies:

- $NA(v) = \emptyset$ and $\text{label}(v) = \bot$, or
• $\text{label}(v) \in \text{NA}(v)$ and $\text{priority}(\text{label}(v)) = \text{maxPri}(v)$.

Our first goal is to show that when \textsc{Propagate}, all live nodes are correctly labeled.

**Lemma 5.3.1.** Consider a call to \textsc{Propagate}(\(G = (V, E), V'\)), where \(V' \subseteq V\) is exactly the subset of live nodes in \(G\) that have label \(\perp\). If all live nodes in \(V \setminus V'\) are correctly labeled before the call, then all live nodes in \(V\) are correctly labeled after the call.

**Proof.** Because no labels are updated for any nodes outside of \(V'\), establishing the correct labeling of \(V'\) is sufficient. The proof is by induction over the iterations of the main loop in \textsc{Propagate}. The iterations are numbered in decreasing order, from \([\log_2 n]\) downto \(1\). Our inductive claim is that after iteration \(i\): (1) all nodes outside \(V'\) are correctly labeled, and (2) all nodes in \(V'\) have \(\text{maxPri}(v) < i\). The remainder of the proof focuses on proving the inductive step. Specifically, we show that all nodes with \(\text{maxPri}(v) = i\) become correctly labeled, and also that those nodes with \(\text{maxPri}(v) < i\) do not get a label incorrectly.

Consider iteration \(i\) and any live node \(v \in V'\) having \(\text{maxPri}(v) \leq i\). Consider also any ancestor \(z \in \text{Anc}(v) \cap V'\). We first claim that if \(z\) gains a label during \textsc{GetNearbyLabel}, then that label would be a valid label for \(v\). To prove the claim, we observe that the only labels considered during this iteration have priority \(i\) by construction. Moreover, by transitivity of reachability, \(\text{NA}(z) \subseteq \text{NA}(v)\) and hence \(\text{maxPri}(v) \geq \text{maxPri}(z)\). Therefore, if \(z\) gains a label, that label would be a correct label for \(v\). It follows that whenever \(\pi(v) \neq \perp\) after running multisource reachability on \(G[V']\), the label \(\text{label}(\pi(v))\) is always a correct label for \(v\).

We next claim that if \(\text{maxPri}(v) = i,\) for \(v \in V'\), then there exists at least one node \(z \in V'\) such that: (1) \(z\) becomes labeled during \textsc{GetNearbyLabel}, and (2) there is
a path from $z$ to $v$ in $G[V']$. If this claim holds, we have that whenever $\maxPri(v) = i$, $\pi(v) \neq \bot$. Coupled with the above claim, $v$ becomes correctly labeled.

To prove the claim, consider any node $v \in V'$ with $\maxPri(v) = i$. By definition, there exists at least one edge $(x, y) \in NA(v)$ with $\text{priority}(x, y) = i$. Consider any path $\Gamma$ from $y$ to $v$ in $G$. If the entire path is in $V'$, then $\Gamma$ is a path in $G[V']$. Moreover, $y$ has an incident negative edge with appropriate priority, so $z = y$ is indeed labeled during $\text{GETNEARBYLABEL}$. Otherwise, let $(r, z)$ be the latest edge on the path with $r \notin V'$ and $z \in V'$. Then the subpath of $\Gamma$ from $z$ to $v$ is a path in $G[V']$. Moreover, $r$ is outside $V'$ so it is correctly labeled by inductive assumption. We have $i = \text{priority}(x, y) \leq \maxPri(r) \leq \maxPri(v) = i$, so $r$ is already labeled with priority exactly $i$. Thus, $z$ is also assigned a label during $\text{GETNEARBYLABEL}$. 

We next turn to correctness of the higher-level algorithm. The following lemma essentially indicates that if finalizing nodes in order of distances, then all labels involving farther-away (more negative distance) nodes are still valid.

**Lemma 5.3.2.** Consider a DAG $G = (V, E)$ with edge weights from $\{0, -1\}$ and source vertex $s \in V$ that can reach all vertices.

Consider the following generic process. A node $v$ is initially correctly labeled with label$(v) = (x, y)$. Let $F \subset V$ be any subset of nodes nearer to the source than $x$, i.e., for all $f \in F$, dist$(s, f) >$ dist$(s, x)$. Suppose that the nodes in $F$ are finalized (or removed from the graph). Then $v$ is still correctly labeled.

**Proof.** We first observe that for all nodes $z \in \text{Des}(y)$, we have dist$(s, z) \leq$ dist$(s, x) + w(x, y) +$ dist$(y, z) <$ dist$(s, x)$, because distances are all nonpositive and finite. Thus, no descendants of $y$ are part of $F$, and in particular the path from $y$ to $v$ remains in the graph. Because no nodes higher priority nodes or paths are created, it follows that $v$ is still correctly labeled. 

116
The next lemma indicates unlabeled nodes can be finalized.

**Lemma 5.3.3.** Consider a DAG $G = (V, E)$ with edge weights from $\{0, -1\}$ and source vertex $s \in V$ that can reach all vertices. Suppose all the nodes, and only the nodes, with distance $> -i$ from the source have been finalized. For any live node $v$, if $NA(v) = \emptyset$ then $\text{dist}(s, v) = -i$.

*Proof.* By assumption, $\text{dist}(s, v) \leq -1$. Suppose for the sake of contraction that $\text{dist}(s, v) < -i$. Then there exists some shortest path $\Gamma$ from $s$ to $v$. Let $(x, y)$ be the last negative edge on the path. Then $\text{dist}(s, v) = \text{dist}(s, y) = \text{dist}(s, x) - 1$, or $-i > \text{dist}(s, v) = \text{dist}(s, x) - 1$, or $-i \geq \text{dist}(s, x)$. Therefore, neither $x$ nor $y$ has not been finalized, and hence $(x, y) \in NA(v)$. This contradicts the assumption that $NA(v) = \emptyset$. \qed

Finally, we prove that the output of the algorithm is correct.

**Theorem 5.3.4.** Consider a directed acyclic graph $G = (V, E)$ with edge weights from $\{0, -1\}$, let $s \in V$ be the source node, and suppose that all nodes in the graph are reachable from $s$. Then Algorithm 8 correctly solves the distance-limited DAG SSSP problem.

More precisely, on completion, for all $v \in V$ we have:

- $d(v) = \text{dist}(s, v)$ if $\text{dist}(s, v) \geq -L$, or $d(v) = -\infty$ otherwise.

- $\text{parent}(v) = (x, y) \in NA(v)$ and $\text{dist}(x) = \text{dist}(v) + 1$.

*Proof.* The proof is by induction over rounds of the Algorithm 8. The inductive claim is that after round $i$, all nodes and only nodes with distance at least $-i$ have been finalized. Moreover, their distance is marked correctly. In addition, we include in the claim that $F$ is exactly the set of live nodes that currently have no label.
Now the inductive step. Consider the start of iteration $i$. By Lemma 5.3.3, all nodes in $F$ have distance equal to $-i$. By Lemma 5.3.2, the labels are still correct for all nodes not in $R$. Thus, by Lemma 5.3.1, all live nodes are correctly labeled when Propagate returns. Finally, $F$ is restored to be the live nodes with no label.

Finally, we argue that the parent references are correct. To do so, we simply observe that the parent only changes when the label changes. Thus, $\text{parent}(v) = (x, y)$ corresponds to the final label that $v$ had before arriving at a label of $\bot$. Since $v$ is finalized in round $-\text{dist}(s, v)$, $v$ must have lost its label in the preceding round. In other words, $\text{dist}(s, x) = \text{dist}(s, v) + 1$.

5.3.2 Key Performance Claims

This section gives the core components of the performance analysis, specifically bounding the total work and span of all multisource reachability invocations. The other straightforward details of the parallel implementation are shown next in Section 5.3.3.

Our main goal is to show that for each vertex $v$, $v$’s label does not change too many times. We do so by first arguing $v$ does not have too many negative originators with priority equal to $\text{maxPri}(v)$.

Lemma 5.3.5. Consider the state of the graph at the start of a round $i$, i.e., when all nodes with distance $>-i$ have already been finalized. Let $v$ be any live vertex. With high probability, $v$ has at most $O(\log n)$ negative originators that have priority equal to $\text{maxPri}(v)$.

Proof. We shall consider a specific priority and argue that some, but not too many, negative originators of $v$ have at least that priority. Specifically, let $\beta$ be the total number of live negative originators of $v$. Consider a priority $x =
\[ \lfloor \log \beta - \log \ln(n) - \log(c) \rfloor, \] for constant \( c > 1 \). (The \( \Theta \) absorbs the rounding of \( x \) to an integer.) Observe that \( c2^x \ln n \leq \beta \leq c2^{x+1} \ln n \).

We first claim that, with failure probability at most \( 1/n^c \), \( v \) has at least one negative originator with priority \( \geq x \). For a particular negative originator \( u \), we have probability \( 1/2^{x-1} \) that \( \text{priority}(u) \geq x \). There are \( \beta \geq c2^x \ln(n) \) negative originators, so the probability that none of them has priority \( \geq x \) is at most \( (1 - 1/2^{x-1})^\beta \leq ((1 - 1/2^{x-1})2^x \ln n \leq (1/e)^c \ln n = 1/n^c \).

We next claim that not too many negative originators have priority \( \geq x \). Each negative originator has an independent priority, and the expected number of originators having at least this priority is \( \beta/2^{x-1} \leq 4c \ln n \). We can thus apply a Chernoff-Hoeffding bound to conclude that with probability at most \( 1/n^c \), no more than say \( 8c \ln n \) negative originators have priority \( \geq x \).

**Corollary 5.3.6.** Consider a particular node \( v \) across the execution of the algorithm, and assume that the algorithm operates correctly. Then \( v \)'s label changes at most \( O(\log^2 n) \) times across the entire execution, with high probability.

**Proof.** Lemma 5.3.5 states that with high probability, at the start of each round, \( v \) has at most \( O(\log n) \) negative originators with priority equal to \( \text{maxPri}(v) \). Because which nodes are peeled is deterministic, there are not many events to take a union bound over. In particular, the probability of even one failure for any vertex across the execution is at most \( n(L + 1)/n^{\Theta(c)} \).

For the remainder, assume no failures. Then Lemma 5.3.5 holds specifically each time \( \text{maxPri}(v) \) changes. Since there are only \( O(\log n) \) nodes with that priority, \( v \) can only get subsequent labels with the same priority \( O(\log n) \) times. Multiplying this \( O(\log n) \) by the number of different priorities completes the proof. \( \square \)
We are now ready to bound the total cost of the calls to multisource reachability, which dominates the costs of the algorithm.

**Lemma 5.3.7.** Consider the total across all invocations of multisource reachability across the execution of the algorithm. The total cost of these calls is $\tilde{O}(m)$ work and $\sqrt{L} \cdot n^{1/2 + o(1)}$, with high probability.

*Proof.* From Corollary 5.3.6, each node’s label changes at most $O(\log^2 n)$ times, with high probability. This also bounds the number of times each node is passed to PROPAGATE. During each call, the node and its incident edges may be built into $O(\log n)$ induced subgraphs, one for each priority. Adding up across all edges and calls, we get a total induced subgraph size of $O(n \log^3 n)$ vertices and $O(m \log^3 n)$ edges. Because the work of multisource reachability [41] is nearly linear, the total across all calls is also $\tilde{O}(m)$ no matter how the edges are divided across calls. Because the span is concave, the worst case (according to Jensen’s inequality) is that all calls are on graphs of size $O(n \log^3 n / L \log n)$ which yields a total span of $L \log n \cdot (n \log^3 n / L \log n)^{1/2 + o(1)} \leq \sqrt{L} n^{1/2 + o(1)}$. 

**Theorem 5.3.8.** For a DAG $G$ containing integer edge weights of $\{0, -1\}$ and a source node $s$, there is a parallel algorithm that outputs for each node $v \in V$, $d(v) = \text{dist}(s, v)$ if $\text{dist}(s, v) \geq -L$, and $-\infty$ otherwise. The algorithm runs in $\tilde{O}(m)$ work and $n^{1/2 + o(1)} L^{1/2}$ span.

*Proof.* Proof follows from Lemmas 5.3.3, 5.3.7, and 5.3.9, where the latter (in Section 5.3.3) includes of the remaining parallel steps. 

120
5.3.3 Parallel Implementation Details

This section discusses some of the more straightforward steps for making the algorithm parallel. Our main claim is the following lemma. Proof appears at the end of the section.

**Lemma 5.3.9.** Algorithm 8 (PeelingAlgorithm) runs in $\tilde{O}(m)$ work and $n^{1/2+o(1)}L^{1/2}$ span.

Before showing the proof, we will show a data structure that we will use.

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 $n \geq m$ [14]. The parallel set can also enumerate all elements in a size $n$ set in $O(n)$ work and $O(\log n)$ span [14].

Parallel implementation. Now that we have parallel sets we can show the parallel implementation of the peeling algorithm. We will first discuss parallel propagate and then the main loop of the peeling algorithm. The PROPAGATE($G, R$) function runs on a subgraph $G'$ which changes during each iteration of the algorithm. We will show how to update the graph in parallel and other steps that are non-trivial. A key point is that we can afford to look at each node in the subgraph $G'$, since as we showed in Corollary 5.3.6 that in each node is added to $O(\log^2 n)$ subgraphs across the whole algorithm. However, we cannot afford to look at the original graph $G$. Also, for a node $v$ in $G'$ we can afford $v$'s edges in $G$, and not just the edges in $G'$. This makes building the induced subgraph trivial once we are given the nodes.

In each iteration of propagate we must update the graph by removing any nodes that get a label. Since we have $O(\log n)$ priorities, we can afford to sort all $V'$ nodes
in each iteration. By sorting we can group the nodes together that are not removed from \( G' \) in previous iterations. The cost of the sort is \( \tilde{O}(V') \) work and \( \tilde{O}(1) \) span.

We implement the \( \text{SentLabel}(u) \) sets as parallel sets. To update the \( \text{SentLabel}(u) \) sets (Line 15 in Algorithm 9), first sort all the nodes \( v \in V' \) by their label, i.e. sort edges \( (u, v) \) by ordering on \( u \). This groups all the nodes together that will be added to \( \text{SentLabel}(u) \). To add these nodes to \( \text{SentLabel}(u) \), we merge these nodes with the current set \( \text{SentLabel}(u) \). In total, the cost is \( \tilde{O}(V') \) work and \( \tilde{O}(1) \) span to update \( \text{SentLabel}(u) \) for all the nodes in each round of \( \text{PROPAGATE}(G, V') \). It is fairly straightforward to implement the rest of the steps in parallel for \( \text{PROPAGATE}(G, V') \).

Before we return to the loop in the main algorithm, we sort the nodes by label, which groups together all the nodes that have no label, and thus are finalized. This has the cost of parallel sort for the number of nodes in \( \text{propagate} \). When we return to the main loop, we need to compute the next set of nodes to run \( \text{propagate} \) on. To implement this step, in parallel each finalized node \( u \) determines the number of nodes in \( \text{SentLabel}(u) \). We can then run parallel prefix sums to allocate space in an array for the nodes in \( R \). Then the nodes can be added to the new array in parallel, which is passed to the next call to \( \text{propagate} \). The work and span of these steps is dominated by the cost of sorting the number of elements in \( \text{propagate} \) at each round.

Proof of Lemma 5.3.9 The initialization steps can be performed in \( \tilde{O}(n) \) work and \( \tilde{O}(1) \) span. Previously we discussed how the \( \text{SentLabel} \) sets can be maintained in \( O(|V'|) \) work and \( \tilde{O}(1) \) span. By Lemma 5.3.7, the cost of all the runs of multisource reachability is \( \tilde{O}(m) \) and \( n^{1/2+o(1)}L^{1/2} \) span. Based on Corollary 5.3.6, the total number of nodes in all calls to \( \text{PROPAGATE}(G, V') \) is \( \tilde{O}(n) \). We also showed that we can update the nodes in \( R \), and perform the updates to the graph \( G' \) in total work \( \tilde{O}(n) \), and \( \tilde{O}(1) \) span.
Combining everything together, the algorithm runs in $\widetilde{O}(m)$ work and $n^{1/2+o(1)}L^{1/2}$ span.
Chapter 6

Asymmetric-RAM Algorithms

6.1 Introduction

In this chapter we turn our attention to the Asymmetric RAM model [15]. In this model, the cost of writing to memory is higher than the cost of reading memory, with the exception of a small symmetric cache. This model of computation reflects existing technology in which reading from memory has become significantly faster than writing to memory [4, 32, 43, 61]. Writes can be higher by an order of magnitude or more in energy and latency than reads [11, 13].

This chapter will study several graph problems in this model for bounded degree graphs. A bounded degree graph is a graph in which each vertex has degree $d$, where $d$ is a constant. In particular, we will show algorithms for breadth first search (BFS), depth first search (DFS), and strongly connected components (SCC), that use a sub-linear number of writes. The standard RAM algorithms for these problems write the solution for each node in the graph. In order to save writes, we build a $o(n)$-size data structure that can answer queries.

6.1.1 Asymmetric RAM Model

Blelloch et al. [15] introduced the $(M, \omega)$-ARAM model. In this model there is a symmetric memory of size $M$, and an unbounded asymmetric memory. In the symmetric memory reads and writes each have unit cost, whereas in the asymmetric memory
reads have unit costs but writes cost $\omega \gg 1$. The cost of an algorithm in this model is the number of reads in the larger memory plus $\omega$ times the number of writes to the larger memory.

6.1.2 Background and Motivation

Blelloch et al. [15] show algorithms for BFS, DFS and SCC with cost $\Theta(\omega n + m)$. The standard RAM algorithms for these three problems output a solution for each node in the graph which costs $\Omega(\omega n)$ in this model. In order to get an improvement, we cannot afford to write out the solution for each node and instead turn to querying. In particular, given a query node or nodes we would like to be able to query a $o(n)$-sized data structure to return the BFS distance, DFS start and finish times, or SCC id. The cost of our algorithm has a cost for preprocessing the graph to create a data structure, and a cost for the query. Our goal is to reduce the number of writes to $o(n)$ at the expense of a higher query cost.

6.1.3 Our Results

In Section 6.2 we consider the breadth first search problem. For a graph $G$ and source node $s$, we want to compute the breadth first search distance from $s$ to each node in the graph. The breadth first search distance is the shortest path distance where the weights of all edges in the graph are one. The classic RAM BFS algorithm costs $O(n\omega + m)$ for an $n$-node, $m$-edge graph in the $(M, \omega)$-ARAM model. Our result is an algorithm that preproccesses the graph with a cost of $O(n\omega / \log \omega)$, and has a query cost of $O(\sqrt{\omega})$ reads.

Next in Section 6.3, we study depth first search. When running DFS there is the discovery and finish time for each node. For different choices of edge orderings this can result in a different discovery and finish time. Therefore, we assume there is an
ordering over the vertices would like to run DFS with respect to this ordering. More specifically, we would like to preprocess the graph so that we can query the discovery and finish time of a vertex. Once the nodes are labeled, we would like to iterate over the nodes in depth first order. Our cost is $O(n / \log \omega)$ writes and $O(n \sqrt{\omega} / \log \omega)$ reads for preprocessing and $O(\sqrt{\omega})$ reads for a discovery or finish time query.

Using the write-efficient DFS, we compute strongly connected components in Section 6.4. In particular, two nodes $u$ and $v$ are strongly connected if there is a path from $u$ to $v$ and a path from $v$ to $u$. Here we preprocess the graph so that we can query whether any two nodes in the graph are strongly connected. The cost of preprocessing the graph is $O(n / \log \omega)$ writes and $O(n \sqrt{\omega} / \log \omega)$ reads, and the query cost is $O(\sqrt{\omega})$.

6.1.4 Main Techniques

Our key idea is to create implicit representations of BFS and DFS labelings. For now we will describe the idea for BFS, and we use a similar strategy for DFS. The algorithm cannot afford to explicitly write the BFS distance so it must do so implicitly. In particular, a set of nodes is carefully selected to be labeled with its BFS. We want a set of nodes that is not too big, and that guarantees a query can be performed quickly.

We select layers of the graph which are not too big, where a layer is all the nodes that are exactly $d$ away from the source for some distance $d$. If we are querying a node that is $d + c$ away from the source for some constant $c$, we can search backwards to find a labeled node. Since every node at distance $d$ is labeled, the shortest path is guaranteed to go through one of these labeled nodes. We carefully select layers of the graph that are not too big and not too far apart so that queries can be answered quickly while not writing too many nodes.
6.1.5 Related Work

Blelloch et al. [15] introduced the ARAM model and construct algorithms for many graph problems in addition to the ones mentioned above.

Blelloch et al. [12] construct data structures for connectivity and biconnectivity in undirected graphs. The data structures can be built in $O(m/\sqrt{\omega})$ writes and $O(m\sqrt{\omega})$ reads. The cost of a query is $O(\sqrt{\omega})$ reads for connectivity and $O(\omega)$ reads for biconnectivity. They also adapt their data structure to the Asymmetric Nested Parallel model [11]. The use an implicit decomposition to label the graph. For connectivity, the select a set of nodes to label that are cluster centers i.e. not too far from nodes in the cluster. By carefully selecting nodes to label, they can guarantee that a queried node does not have to search too far to find a labeled node to identify which component it is in.

Finally, there has been further work on studying write efficient algorithms in other computational models including the PRAM and the external memory model and the ideal cache model [11, 13, 17, 40].

6.2 Breadth-First Search

The shortest path distance problem for unweighted graphs is given a given a graph $G$, and source vertex $s$, return the shortest path distance from to the source to a queried vertex. The standard algorithm runs breadth first search and labels each vertex with its shortest path distance [31].

We introduce the idea of an implicit representation of a BFS labeling. A set of vertices are labeled with shortest path distance such that for the rest of the vertices, there is a vertex on the shortest path from the source that is labeled and within $k$ hops. The graph is preprocessed to add these labels, and then can be used to answer
queries. We consider a layer of the graph to be the set of nodes where the shortest path distance to each node is \( j \) for some integer \( j \). The main idea of the BFS labeling is to label layers that are multiples of \( k \) starting with zero with the shortest path distance to layer. Unfortunately, building each layer at distance exactly \( k \) could lead to too many writes. For example, if the \( k \)th layer has \( \theta(n) \) nodes in it, this would lead to \( \theta(n) \) writes. Instead we choose layers that are close to \( k \) and don’t contain too many nodes. By carefully selecting which layers to label we can guarantee to not write too many nodes.

A query is answered by searching backwards in the labeled graph to distance \( k \) to find labeled nodes, and returning the shortest distance among labeled nodes plus the distance searched backwards on the path. Since every node on a layer is labeled, the shortest path must go through a labeled node.

The main result of this section is captured by the following theorem.

**Theorem 6.2.1.** For an unweighted, \( d \)-bounded degree graph, there exists a data structure that takes \( O(n \log^d \omega) \) writes and \( O(n \omega \log^d \omega) \) reads to build, and can answer queries for BFS distance in \( O(\sqrt{\omega}) \) reads.

Next we will show the algorithm for labeling the graph.

### 6.2.1 BFS Labeling Algorithm

At a high level, our algorithm builds the data structure layer by layer. Define layer \( i \) of graph \( G \) with source node \( s \) to be the set of nodes \( v \) such that \( \text{dist}(s, v) = i \). The source node is the zeroth layer with distance zero. From each layer we build the next layer by searching out \( k \) steps to find all the nodes exactly distance \( k \) away from the current layer. For each reached vertex \( v \) at distance \( k \) we must verify that there is no shorter path to \( v \). Once verified, \( v \) is written to the data structure as part of the next
layer. After all the nodes in a layer are written the algorithm moves onto the next
layer. The algorithm terminates once every vertex in the last layer has been searched
from.

In order to not write too many nodes we instead pick a distance \( j \) that is \( k \) to
\( 2k \) away and has minimal size. This complicates the algorithm slightly because we
have to find the size of each layer \( k \) to \( 2k \) away. One we find the size each layer \( k \) to
\( 2k \) away from the current layer, the layer with the fewest nodes is selected and the
algorithm proceeds with finding nodes in this layer and labelling them.

\textbf{Algorithm 10} Label BFS Distance Layers
Input: Graph \( G \) and source vertex \( s \)
Output: A set of nodes \( L \) with each \( v \in L \) labeled with BFS distance \( d(v) \)

1: \textbf{function} BFSLabeling(\( G, s \))
2: \hspace{1em} \( L \leftarrow \emptyset \)
3: \hspace{1em} Add \( s \) to \( L \) with \( d(s) = 0 \)
4: \hspace{1em} \textit{distance} = 0
5: \hspace{1em} \textbf{while} there is a layer that has not been searched from
6: \hspace{2em} \( j \leftarrow \text{NextLayer}(\textit{distance}) \) \hspace{2em} \( \triangleright \) Find distance to next layer
7: \hspace{2em} \textbf{for} \( v \) in \( L \) with \( d(v) = \textit{distance} \)
8: \hspace{3em} \textbf{while} there are unexplored vertices \( \leq j \) steps forward from \( v \)
9: \hspace{4em} Search from \( v \) to find a vertex \( u \) that is \( j \) steps forward
10: \hspace{4em} \textbf{if} \( \text{dist}(u) = \textit{distance} + j \) and \( u \notin L \) \textbf{then}
11: \hspace{5em} Add \( u \) to \( L \) with \( d(u) = \textit{distance} + j \).
12: \hspace{1em} \textit{distance} \leftarrow \textit{distance} + j

13: \textbf{function} NextLayer(\( \textit{distance} \))
14: \hspace{1em} Maintain an array of counters indexed \( k \) to \( 2k \)
15: \hspace{1em} \textbf{for} each vertex \( x \in L \) with \( d(x) = \textit{distance} \)
16: \hspace{2em} Search from \( x \) for all vertices \( \leq 2k \) steps away
17: \hspace{2em} \textbf{for} each reached vertex \( u \) that is \( s \) steps forward from \( x \), where \( k \leq s \leq 2k \)
18: \hspace{3em} Search backwards from \( u \) (only through incoming edges) for \( s \) steps
19: \hspace{3em} \textbf{if} no recorded vertices are reached in \( < s \) steps and \( u \) hasn’t already
been counted \textbf{then}
20: \hspace{4em} Increment count for this layer
21: \hspace{1em} Return the index of the minimum sized layer
BFS labeling algorithm overview. The algorithm for producing a BFS labeling is shown in Algorithm 10. It takes as input a graph $G$ and source node $s$ and returns a set of nodes that are labeled. The algorithm maintains an array $L$ which will be the set of labeled nodes to return.

The algorithm starts by labeling $s$ with distance 0 and adding $s$ to $L$. Throughout the execution of the algorithm, we maintain a pointer to the next open slot in $L$, and increment the pointer as we add nodes. Each labeled layer is found from the previous one which we set as distance, but first we have to find the layer that is $k$ to $2k$ away with the fewest nodes in it. We will explain later how we do this step. Let the next layer be $j$ away from the current layer. From each vertex $v$ on the recorded layer, search for vertices that are $j$ hops forwards, and let $u$ be a reached node. In Line 10 we verify that the shortest path distance from $s$ to $u$ is $distance + j$, and that $u$ has not already been labeled. To do this check, search backwards from $u$ (only through incoming edges) for $2k$ steps. If any reached vertex $x$ has label $d(x) = i$ where $i < distance$ then $u$ is no added to $L$, because there is a shorter path to $u$ through $x$. If $u$ reaches nodes $x$ where $d(x) = i$, and the distance searched backwards is less than $j$ then there is a shorter path to $u$ through $x$ and $u$ is not labeled. Finally, if $d(x) = i$, and the distance searched backwards is $j$, then only label $u$ if $v$ is earliest in the array among all reached nodes with $d(x) = i$. This ensures that $u$ is only added to $L$ once, namely when the earliest node in the array searches. To label $u$, set $d(u) = distance + j$ and add $u$ to $L$. One we label all the nodes in this layer, continue searching for more layers until every vertex in the array has been searched from.

Select the distance to the next layer. To select the distance of the next recorded layer, the algorithm counts the number of vertices in each layer $k$ to $2k$ steps away and takes the minimum. Maintain an array indexed $k$ to $2k$ to keep track of the counts
for each layer. To count the number of vertices in a layer $j$ steps forward, the algorithm does the same thing as recording a new layer but instead of writing the distance of the vertex once it is reached, increment a counter for that layer. Specifically, when search forwards from $v$ for a vertex $u$ that is $j$ steps forward, search backwards $2k$ from $u$ to ensure the shortest path goes through $v$, and $v$ is the earliest node in the array that reaches $u$ at distance $distance + j$. This part is exactly the same as when we label $u$, and ensures that no vertices are double counted for a layer. Once all vertices $k$ to $2k$ steps forwards have been counted the layer of minimum size is selected.

6.2.2 A k-BFS-Labeling

For a graph $G$ and source node $s$, we define a $k$-$BFS$-labeling to be a set $L$ nodes with labels such that:

1. The source node $s$ is labeled with $d(s) = dist(s, s) = 0$

2. A layer $i$ is labeled if every node in the layer $v$ has $d(v) = dist(s, v) = i$ recorded.

3. Aside from layer 0, each labeled layer is $k$ to $2k$ distance away from the previous labeled layer.

We will show that the algorithm $BFSLABELING$ produces a $k$-BFS labeling for a graph $G$ and source node $s$. It is not hard to see that the algorithm by construction selects layers to label that are $k$ to $2k$ from the previous layer. It remains to show that only nodes $v$ with $dist(s, v) = i$ are recorded to a labeled layer $i$.

**Lemma 6.2.2.** Consider an execution of $BFSLABELING$ on an unweighted graph $G$ and source node $s$. If a node $v$ is labeled with $d(v) = j$, then $dist(s, v) = j$.

**Proof.** Proof by induction on labeled layers. To start, $s$ is labeled with $d(s) = 0$, and $dist(s, s) = 0$. Assume the lemma holds for layers 0 to $i$. Specifically that layers 0
to $i$ are $k$ to $2k$ apart and labeled nodes $v$ in layer $x$ have $d(v) = dist(v) = x$. Let the next labeled layer be $j$, and so $i + k \leq j \leq i + 2k$. Let $u$ be a node that the algorithm labels $d(u) = j$. Since the algorithm searches $j - i$ steps from a node $y$ on layer $i$ to reach $u$ we know there is a path from $s$ to $y$ to $u$ with distance $i + j$ and so $dist(u) \leq j$. If $dist(u) < j$, then consider the last $2k$ nodes on the shortest path to $u$. By the inductive assumption, one of these nodes must be on a labeled layer. Since the algorithm searches $2k$ steps backwards the labeled node on the shortest path is reached and the algorithm knows there is a shorter path to $u$ and so $u$ would not be labeled. Therefore $u$ is only labeled $d(u) = j$ if $dist(u) = j$.

6.2.3 Cost of Building the Data Structure

Next we will show the cost of building the data structure. We assume that $dk < M$, so we can write $O(dk)$ words to the symmetric memory.

**Lemma 6.2.3.** The cost of the BFS labeling algorithm is $O(n/k)$ writes and $O(nd^{4k}/k)$ reads.

**Proof.** For each vertex on a recorded layer, the algorithm searches forwards up to $2k$ steps. This search explores $O(d^{2k})$ edges and nodes, since the graph has bounded degree $d$. We can implement the searches using depth first search with a stack of side $2k$. Since $dk < M$, this can be done in the symmetric memory. For each reached node $u$, the algorithm searches backwards from $u$ for $2k$ steps to check for a shorter path to $u$. This search again reaches $O(d^{2k})$ edges and nodes. For each node on a recorded layer the cost of searches to nodes, and checking the reached nodes shortest path is $O(d^{2k}) \cdot O(d^{2k}) = O(d^{4k})$ reads and zero writes. When selecting the next layer, the algorithm selects the minimum among $k$ layers which means at most $n/k = O(n/k)$
vertices are labeled, and this is the total number of writes. The $O(d^k)$ cost per written vertex makes the total number of reads $O(nd^{2k}/k)$.

6.2.4 Queries

Now that we have built the data structure, we can show how to query the BFS distance for a node, and the cost of the queries.

Algorithm for answering queries. To answer a query for an unrecorded vertex $v$, search backwards $2k$ hops from $v$ for labeled vertices. For each reached labeled node $u$, there is a path to $v$ with cost $d(u)$ plus the distance searched backwards to reach $u$. Return the minimum distance of these paths to $v$. If no node is reached in the backwards search, then report infinity, as there is no path from the source to $v$. Since every node in a given layer is labeled, we can guarantee that the shortest path goes through the labeled layer and the shortest path distance is found.

**Lemma 6.2.4.** Given a graph $G$ and source node $s$ with a $k$-BFS labeling, a node $v$ can be queried for its BFS distance $\text{dist}(s, v)$ in zero writes and $O(d^{2k})$ reads.

**Proof.** The cost of a query is the cost of searching backwards $2k$ hops from the queried vertex which involves exploring at most $O(d^{2k})$ vertices and edges. By assumption that $dk < M$, this search can be done in the symmetric memory using DFS, so there are no additional writes. The number of reads is the number of vertices and edges explored, $O(d^{2k})$.

It remains to show that the algorithm returns the correct distance to $v$. If $\text{dist}(v) \leq 2k$, then when the algorithm searches backwards for $2k$ steps it will reach $s$ and return the correct distance. Otherwise $\text{dist}(v) > 2k$. By construction of the $k$-BFS labeling some layer $j$ where in $\text{dist}(v) - 2k \leq j \leq \text{dist}(v)$ is labeled. The shortest path to $v$
must go through the labeled layer \( j \), and so when the query searches backwards from \( v \) it will reach a labeled node at layer \( j \) which is on the shortest path and therefore return the correct distance to \( v \).

\[ \]

\textit{Proof of Theorem 6.2.1.} Let \( k = (\log_d \omega)/4 \). By Lemmas 6.2.3 and 6.2.4 the theorem holds.

\[ \]

6.3 Depth-First Search

We would like to run depth first search on a directed graph \( G \) from source vertex \( s \) and label vertices with their time of discovery and finish. Once the graph is labeled, we would like to be able to iterate over the vertices in depth first order, and query a vertex for its discovery and finish time. We assume there is an ordering on edges so that we are labeling a specific depth first search over \( G \). We also assume that we have the transpose of \( G \) already written in memory. The standard algorithm runs depth first search and writes the time of each vertex upon discovering it and the time when it finishes searching \([31]\). This algorithm costs \( O(n\omega + m) \) and for bounded degree graphs \( O(n\omega) \). We improve this cost on bounded degree graphs to be \( O(n/\log \omega) \) writes and \( O(n\sqrt{\omega}/\log \omega) \) reads.

We introduce the concept of an implicit representation of a DFS tree on a graph \( G \), and use it to iterate over the vertices of \( G \) in depth first order and answer queries on the discovery and finish time of a vertex.

\textbf{Definition 6.3.1.} \textit{Implicit Labeling of a DFS Tree} A DFS Tree on graph \( G \) is all of the vertices of \( G \) and only the edges that are followed in depth first search on \( G \) from source vertex \( s \). The implicit labeling of a DFS tree labels vertices with discovery and finish time such that for each unlabeled vertex, there is a labeled vertex within \( k \) hops backwards that is an ancestor in the DFS tree.
After the implicit labeling of the DFS tree is completed the cost of iterating over the vertices in depth first order is $O(n/\log \omega)$ writes and $O(n\sqrt{\omega}/\log \omega)$ reads. The cost of querying a vertex is zero writes and $O(\sqrt{\omega})$ reads.

### 6.3.1 Algorithm for DFS Labeling

Suppose we already have the DFS tree, i.e. we know which edges are in the DFS tree and which are not. Then we want to simulate DFS but only writing a $1/k$ fraction of the nodes. Next we will show how we simulate DFS only writing some of the nodes assuming we have the DFS tree, then we will show how we get the DFS tree edges.

Maintain a counter for the time to track the discovery and finish time of nodes. Maintain a second counter for keeping track of the distance away from a specified written nodes. We will simulate DFS using a stack, but only storing the recorded vertices in the stack. We define a $k$-hop descendant and a $k$-hop subtree as follows.

**Definition 6.3.2.** For a vertex $u$, a **$k$-hop descendant** of $u$ is a vertex $v$ such that there is a $k$-edge path from $u$ to $v$ in the DFS tree.

**Definition 6.3.3.** For a vertex $u$ A **$k$-hop subtree** of $u$ is the set of vertices that are within $k$ forwards hops of $v$ in the DFS tree.

Start by recording the source vertex $s$ with discovery time 0, and push $s$ onto the stack. From a recorded vertex $v$, follow edges in DFS order until finding a $k$-hop descendant $x$. Then verify that $x$ has a $k$-hop subtree of at least size $k$. To check the size of the $k$-hop subtree, simulate depth first search until finding $k$ more vertices forwards of $x$, again assuming we have the DFS tree. If $x$ has a $k$-hop subtree of at least size $k$, then push $x$ onto the stack and record the discovery time for $x$. Since a node is only written if it has a $k$ sized $k$-hop subtree at most a $1/k$ fraction of the nodes will be labeled. If $x$ has less than $k$ descendants, then don’t write $x$. Since there
are less than $k$ descendants the distance from a descendant to a recorded vertex is less than $2k$. Upon pushing a vertex onto the stack, start the simulation from this vertex as normal depth first search would, and update the discovery and finish time counter. After a vertex has explored all of its descendants and labeled any vertices, it is popped off the stack and its finish time is written.

Since we don’t have the DFS tree we must modify the algorithm slightly. Before following any edge in the DFS we will check that the edge is in the DFS tree using the algorithm \texttt{CheckEdge}(u, v). If \texttt{CheckEdge}(u, v) returns true then we search over the edge. Otherwise, ignore the edge as it is not in the DFS tree. Algorithm 11 shows the algorithm for labeling the DFS tree. Next we will show the algorithm for \texttt{CheckEdge}(u, v).

\begin{algorithm}
\caption{Label Write-Efficient DFS Tree}
\textbf{Input:} Graph $G$ and source vertex $s$
\begin{algorithmic}[1]
\State Maintain counter for start and finish time
\State Push $s$ onto the stack $S$
\State $s$.start $\leftarrow 0$
\While{$S$ is not empty}
\State Let $v$ be the vertex on top of $S$ (don’t pop)
\State Find all $k$-hop descendant of $v$, but only follow an edge to vertex $u$ according to \texttt{CheckEdge}(u, v)
\State \For{each $k$-hop descendant $x$}
\State Find the $k$-hop subtree of $x$ following an edge to a vertex $u$ according to \texttt{CheckEdge}(u, x)
\If{the size of the $k$-hop subtree is $\geq k$}
\State Push $x$ onto the $S$ and write $x$.start time
\State Break to line 4
\EndIf
\EndFor
\State Pop $v$ off $S$ and write $v$.finish time incrementing the time for finishing searching
\EndWhile
\end{algorithmic}
\end{algorithm}

Check if an edge is in the DFS Tree. Algorithm 12 shows the algorithm for checking if an edge is in the DFS tree. In order to check whether an edge $(u, v)$ is in the DFS tree, it must checked if $v$ was discovered by $u$ or another vertex before $u$. Search
Algorithm 12 Check if Edge \((u, v)\) is in the DFS Tree

Input: Edge \((u, v)\)

Returns true if edge \((u, v)\) is in the DFS Tree

1: function CheckEdge(vertex \(u\), vertex \(v\))
2: Search backwards to find \(C \leftarrow \{x \mid \text{dist}(x, v) \leq 2k, x \text{ is marked}\}\)
3: while \(C\) is non-empty
4: Let \(x\) be the node in \(C\) with earliest discovery time
5: Run DFS from \(x\)
6: if a recorded node \(y\) is reached where \(y \in C\), or \(y \notin C\) but there is a node \(z \in C\) where \(z\)'s interval is nested within \(y\)'s then
7: remove \(x\) from \(C\) and break to Line 3
8: if \(v\) is reached first and edge to \(v\) is \((u, v)\) then return true
9: if \(v\) is reached first and edge to \(v\) is not \((u, v)\) then return false

backwards for up to \(2k\) distance and find all marked vertices, and call this set \(C\). By the labeling on \(G\) a marked vertex will be found within \(2k\) steps. Starting with the node \(x\) in \(C\) with earliest discovery time, simulate DFS until reaching \(v\) or another marked vertex. If \(v\) is reached then check whether the edge that reached \(v\) is \((u, v)\), return true if so, and false if not. If a marked node \(y\) is reached where \(y\) is in \(C\) then \(y\) will find \(v\) before \(x\) does so we can remove \(x\) from \(C\) and continue trying nodes in \(C\). If a marked node \(y\) is reached where \(y \notin C\) but there is a node \(z \in C\) where \(z\)'s interval is nested within \(y\)'s, then \(z\) will find \(v\) before \(x\) does so we can remove \(x\) from \(C\) and continue to search for \(x\) from the next earliest start time in \(C\).

6.3.2 Depth-First Search Traversal Algorithm

In this section we will show how to traverse a graph \(G\) in depth first order, given the write-efficient DFS labeling in Algorithm 11. Our algorithm is given in Algorithm 13.
We will maintain a stack of nodes, and only recorded nodes are added to the stack. To start, the source node $s$ is pushed onto the stack, and we begin simulating DFS by following edges in DFS order. Before searching over any edge $(u, v)$, check that $(u, v)$ is in the DFS tree by using Algorithm 12, CHECKEDGE($u, v$). This algorithm indicates whether to follow the edge in the depth first traversal. When a vertex is recorded vertex is reached, push it onto the stack and start searching from it as with normal depth first search. Likewise, when a vertex is done searching pop it off the stack and return to the new top of the stack. As with normal depth first search, continue to search until the stack is empty.

Algorithm 13 Iterate over the vertices of a graph in depth first order
Input: Graph $G$ with write-efficient DFS labeling and source vertex $s$

1: Push $s$ onto stack $S$
2: while $S$ is not empty
3: Let $v$ be the vertex on the top of $S$ (don’t pop)
4: while there are unexplored vertices from $v$
5: Explore vertices in DFS order from $v$ except check each edge $(x, y)$ is in the DFS tree by calling CHECKEDGE($x, y$), and return vertices in order of discovery
6: if a marked vertex $x$ is reached then
7: Push $x$ onto $S$ and break to Line 2
8: Pop $v$ off of $S$

6.3.3 Algorithm for Querying Start and Finish Time of a Vertex

In this section we will show how to query the start and finish time of a vertex given the write-efficient DFS tree. If a query node is a recorded node then we have the start and finish time recorded already, so we assume the query node is not recorded.

For a node $v$, define the closest marked DFS ancestor of $v$ to be a recorded node $u$ which is the closest ancestor of $v$ in the DFS tree. The query algorithm is as follows. First find the node $u$ which is the closest marked DFS ancestor of $v$. We will describe how to find the closest marked DFS ancestor below. The closest marked DFS ancestor
finds \( v \) before any other vertex in the DFS and within \( 2k \) steps. This means that if DFS is run from \( u \) on only the edges in the DFS tree, the path followed is the path that found \( v \) in DFS on \( G \).

First we need to find which descendants of \( u \) are descendants of \( u \) in the DFS tree so that we can accurately calculate the start and finish times for \( v \). For each node \( x \) reached that is a descendant of \( u \) in the DFS tree, we will add a label on \( x \) that points to the incoming edge to \( x \) in the DFS tree. Once we have these labels we will be able to simulate DFS to find the start and finish times.

Finding the descendants of \( u \) is done by searching forward from \( u \) in DFS order. For each vertex \( x \) reached, check if it already points to an edge that found it, which means it was already visited in depth first search. Next, check if its closest marked ancestor vertex is also \( u \). If it is not, then another vertex discovered \( x \) and so the edge to \( x \) is not in the DFS tree. If it is, then the edge is in the DFS tree, so mark that this edge found \( x \). Explore all the vertices that are forwards of \( u \) and stop each searching along each path that is forwards of \( u \) only when reaching another marked vertex. This distance is up to \( 2k \) steps forward by the labeling on \( G \). Now that all the vertices that are descendants of \( u \) in the DFS tree are marked with which edge found them, run depth first search from \( u \). For each vertex that is reached, only follow the edge to that vertex if the vertex indicates it was reached by that edge. Label each vertex with start time upon discovery and finish time when it finishes searching. When a marked vertex is reached, update the counter to be its finish time and continue to search as if the marked vertex just finished searching. Once \( v \) is labeled with a start and finish time, answer the query.

To find the closest marked DFS ancestor of a vertex \( v \), search backwards for \( 2k \) steps from \( v \). By the labeling on \( G \), a marked vertex will be found. Also because the path that discovered \( v \) must be marked at every \( k^{th} \) vertex, the algorithm guarantees
that the closest marked ancestor in the DFS tree will be found. Any marked vertices that are found are candidates for the closest marked DFS vertex. Let the candidate with the earliest start time vertex be \( u \). By properties of DFS, a nested interval of start and finish times indicates the vertex with inner interval is a descendent of the outer interval vertex [1]. Starting from the earliest start time candidate vertex \( u \), eliminate any other candidates whose intervals are disjoint from \( u \)'s interval. These candidates can be eliminated because \( u \) searched first among candidates and must have found \( v \) since there is a path to \( v \) and finished its search before the other vertex could begin based on it having a disjoint interval. For any remaining candidate vertices, it must be checked whether the \( u \) discovered \( v \) or a descendent of \( u \) discovered \( v \) first, so run DFS from the \( u \) and check whether the a candidate vertex was found first or \( v \) was found first. If \( v \) was found first, then this is DFS ordering, so \( u \) is the marked vertex that discovered \( v \) first. If another candidate vertex was found first or another marked vertex with a candidate nested in its interval, then \( v \) would be found by the next candidate vertex. This is because there is a path from the next candidate to \( v \) and the next candidate will finish searching before \( u \) goes on to search for \( v \). This indicates that the next candidate vertex finds \( v \) before \( u \) does, and it must be checked against any further candidates, so repeat the algorithm with the new candidate as \( u \). Remove all candidates that find \( v \) after another candidate and the closest one will remain.

6.3.4 Depth-First Search Analysis

In this section we will show the costs of running the algorithms in the previous sections. We again consider bounded degree graphs where the maximum degree for any vertex is \( d \). Our main result is captured by the following theorem.

**Theorem 6.3.4.** For a directed graph \( G \), a write-efficient DFS labeling can be computed, such that the graph can be traversed in DFS order. The cost of constructing
Algorithm 14 Query Start and Finish Times

Input: Write-efficient DFS Tree on Graph $G$ and queried vertex $v$

1: function QUERYSTARTFINISHTIMES(vertex $v$)
2:    Let $u$ be CLOSESTMARKEDDFSANCESTOR($v$)
3:    while there are unexplored paths from $u$
4:        Depth first search from $u$, except only follow an edge to vertex $x$ if CLOSESTMARKEDDFSANCESTOR($x$) = $u$, and label $x$ was found by this edge
5:    Simulate DFS from $u$ starting with $start(u)$ for starting time
6:    for each reached vertex $x$
7:        if $x$ does not point to edge that we are following to reach it then
8:            Do not follow the edge to $x$
9:        else
10:            Label with start time upon discovery and finish time when finished searching
11:    if $x$ is a recorded vertex in the DFS tree then
12:        Update time to be $finish(x) + 1$ and do not search past $x$
13:    return $start(v)$ and $finish(v)$

14: function CLOSESTMARKEDDFSANCESTOR(vertex $v$)
15: Search backwards from $v$ for $2k$ steps along all paths for marked vertices
16: Let any found recorded vertex be a candidate
17: while there are still multiple candidates
18:    Let $u$ be the vertex with earliest start time
19:    Any candidate with start to finish time interval not nested in $u$ can be removed from possible candidates
20:    Run DFS from $u$
21:    if another candidate vertex $x$ is reached before $v$ then
22:        remove $u$ from candidates and search from $x$
23:    else if a marked vertex is reached before reaching $v$, and some candidate $x$ has an interval nested within the marked vertex then
24:        remove $u$ from candidates and search from $x$
25:    else
26:        return $u$
27: return only candidate
the labeling and traversing the graph in DFS order is $O(n \log d / \log \omega)$ writes and $O(n \sqrt{\omega} \log d / \log \omega)$ reads. Given the write-efficient DFS labeling and a query node $v$ the start and finish times of $v$ in the DFS ordering can be returned in zero writes and $O(\sqrt{\omega})$ reads.

First we will show the cost of labeling the write-efficient DFS tree.

Lemma 6.3.5. The cost of the write efficient DFS labeling algorithm, Algorithm 10, is $O(n/k)$ writes and $O(nd^{6k}/k)$ reads.

Proof. The algorithm searches out from a marked vertex at most $d^k$ times, specifically for each time that it pops a descendant off the stack and this vertex returns to be the top of the stack. The cost of searching out from a marked vertex is looking at $d^{2k}$ vertices. The cost of exploring any vertex is $d^{4k}$ because it search backwards for up to $d^k$ candidate vertices and then runs depth first search from each of those. Therefore, the cost per recorded vertex of labeling the graph is $d^{2k} \cdot d^{2k} \cdot d^{2k} = d^{6k}$. Since a node is only recorded if it has a $k$-hop subtree of at least size $k$, at most $n/k$ vertices are recorded. Since at most $n/k$ vertices are labeled, the total number of reads is $O(nd^{6k}/k)$. The only writes are the $n/k$ recorded vertices which are written a constant number of times and so there are $O(n/k)$ writes.

Next we turn to the cost of the depth first traversal of a graph $G$, which is Algorithm 13.

Lemma 6.3.6. For a graph $G$, the cost of a depth first traversal on $G$ is $O(n/k)$ writes and $O(nd^{6k}/k)$ reads.

Proof. The traversal uses a stack of all the recorded vertices. Each recorded vertex is pushed onto the stack once and popped off the stack once, which is $O(n/k)$ writes. A recorded vertex searches multiple times because it researches any time one of its
direct ancestors is popped off the stack. Therefore a recorded vertex can search up
to $d^{2k}$ times and each forwards search looks at $d^{2k}$ vertices. Each time a vertex is
explored it costs $d^{2k}$ to check the edge to the vertex is in the tree. For each recorded
vertex the number of reads is $d^{6k}$. In total, the number of reads is $O(nd^{6k}/k)$, and the
number of writes is $O(n/k)$.

**Lemma 6.3.7.** The cost of querying a vertex for discovery and finish time is zero
writes and $O(d^{6k})$ reads.

**Proof.** The number of vertices explored when searching out for $2k$ steps is $d^{4k}$. We
assume that $d^{4k} < M$ so depth first search from a marked vertex to $2k$ steps can be
done in the symmetric memory, and requires no additional writes. The cost of finding
the closest marked DFS ancestor is $d^{4k}$ because the algorithm searches backwards $d^{2k}$
times and searches forwards at most once for each candidate of which there cannot
be more than $d^{2k}$ of. The algorithm finds the closest DFS ancestor for each vertex $2k$
forwards which there can be at most $d^{2k}$ of. The total cost of the query is $d^{6k} = O(\sqrt{\omega})$
reads.

Now we are ready to prove our main theorem of this section.

**Proof of Theorem 6.3.4.** Set $k$ to $\log_d \omega/12$. By Lemmas 6.3.5, 6.3.6 and 6.3.7 we get
the stated number of reads and writes.

6.4 **Strongly Connected Components**

For a directed graph $G$, two nodes $u$ and $v$ are strongly connected if there exists
a path from $u$ to $v$ and a path from $v$ to $u$. A strongly connected component is
a maximal set of nodes in which every pair of nodes is strongly connected. Given a
graph $G$ and a pair of query nodes $u$ and $v$, we would like to be able to return whether
$u$ and $v$ are strongly connected. By using the write-efficient DFS labeling from the previous section, this section will show how to answer a query for whether two nodes are strongly connected. Our algorithm is based on Kosaraju’s linear time algorithm [31] for strongly connected components which we will describe next.

Kosaraju’s algorithm [31] for identifying strongly connected components involves performing two passes of DFS. The first starts from an arbitrary vertex and runs DFS while labeling each vertex with discovery and finish time. The second DFS goes in order of latest finish time in the first search on $G^T$, where $G^T$ is the transpose of $G$. The components are then written out in order based on the second pass of depth first search. Specifically, starting with the node $v$ with latest finish time, each node that is reached in the second DFS from $v$ is the strongly connected component containing $v$, and removes the component from the graph. Then the algorithm moves to the node with the next latest finish time and repeats until each strongly connected component is determined. Kosaraju’s algorithm requires that each component be written out in order to answer a query of whether two vertices are strongly connected. For bounded degree graphs, Kosaraju’s algorithm costs $O(n\omega + m)$.

Our goal is to reduce the size of the output by answering queries without having to write out each component. Our algorithm follows Kosaraju’s algorithm by performing two passes of DFS, with the ordering of the second DFS based on the first one. However, instead of writing out each component, we show that we can query the finish time in each of the DFS trees to determine if two nodes are strongly connected. We can use the write efficient DFS trees from the previous section and the query algorithm to query within them. The result, for bounded degree graphs, is an algorithm which costs $O(n/\log \omega)$ writes and $O(n\sqrt{\omega}/ \log \omega)$ reads for preprocessing, and answers queries in zero writes and $O(\sqrt{\omega})$ reads.
6.4.1 Strongly Connected Components Algorithm

Algorithm 15 shows our algorithm for strongly connected components. The algorithm starts by preprocessing the graph in Lines 1 - 3. First the algorithm builds a write-efficient DFS from an arbitrary vertex, assuming an ordering over the edges, and labels recorded nodes $v$ with discovery $d(v)$ and finish $f(v)$ times. Then the algorithm traverses the DFS tree except following edges in opposite order to get a new ordering over the nodes, $\bar{d}(v)$. We show in the analysis that the discovery time when following the edges in opposite order in the DFS tree is the same as latest finish time order in the original DFS.

Next the algorithm builds a second write-efficient DFS tree on $G^T$, but using the order of nodes $\bar{d}(v)$, starting with the earliest discovery time. The second write-efficient DFS tree records nodes $v$ with discovery $d'(v)$ and finish $f'(v)$ times. The write-efficient DFS trees and the traversal are implemented using the algorithms from Section 6.3.

Once the two DFS trees are built, query finish times of $u$ and $v$. If either vertex has earlier finish time in both of the DFS trees, then the two vertices are strongly connected and otherwise they are not. Note that its possible that the query for one of the nodes $u$ does not reach a marked node. In this case $u$ must have less than $2k$ ancestors, and so all of the $2k$ ancestors can be explored to check if $v$ is one of them.

6.4.2 Strongly Connected Components Analysis

This section will analyze the algorithm from the previous section. First we will show that the algorithm correctly determines if a pair of query nodes are strongly connected. Then we will show the cost of preprocessing the graph and the cost of queries. Our main result is captured by the following theorem.
**Algorithm 15 Write-Efficient Strongly Connected Components**

**Input:** Graph $G$, query nodes $u$ and $v$

1. Build write-efficient DFS tree on $G$ and get discovery $d(v)$ and finish $f(v)$ times for labeled nodes $v$
2. Traverse the DFS tree, following opposite order of edges, and record discovery times $\hat{d}(v)$ for recorded nodes
3. Build write-efficient DFS tree on $G^T$, in order of $\hat{d}(v)$ for nodes, to get discovery $d'(v)$ and finish $f'(v)$ times for labeled nodes $v$
4. Query finish time of $u$ and $v$ on both DFS trees
5. **if** $f(v) < f(u)$ and $f'(v) < f'(u)$, or $f(u) < f(v)$ and $f'(u) < f'(v)$ **then**
6. Return strongly connected
7. **else** Return not strongly connected

**Theorem 6.4.1.** Given a directed graph $G$ with maximum degree $d$, there exists an algorithm which can preprocess the graph such that a pair of nodes $u,v$ can be queried for if they are strongly connected. The cost of preprocessing is $O(n \log d/\log \omega)$ writes and $O(n\sqrt{\omega}\log d/\log \omega)$ reads. The cost of answering a query is zero writes and $O(\sqrt{\omega})$ reads.

We will start by showing that the algorithm correctly determines if a pair of query nodes are strongly connected. Our first claim is that we can run DFS but explore nodes in opposite order to get the latest discovery time. This allows the algorithm to iterate over the graph by discovery time instead of finish time which we showed how to do in Section 6.3.2.

**Lemma 6.4.2.** Given a graph $G$, source node $s$ and an ordering $\pi$ over the nodes, and consider a depth first traversal of $G$ where edges are followed according to the ordering $\pi$. Let $\pi'$ be the reverse order of $\pi$. Consider a depth first traversal of $G$ starting from $s$, following edges in order of $\pi'$. The order of nodes in the first DFS
according to latest finish time, is the same as the order of nodes in the second DFS according to earliest discovery time.

Proof. We show this recursively, by showing that a vertex is in the correct ordering relative to its sibling vertices and descendants. A vertex that is found first in depth first traversal will finish searching after all of its descendants. Therefore a vertex is correctly ordered ahead of all of its descendants for discovery time with opposite ordering and latest finish time. When a vertex searches among its descendants it traverses them in opposite order. This order is the same as the ordering of latest finish times because each descendant starts and finishes searching before the next descendant begins. This means that a vertex is correctly ordered relative to its sibling vertices.

Now we show that given the two passes of depth first search, we can answer a query based on the finish times of the nodes in each of the DFS traversals.

Lemma 6.4.3. Let \( f(v) \) be the finish time of vertex \( v \) in depth first search on \( G \). Let \( f'(v) \) be the finish time of \( v \) on the second search of the strongly connected components algorithm. Vertices \( u \) and \( v \) are in different strongly connected components if and only if \( f(u) < f(v) \) and \( f'(v) < f'(u) \), or \( f(v) < f(u) \) and \( f'(u) < f'(v) \).

Proof. (\( \Rightarrow \)) Different strongly connected components implies \( f(u) < f(v) \) and \( f'(v) < f'(u) \). Assume \( u \) searches before \( v \) according to the ordering on vertices. The case where \( v \) searches first is symmetric.

Case 1: There is no path from \( u \) to \( v \) or \( v \) to \( u \). In this case \( u \) searches first and does not find \( v \) so it has a finish time earlier than \( v \) so \( f(u) < f(v) \). In the second search \( v \) does not find \( u \) because there is no path in the transpose graph. Therefore \( v \) has a earlier finish time than \( u \), and so \( f'(v) < f'(u) \).
Case 2: There is a path from $u$ to $v$ but no path from $v$ to $u$. In the first search $u$ reaches $v$ and so $v$ has an earlier finish time than $u$, $f(v) < f(u)$. In $G^T$ there is no path $u$ to $v$ so $u$ finishes its search before $v$, $f'(u) < f'(v)$.

Case 3: There is a path from $v$ to $u$ but no path from $u$ to $v$. In the first search, $u$ does not find $v$ so $f(u) < f(v)$. In the second search, there is no path from $v$ to $u$ in $G^T$ so $v$ searches and finishes before $u$, which means $f'(v) < f'(u)$.

$(\Leftarrow) f(u) < f(v)$ and $f'(v) < f'(u)$ implies different strongly connected components, which we can prove by showing if $u$ and $v$ are in the same component then one of them has an earlier start time in both the first and second search. Since they are in the same component there is a path from $u$ to $v$ and $v$ to $u$. In the first search $u$ finds $v$ and so $v$ finishes earlier than $u$. In the second search, $u$ searches first and since there is a path from $u$ to $v$ in $G^T$, $u$ finds $v$ and $v$ again has the earlier finish time. Therefore $f(v) < f(u)$ and $f'(v) < f'(u)$ when $u$ and $v$ are in the same component. \hfill \Box

Now we will show the cost of preprocessing the graph. The cost comes from the DFS labeling and DFS traversals which we showed in Section 6.3.

**Lemma 6.4.4.** The cost of preprocessing the graph for the strongly connected components algorithm is $O(n \log d / \log \omega)$ writes and $O(n \sqrt{\omega} \log d / \log \omega)$ reads for bounded degree graphs.

*Proof.* The algorithm builds two write-efficient DFS trees, and iterates over one of the DFS trees in DFS order. By Theorem 6.3.4, these steps can be done in $O(n \log d / \log \omega)$ writes and $O(n \sqrt{\omega} \log d / \log \omega)$ reads. \hfill \Box

The cost of a query is simply the cost of querying each node’s finish time in both of the DFS trees.

**Lemma 6.4.5.** After preprocessing the graph, the cost of a querying whether two vertices are strongly connected is zero writes and $O(\sqrt{\omega})$ reads.

148
Proof. The cost of the query is just the cost of querying both vertices in each of the two DFS trees. By Theorem 6.3.4 the cost of these four queries is $O(\sqrt{\omega})$. \hfill \Box

Now we can show the proof of the main theorem of this section, Theorem 6.4.1.

Proof. By Lemmas 6.4.2 and 6.4.3 and Theorem 6.3.4 the strongly connected components algorithm correctly determines if a pair of nodes are strongly connected. By Lemmas 6.4.4 and 6.4.5 the cost of the algorithm is the stated bounds. \hfill \Box
REFERENCES


[40] Riko Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In Proceedings of the 29th ACM Symposium on Parallelism in


