The Dissertation Committee for Udit Agarwal
certifies that this is the approved version of the following dissertation:

Algorithms, Parallelism and Fine-Grained Complexity for Shortest Path Problems in Sparse Graphs

Committee:

Vijaya Ramachandran, Supervisor

Valerie King

Greg Plaxton

David Zuckerman

Algorithms, Parallelism and Fine-Grained Complexity for Shortest Path Problems in Sparse Graphs

by

Udit Agarwal

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2019

Acknowledgments

I would like to thank my advisor, Prof. Vijaya Ramachandran, for her guidance during my PhD. I would also like to thank the rest of my thesis committee: Prof. Valerie King, Prof. Greg Plaxton, and Prof. David Zuckerman.

I thank the Department of Computer Science for providing me with the TA support for the major duration of my PhD studies as well as a supplemental Graduate Calhoun Fellowship for the first three years of my PhD. This research was also supported in part by National Science Foundation under grant NSF-CCF-1320675 for first three years.

Udit Agarwal

The University of Texas at Austin

December 2019

Algorithms, Parallelism and Fine-Grained Complexity for Shortest Path Problems in Sparse Graphs

Publication No. _____

Udit Agarwal, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Vijaya Ramachandran

Computation of shortest paths is one of the classical problems in theoretical computer science. Given a pair of nodes s and t in a graph G, the goal is to find a path of minimum weight from s to t. Most graphs that commonly occur in practice are sparse graphs. In this work, we deal with several computational problems related to shortest paths in sparse graphs and we present algorithms that provide significant improvements in performance in both sequential and distributed settings. We also present fine-grained reductions that establish fine-grained hardness for several problems related to shortest paths.

In the sequential context, we consider the fine-grained complexity of sparse

graph problems whose time complexities have stayed at $\tilde{O}(mn)$ over the past several decades, where m is the number of edges and n is the number of vertices in the input graph. All of these problems are known to be subcubic equivalent and this shows that achieving sub-mn running time is hard, but only for dense graphs where $m = \Theta(n^2)$. We introduce the notion of a sparse reduction which preserves the sparsity of graphs, and we present near linear-time sparse reductions between various pairs of graph problems in the $\tilde{O}(mn)$ class. We also introduce the MWC-hardness conjecture, which states that Minimum Weight Cycle problem cannot be solved in sub-mn time. We establish that several important graph problems in the $\tilde{O}(mn)$ class such as APSP, second simple shortest path (2-SiSP), Radius, and Betweenness Centrality are MWC-Hard, establishing sub-mn fine-grained hardness for these problems.

A well-known generalization of the shortest path problem is the k-simple shortest paths (k-SiSP) problem, where we want to find k simple paths from s to t in a non-decreasing order of their weight. In this thesis we present a new approach for computing all pairs k simple shortest paths (k-APSiSP), which is based on forming suitable path extensions to find simple shortest paths; this method is different from the 'detour finding' technique used in all prior work on computing multiple simple shortest paths, replacement paths, and distance sensitivity oracles. The $\tilde{O}(mn)$ time bound of our 2-APSiSP algorithm matches the fine-grained time complexity for the simpler 2-SiSP problem, which is the single source-sink version of this problem.

Computing APSP is one of the most fundamental problems in distributed computing. We present a simple $\tilde{O}(n^{3/2})$ rounds deterministic algorithm for computing APSP in the well-known CONGEST model which is the first $o(n^2)$ round deterministic algorithm for this problem. We then improve this further by reducing the round complexity to $\tilde{O}(n^{4/3})$. We also present a faster algorithm for graphs with moderate integer edge weights. We develop several derandomization techniques for our deterministic APSP algorithms. These include efficient deterministic distributed

algorithms for computing a small blocker set, which is a set that intersects a desired collection of shortest paths, and several deterministic pipelined approaches for computing the shortest path distance values as well as for propagating the messages in the network. Aside from our deterministic results, all non-trivial distributed algorithms currently known for computing APSP are randomized.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Shortest Paths

Computation of shortest paths is one of the well studied problems in theoretical computer science. The goal is to find a shortest path from a source vertex s to a sink t in a given graph G. In this thesis we study the all-pairs version of shortest path problem, known as all-pairs shortest paths (APSP).

This problem has been well studied over the years. A simple solution is to run n single-source shortest path (SSSP) computations using either Dijkstra [27] for non-negative edge weights or Bellman-Ford [15] for negative edge weights. In fact there exists algorithms that improve on these running bounds. For APSP with arbitrary edge weights, there is an $O(mn + n^2 \log\log n)$ time algorithm for directed graphs [76] and an even faster $O(mn \log \alpha(m, n))$ time algorithm for undirected graphs [77], where α is a certain natural inverse of the Ackermann's function. For integer weights, there is an $O(mn)$ time algorithm for undirected graphs [85] and an $O(mn + n^2 \log\log n)$ time algorithm for directed graphs [42].

For dense graphs (when $m = O(n^2)$), Floyd-Warshall's algorithm [31] can be used to compute APSP in $O(n^3)$ time, and without any modification, this algorithm

works even if the graph has negative edge weights (but no negative weight cycle). For sparse graphs with negative edge weights, one can use Johnson's transformation [54] to transform the graph into a non-negative edge weighted graph in $O(mn + n^2 \log n)$ time and then compute APSP on it.

Since most graphs that occur in practice are sparse, in this work our emphasis is on algorithms where input size is parameterized in terms of m and n. In this thesis we explore the shortest path and cycle problems in both sequential and distributed/parallel settings. We also look at the fine-grained hardness for some of these problems. In the next section we give an overview of the topics covered in this thesis.

1.2 Overview of Topics

1.2.1 Sequential Results

1.2.1.1 Fine-Grained Complexity for Sparse Graphs

In Chapter 2 we consider the fine-grained complexity of sparse graph problems whose time complexities have stayed at $\tilde{O}(mn)$ over past several decades, where m is the number of edges and n is the number of vertices in the input graph. All of these problems are known to be subcubic equivalent and this shows that achieving sub-mn running time is hard, but only for dense graphs where $m = \Theta(n^2)$. Hence the dense subcubic results are not relevant for sparse graphs.

We introduce the notion of a sparse reduction which preserves the sparsity of graphs, and we present near linear-time sparse reductions between various pairs of graph problems in the $\tilde{O}(mn)$ class. We also introduce the MWC-hardness conjecture, which states that Minimum Weight Cycle problem cannot be solved in sub-mn time. We establish that several important graph problems in the $\tilde{O}(mn)$ class such as APSP, second simple shortest path (2-SiSP), Radius, and Betweenness Central-

ity are MWC-Hard, establishing sub-mn fine-grained hardness for these problems. Currently the Minimum Weight Cycle problem can be solved in $O(mn)$ time [71] and the problem of obtaining a $o(mn)$ time algorithm for MWC is open from many decades.

We also identify Eccentricities and BC as key problems in the $\tilde{O}(mn)$ class which are simultaneously MWC-hard, SETH-hard and k-DSH-hard, where SETH is the Strong Exponential Time Hypothesis, and k-DSH is the hypothesis that a dominating set of size k cannot be computed in time polynomially smaller than n^k.

1.2.1.2 k-Simple Shortest Paths and Cycles

In Chapter 2 we showed that k-SiSP does not have a sub-mn time algorithm unless Minimum Weight Cycle can be solved in sub-mn time. In Chapter 3 we present several results for the problem of computing k simple shortest paths (k-SiSP), where we want to find k simple paths from s to t in a non-decreasing order of their weight. We present a new approach for computing all pairs k simple shortest paths (k-APSiSP), which is based on forming suitable path extensions to find simple shortest paths; this method is different from the 'detour finding' technique used in all prior work on computing multiple simple shortest paths, replacement paths, and distance sensitivity oracles. The $\tilde{O}(mn)$ time bound of our 2-APSiSP algorithm matches the fine-grained time complexity for the simpler 2-SiSP problem, which is the single source-sink version of this problem. For $k = 3$ our algorithm runs in $O(mn^2 + n^3 \log n)$ time, which is almost a factor of n faster than the best previous algorithm.

We also present new results for related paths and cycle problems, such as enumerating k simple cycles in non-decreasing order of weight, for which we give an $\tilde{O}(mn)$ time algorithm and we also show that it is MWC-hard for any constant k.

1.2.2 Distributed Results

1.2.2.1 Deterministic Distributed All Pairs Shortest Paths in $\tilde{O}(n^{3/2})$ Rounds

In Chapters 4-7 we consider the shortest path problems in the distributed setting, specifically the all pairs shortest path (APSP) problem. After giving an initial introduction to the distributed APSP problem in Chapter 4, in Chapter 5 we present an $\tilde{O}(n^{3/2})$ rounds algorithm for computing APSP in directed graphs with arbitrary edge weights in the well-known CONGEST model (Section 4.2). This was the first $o(n^2)$ round non-trivial deterministic algorithm for this problem. The most critical component of this algorithm is a new distributed algorithm for computing a small blocker set deterministically, which is a set that intersects a desired collection of shortest paths.

1.2.2.2 Deterministic Distributed All Pairs Shortest Paths Through Pipelining

Chapter 6 presents an improved deterministic algorithm for the distributed APSP problem in the CONGEST model for specific range of integer edge weights (including zero edge weights) and shortest path distances. The most non-trivial component of this algorithm is a novel deterministic pipelined algorithm for computing h-hop shortest path distances for all pairs in a weighted graph. We also introduce the notion of h-hop Consistent SSSP (CSSSP) collection to create a consistent collection of h-hop shortest paths across all source nodes.

1.2.2.3 Faster Deterministic Distributed All Pairs Shortest Paths

In Chapter 7 we present an $\tilde{O}(n^{4/3})$ rounds deterministic algorithm for computing weighted APSP problem in the CONGEST model. This algorithm works for directed graphs with arbitrary real edge weights, and it improves on the round bound pre-

sented in Chapter 5. The main component of this algorithm is a new faster technique for computing a small blocker set deterministically and a new pipelined method for deterministically propagating distance values from source nodes to the blocker nodes in the network.

1.3 Organization

This thesis is organized as follows: In Chapter 2, we describe our fine-grained results for the shortest path problems. Chapter 3 describes our results for the k-simple shortest path and cycle problems. In Chapter 4, we present an overview of the general strategy used in the distributed APSP algorithms. Chapter 5 describes our $\tilde{O}(n^{3/2})$ rounds deterministic algorithm for computing all pairs shortest path in the CONGEST model. Chapter 6 presents our pipelined approach for the distributed all pairs shortest paths problem, which yield improvements for specific range of edge weights and shortest path distances. In Chapter 7, we describe our $\tilde{O}(n^{4/3})$ rounds deterministic algorithm for the all pairs shortest path problem with arbitrary edge weights, and it improves on the bound in Chapter 5.

Part I

Sequential Results

Chapter 2

Fine-Grained Complexity for Sparse Graphs

2.1 Introduction

In recent years there has been considerable interest in determining the fine-grained complexity of problems in P, see e.g. [88]. For instance, the 3SUM [35] and OV (Orthogonal Vectors) [90, 14] problems have been central to the fine-grained complexity of several problems with quadratic time algorithms, in computational geometry and related areas for 3SUM and in edit distance and related areas for OV. APSP (all pairs shortest paths) has been central to the fine-grained complexity of several path problems with cubic time algorithms on dense graphs [91].

For several graph problems related to shortest paths that currently have $\tilde{O}(n^3)$ [1] time algorithms, *equivalence* under sub-cubic reductions has been shown in work starting with [91] between APSP, finding a minimum weight cycle (MWC), finding a second simple shortest path from a given vertex u to a given vertex v (2-SiSP) in a weighted directed graph, finding a minimum weight triangle (Min-Wt-Δ),

[1] $\tilde{O}$ and $\tilde{\Theta}$ can hide sub-polynomial factors; in our new results they only hide polylog factors.

etc. This gives compelling evidence that a large class of problems on dense graphs is unlikely to have sub-cubic algorithms as a function of n, the number of vertices, unless fundamentally new algorithmic techniques are developed.

We consider a central collection of graph problems related to APSP, which refines the subcubic equivalence class. We let n be the number of vertices and m the number of edges. All of the sub-cubic equivalent graph problems mentioned above (and several others) have $\tilde{O}(mn)$ time algorithms; additionally, many sub-cubic equivalent problems related to minimum triangle detection and triangle listing have $O(m^{3/2})$ time complexities for sparse graphs [52]. When a graph is truly sparse with $m = O(n)$ the $\tilde{O}(mn)$ APSP bound is essentially optimal or very close to optimal, since the size of the output for APSP is n^2. Thus, a cubic in n bound for APSP does not fully capture what is currently achievable by known algorithms, especially since graphs that arise in practice tend to have m close to linear in n or at least are *sparse*, i.e., have $m = O(n^{1+\delta})$ for $\delta < 1$. This motivates our study of the fine-grained complexity of graph path problems in the $\tilde{O}(mn)$ class.

Another fundamental problem in the $\tilde{O}(mn)$ class is MWC (Minimum Weight Cycle). In both directed and undirected graphs, MWC can be computed in $\tilde{O}(mn)$ time using an algorithm for APSP. Recently Orlin and Sedeno-Noda [71] gave an improved $O(mn)$ time algorithm for directed MWC. This is an important result but the bound still remains $\Omega(mn)$. Finding an MWC algorithm that runs polynomially faster than mn time is a long-standing open problem in graph algorithms.

We present both fine-grained reductions and hardness results for graph problems in the $\tilde{O}(mn)$ class, most of which are equivalent under sub-cubic reductions on dense graphs, but now taking sparseness of edges into consideration. We use the current long-standing upper bound of $\tilde{O}(mn)$ for these problems as our reference, both for our fine-grained reductions and for our hardness results. Our results give a partial order on hardness of several problems in the $\tilde{O}(mn)$ class, with equivalence

within some subsets of problems, and gives rise to a hardness conjecture (MWC hardness) for this class. Most of the sub-cubic reductions in previous work (including all in [91]) created dense intermediate problems and hence are not fine-grained reductions for the $\tilde{O}(mn)$ class. Fine-grained reductions and hardness results with respect to time bounds that consider only n or only m, such as bounds of the form m^2, n^2, n^c, $m^{1+\delta}$, are given in [78, 5, 3, 4, 45]. One exception is in [2] where some reductions that preserve graph sparsity are given for problems with $\tilde{O}(mn)$ time algorithms, such as diameter and some betweenness centrality problems. However, these are either reductions that start from a triangle finding problem that can be solved in $\tilde{O}(m^{3/2})$, hence not fine-grained reductions for the $\tilde{O}(mn)$ class, or start from a problem, such as Diameter, that is not known to be sub-cubic equivalent to APSP. In contrast, the time bounds in our fine-grained results for the $\tilde{O}(mn)$ class consider both m and n.

In [66], Lincoln et al. formulated the Min-Weight-k-Clique hypothesis, which states that a minimum weight k-clique cannot be found in time smaller than $n^{k-o(1)}$ for sufficiently large edge weights. They show that for all sparsities of the form $m = \Theta(n^{1+1/l})$ where l is a constant, MWC cannot be computed in $O(mn^{1-\epsilon} + n^2)$ time under Min-Weight-k-Clique hypothesis. Our sparse reduction results for directed graphs in conjunction with their result show that a large class of problems in the mn class including Second Shortest Path, Replacement Paths, Radius, and Betweenness Centrality also do not have $O(mn^{1-\epsilon} + n^2)$ time algorithm under Min-Weight-k-Clique hypothesis. Similarly our sparse reduction result from MWC to APSP in undirected graphs in conjunction with this result also establishes that there is no $O(mn^{1-\epsilon} + n^2)$ time algorithm for undirected APSP under Min-Weight-k-Clique hypothesis.

2.2 Definitions of Graph Problems

We will deal with either an unweighted graph $G = (V, E)$ or a weighted graph $G = (V, E, w)$, where the weight function is $w : E \rightarrow \mathcal{R}^+$. We assume that the vertices have distinct labels with $\lceil \log n \rceil$ bits. Let W_{max} and W_{min} denote the largest and the smallest edge weight, and let the *edge weight ratio* be $\rho = W_{max} / W_{min}$. Let $d_G(x, y)$ denote the length (or weight) of a shortest path from x to y in G, and for a cycle C in G, let $d_C(x, y)$ denote the length of the shortest path from x to y in C. We deal with only simple graphs in this thesis.

We present results for the following graph problems in this chapter.

All Pairs Shortest Paths (APSP). This is the problem of computing the shortest path distances for every pair of vertices in G together with a concise representation of the shortest paths, which in our case is an $n \times n$ matrix, $Last_G$, that contains, in position (x, y), the predecessor vertex of y on a shortest path from x to y.

All Pairs Shortest Distances (APSD). Given a graph $G = (V, E)$, the APSD problem only involves computing the shortest path distances for every pair of vertices in G. Most of the currently known APSD algorithms, including matrix multiplication based methods for small integer weights [82, 83, 94], can compute APSP in the same bound as APSD.

Minimum Weight Cycle (MWC). Given a graph $G = (V, E)$, the minimum weight cycle problem is to find the weight of a minimum weight cycle in G.

All Nodes Shortest Cycles (ANSC). Given a graph $G = (V, E)$, the ANSC problem is to find the weight of a shortest cycle through each vertex in G.

Replacement Paths. Given a graph $G = (V, E)$ and a pair of vertices s, t, the replacement paths problem is to find, for each edge e on the shortest path from s to t, a shortest path from s to t avoiding e.

k-SiSP. Given a graph $G = (V, E)$ and $s, t \in V$, the k-SiSP problem is to find the k

shortest simple paths from s to t: the i-th path must be different from first $(i-1)$ paths and must have weight greater than or equal to the weight of any of these $(i-1)$ paths.

k-SiSC. The corresponding cycle version of k-SiSP is known as k-SiSC, where the goal is to compute the k shortest simple cycles through a given vertex x, such that the i-th cycle generated is different from all previously generated $(i-1)$ cycles and has weight greater than or equal to the weight of any of these $(i-1)$ cycles.

Radius. For a given graph $G = (V, E)$, the Radius problem is to compute the value $\min_{x \in V} \max_{y \in V} d_G(x, y)$. The *center* of a graph is the vertex x which minimizes this value.

Diameter. For a given graph $G = (V, E)$, the Diameter problem is to compute the value $\max_{x,y \in V} d_G(x, y)$.

Eccentricities. For a given graph $G = (V, E)$, the Eccentricities problem is to compute the value $\max_{y \in V} d_G(x, y)$ for each vertex $x \in V$.

Betweenness Centrality (BC). For a given graph $G = (V, E)$ and a node $v \in V$, the Betweenness Centrality of v, $BC(v)$, is the value $\sum_{s,t \in V, s,t \neq v} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$, where $\sigma_{s,t}$ is the number of shortest paths from s to t and $\sigma_{s,t}(v)$ is the number of shortest paths from s to t passing through v.

As in [2] we assume that the graph has unique shortest paths, hence $BC(v)$ is simply the number of s, t pairs such that the shortest path from s to t passes through v.

All Nodes Betweenness Centrality (ANBC). The all-nodes version of Betweenness Centrality: determine $BC(v)$ for all vertices.

Positive Betweenness Centrality (Pos BC). Given a graph $G = (V, E)$ and a vertex v, the Pos BC problem is to deternine if $BC(v) > 0$.

All Nodes Positive Betweenness Centrality (Pos ANBC). The all-nodes ver-

sion of Positive Betweenness Centrality.

Reach Centrality (RC). For a given graph $G = (V, E)$ and a node $v \in V$, the Reach Centrality of v, $RC(v)$, is the value
$\max_{s,t \in V: d_G(s,v)+d_G(v,t)=d_G(s,t)} \min(d_G(s, b), d_G(b, t))$

2.3 Our Contributions

In this section we give an highlight of our fine-grained results in [7].

I. Sparse Reductions and the mn Partial Order. In Definition 2.3.1 below, we define the notion of a sparsity preserving reduction (or *sparse reduction*) from a graph problem P to a graph problem Q that allows P to inherit Q's time bound for the graph problem as a function of both m and n, as long as the reduction is efficient. Our definition is in the spirit of a Karp reduction [56], but slightly more general, since we allow a constant number of calls to Q instead of just one call in a Karp reduction (and we allow polylog calls for $\tilde{O}(\cdot)$ bounds).

One could consider a more general notion of a sparsity preserving reduction in the spirit of Turing reductions as in [37] (which considers functions of a single variable n). However, for all of the many sparsity preserving reductions we present here, the simpler notion defined below suffices. It should also be noted that the simple and elegant definition of a Karp reduction suffices for the vast majority of known NP-completeness reductions. The key difference between our definition and other definitions of fine-grained reductions is that it is fine-grained with regard to both m and n, and respects the dependence on *both parameters.*

It would be interesting to see if some of the open problems left by our work on fine-grained reductions for the mn class can be solved by moving to a more general sparsity preserving reduction in the spirit of a Turing reduction applied to functions of both m and n. We do not consider this more general version here since we do not

need it for our reductions.

Definition 2.3.1 (**Sparsity Preserving Graph Reductions**). *Given graph problems P and Q, there is a* sparsity preserving $f(m,n)$ reduction from P to Q, *denoted by $P \leq^{\boldsymbol{sprs}}_{f(m,n)} Q$, if given an algorithm for Q that runs in $T_Q(m,n)$ time on graphs with n vertices and m edges, we can solve P in $O(T_Q(m,n)+f(m,n))$ time on graphs with n vertices and m edges, by making a constant number of oracle calls to Q.*

For simplicity, we will refer to a sparsity preserving graph reduction as a *sparse reduction*, and we will say that P *sparse reduces to* Q . Similar to Definition 2.3.1, we will say that P *tilde-$f(m,n)$ sparse reduces to* Q, denoted by $P \lesssim^{\boldsymbol{sprs}}_{f(m,n)} Q$, if, given an algorithm for Q that runs in $T_Q(m,n)$ time, we can solve P in $\tilde{O}(T_Q(m,n) + f(m,n))$ time (by making polylog oracle calls to Q on graphs with $\tilde{O}(n)$ vertices and $\tilde{O}(m)$ edges). We will also use $\equiv^{\boldsymbol{sprs}}_{f(m,n)}$ and $\cong^{\boldsymbol{sprs}}_{f(m,n)}$ in place of $\leq^{\boldsymbol{sprs}}_{f(m,n)}$ and $\lesssim^{\boldsymbol{sprs}}_{f(m,n)}$ when there are reductions in both directions. In a weighted graph we allow the $\tilde{O}$ term to have a $\log \rho$ factor. (Recall that $\rho = W_{max}/W_{min}$.

We present several sparse reductions for problems that currently have $\tilde{O}(mn)$ time algorithms. This gives rise to a partial order on problems that are known to be sub-cubic equivalent, and currently have $\tilde{O}(mn)$ time algorithms. For the most part, our reductions take $\tilde{O}(m+n)$ time (many are in fact $O(m+n)$ time), except reductions to APSP take $\tilde{O}(n^2)$ time. This ensures that any improvement in the time bound for the target problem will give rise to the same improvement to the source problem, to within a polylog factor. Surprisingly, very few of the known sub-cubic reductions for the problems we consider carry over to the sparse case (and none from [91]). This is due to one or both of the following features.

1. A central technique used in many of the reductions that show sub-cubic equivalence to APSP is to reduce from triangle finding problems such as Min-Wt-Δ (e.g., [2, 91]). These reductions start with a triangle finding problem and proceed by constructing suitable tripartite graphs where the property of the target

Table 2.1: Our sparse reduction results for undirected graphs. The definitions for these problems are in Section 2.2. Note that Min-Wt-Δ can be solved in $m^{3/2}$ time.

Reduction	Prior Results (Undirected)	Our Results
MWC $\leq$ APSD	MWC $\leq$ Min-Wt-Δ [79] *a.* Dense $\tilde{O}(n^2)$ reduction *b.* $\Theta(n^2)$ edges in reduced graphs Min-Wt-Δ $\leq$ APSD (trivial)	***a.* Sparse $\tilde{O}(n^2)$ Reduction** ***b.* $\Theta(m)$ edges in reduced graphs**
ANSC (Unweighted) $\leq$ APSP	Sparse $\tilde{O}(mn^{\frac{3-\omega}{2}})$ reduction [93] *a.* randomized *b.* polynomial calls to APSP *c.* gives randomized $\tilde{O}(n^{\frac{\omega+3}{2}})$ time algorithm [93]	**Sparse $\tilde{O}(n^2)$ Reduction** ***a.* deterministic** ***b.* $\tilde{O}(1)$ calls to** APSP ***c.* gives deterministic $\tilde{O}(n^\omega)$ time algorithm**

problem can be used to detect a desired triangle. However, a cycle can have $\Theta(n)$ vertices, and using such an approach starting with MWC would create n-partite graphs with $\Theta(n^2)$ vertices. Hence this approach does not work in our sparse setting and this highlights the need to develop new techniques to reduce from MWC. Further as noted above, in the sparse setting, all triangle finding and enumeration problems are in $\tilde{O}(m^{3/2})$ time, which is an asymptotically smaller bound than mn for graphs with $m = o(n^2)$. Hence reductions to triangle finding problems (e.g., [79] in the dense case) are not relevant in the sparse setting (unless the mn time bound can be improved).

2. Many of the known sub-cubic reductions convert a sparse graph into a dense one (e.g., [79, 91]), and again these are not relevant in the sparse setting.

Tables 2.1 and 2.2 summarize the improvements our reductions achieve over prior results.

(a) Undirected Graphs: Finding the weight of a minimum weight cycle (MWC) is

Table 2.2: Our sparse reduction results for directed graphs. The definitions for these problems are in Section 2.2. Note that Min-Wt-Δ can be solved in $m^{3/2}$ time.

Reduction	Prior Results (Directed)	Our Results
MWC	Dense $\tilde{O}(n^2)$ MWC $\leq$ Min-Wt-Δ [79]	**Sparse $O(m)$**
$\leq$ 2-SiSP	Dense $O(n^2)$ Min-Wt-Δ $\leq$ 2-SiSP [91]	**Reduction**
2-SiSP	Sparse $O(n^2)$ 2-SiSP $\leq$ APSP [41]	**Sparse $\tilde{O}(m)$**
$\leq$ Radius;	but Dense sub-n^3 APSP $\leq$ Min-Wt-Δ [91]	**Reductions**
$\leq$ BC	Sparse $\tilde{O}(m)$ Min-Wt-Δ $\leq$ Radius, BC [2]	
Replacement	Sparse $O(n^2)$ Rep. Paths $\leq$ APSP [41]	**Sparse $\tilde{O}(m)$**
paths	but Dense sub-n^3 APSP $\leq$ Min-Wt-Δ [91]	**Reductions**
$\leq$ ANSC;	Sparse $O(m)$ Min-Wt-Δ $\leq$ ANSC (trivial)	
$\leq$ Eccentricities	Sparse $\tilde{O}(m)$ Min-Wt-Δ $\leq$ Eccentricities [2]	
ANSC	Sparse $O(m)$ ANSC $\leq$ APSP (trivial)	**Sparse $\tilde{O}(m)$**
$\leq$ ANBC	but Dense sub-n^3 APSP $\leq$ Min-Wt-Δ [91]	**Reduction**
	Sparse $\tilde{O}(m)$ Min-Wt-Δ $\leq$ ANBC [2]	

a fundamental problem. A simple sparse $O(m+n)$ reduction from MWC to APSD is known for directed graph but it does not work in the undirected case mainly because an edge can be traversed in either direction in an undirected graph, and known algorithms for the directed case would create non-simple paths when applied to an undirected graph. Roditty and Williams [79], in a follow-up to [91], pointed out the challenges of reducing from undirected MWC to APSD in sub-n^ω time, where ω is the matrix multiplication exponent, and then gave a $\tilde{O}(n^2)$ reduction from undirected MWC to undirected Min-Wt-Δ in a *dense* bipartite graph. But a reduction that increases the density of the graph is not helpful in our sparse setting. Instead, in this chapter we give a sparse $\tilde{O}(n^2)$ time reduction from undirected MWC to APSD. Similar techniques allow us to obtain a sparse $\tilde{O}(n^2)$ time reduction from undirected ANSC (All Nodes Shortest Cycles) [93, 81], which asks for a shortest cycle through every vertex) to APSP. This reduction improves the running time for unweighted ANSC in *dense* graphs [93], since we can now solve it in $\tilde{O}(n^\omega)$ time using the unweighted APSP algorithm in [82, 12]. Our ANSC reduction and resulting improved algorithm is only for unweighted graphs and extending it to weighted graphs appears to be challenging.

We introduce a new *bit-sampling technique* in these reductions. This technique contains a simple construction with exactly $\log n$ hash functions for Color Coding [13] with 2 colors. Our bit-sampling method also gives the first near-linear time algorithm for k-SiSC in weighted undirected graphs.

The full proofs of Theorems 2.3.2 and 2.3.3 are in Section 2.4.1 and Section 2.9.1 respectively.

Theorem 2.3.2. *In a weighted undirected n-node m-edge graph with edge weight ratio $\rho = W_{max}/W_{min}$ where W_{max} is the largest edge weight and W_{min} is the smallest edge weight, MWC can be computed with $2 \cdot \log n \cdot \log \rho$ calls to APSD on graphs with $2n$ nodes, at most $2m$ edges, and edge weight ratio at most ρ, with $O(n+m)$ cost for*

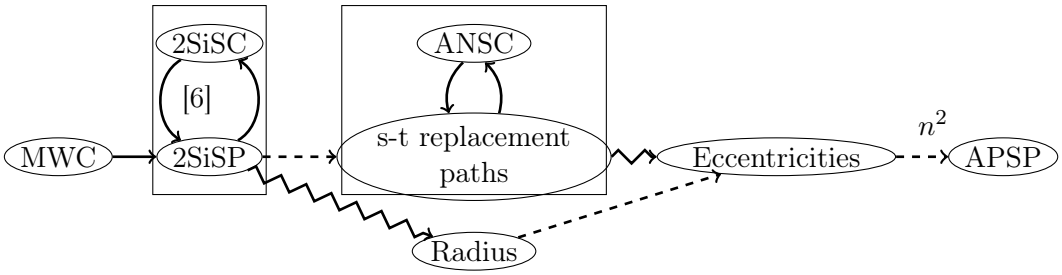


Figure 2.1: Our sparse reductions for weighted directed graphs. The regular edges represent sparse $O(m+n)$ reductions, the squiggly edges represent tilde-sparse $O(m+n)$ reductions, and the dashed edges represent reductions that are trivial. The n^2 label on dashed edge to APSP denotes an $O(n^2)$ time reduction.

constructing each reduced graph, and with additional $O(n^2 \cdot \log n \cdot \log(n\rho))$ *processing time. Additionally, every edge in the reduced graph retains its corresponding edge weight from the original graph. Hence, MWC* $\lesssim_{n^2}^{\boldsymbol{sprs}}$ *APSD.*

In undirected graphs with integer weights at most W_{max} , APSD can be computed in $\tilde{O}(W_{max} \cdot n^\omega)$ time [82, 83]. In [79], the authors give an $\tilde{O}(W_{max} \cdot n^\omega)$ time algorithm for undirected MWC in such graphs by preprocessing using a result in [67] and then making $\tilde{O}(1)$ calls to an APSD algorithm. By applying our sparse reduction in Theorem 2.3.2 we can get an alternate *simpler* $\tilde{O}(W_{max} \cdot n^\omega)$ time algorithm for MWC in undirected graphs.

The following result gives an improved algorithm for ANSC in undirected unweighted graphs.

Theorem 2.3.3. *In undirected unweighted graphs, ANSC* $\lesssim_{n^2}^{\boldsymbol{sprs}}$ *APSP and ANSC can be computed in* $\tilde{O}(n^\omega)$ *time.*

(b) Directed Graphs. We give several nontrivial sparse reductions starting from MWC in directed graphs, as noted in the following theorem (also highlighted in Figure 2.1).

Theorem 2.3.4 (**Directed Graphs.**)**.** *In weighted directed graphs:*

1. *MWC* $\leq^{\boldsymbol{sprs}}_{m+n}$ *2-SiSP* $\leq^{\boldsymbol{sprs}}_{m+n}$ *s-t replacement paths* $\equiv^{\boldsymbol{sprs}}_{m+n}$ *ANSC* $\lesssim^{\boldsymbol{sprs}}_{m+n}$ *Eccentricities*

2. *2-SiSP* $\lesssim^{\boldsymbol{sprs}}_{m+n}$ *Radius* $\leq^{\boldsymbol{sprs}}_{m+n}$ *Eccentricities, and*

3. *2-SiSP* $\lesssim^{\boldsymbol{sprs}}_{m+n}$ *BC*

Sparse Reductions for Directed versus Undirected Graphs. We present fewer fine-grained reductions for undirected graphs than for directed graphs. This is due to the following reasons: First, the 2-SiSC, 2-SiSP and Replacement Paths problem can be solved in near-linear time in undirected graphs and hence they are not part of our partial order for undirected graphs. Second, as highlighted before, an edge can be traversed in either direction in an undirected graph hence a shortest cycle problem, especially when the size of the cycle is unrestricted, cannot be readily reduced to a shortest path problem in undirected graphs. We confront this problem with our bit-sampling technique, which we use in all of our reductions involving cycle problems in undirected graphs. This technique also gives a new near-linear time algorithm for weighted k-SiSC, a new $\tilde{O}(n^\omega)$ time algorithm for unweighted ANSC, and a simple $\tilde{O}(W_{max} \cdot n^\omega)$ algorithm for MWC with integer weights in $[1 \dots W_{max}]$.

II. <u>Conditional Hardness Results</u>. Conditional hardness under fine-grained reductions falls into several categories such as 3SUM hardness [35], OV hardness [90, 14, 88], SETH hardness [51, 90, 88] and sub-cubic APSP hardness for graph problems on dense graphs [91].

In our work we focus on the mn class, the class of graph path problems for which the current best algorithms run in $\tilde{O}(mn)$ time. This class differs from all previous classes considered for fine-grained complexity since it depends on two parameters of the input, m and n. To formalize hardness results for this class we use the following notion of a sub-mn time bound.

Definition 2.3.5 (**Sub-mn**). *A function $g(m,n)$ is* sub-mn *if $g(m,n) = O(m^{\alpha} \cdot n^{\beta})$, where α, β are constants such that $\alpha + \beta < 2$.*

A straightforward application of the notions of sub-cubic and sub-quadratic to the two-variable function mn would have resulted in a simpler but less powerful definition for sub-mn, namely that of requiring a time bound $O((mn)^{\delta})$, for some $\delta < 1$. Another weaker form of the above definition would have been to require $\alpha \leq 1$ and $\beta \leq 1$ with at least one of the two being strictly less than 1. The above definition is more general than either of these. It considers a bound of the form m^3/n^2 to be sub-mn even though such a bound for the graph problem with be larger than n^3 for dense graphs. Thus, it is a very strong definition of sub-mn when applied to hardness results. (Such a definition could be abused when giving sub-mn reductions, but all of our sub-mn reductions are linear or near-linear in the sizes of input and output, and thus readily satisfy the sub-mn definition while being very efficient.)

We say that a problem is MWC-hard if there is a sub-mn reduction from Minimum Weight Cycle to this problem. Our fine-grained reductions are based on this MWC hardness assumption, i.e. there is no sub-mn time algorithm for MWC (Minimum Weight Cycle).

Definition 2.3.6 (**MWC Hardness**). *A problem Q is said to be MWC-Hard if there is a sub-mn reduction from MWC (Minimum Weight Cycle) to Q.*

The following observation follows from this definition.

Observation 2.3.7. *If a problem Q is MWC-Hard, then there is no sub-mn time algorithm for Q unless MWC (Minimum Weight Cycle) can be computed in sub-mn time.*

Directed MWC is a natural candidate for hardness for the mn class since it is a fundamental problem for which a simple $\tilde{O}(mn)$ time algorithm has been known for many decades, and very recently, an $O(mn)$ time algorithm [71]. But a sub-mn time

algorithm remains as elusive as ever. Further, through the fine-grained reductions that we came up with in this work, many other problems in the mn class have MWC hardness for sub-mn time as noted in the following theorem.

Theorem 2.3.8. *The following problems on directed graphs do not have sub-mn time algorithms: 2-SiSP, 2-SiSC, s-t Replacement Paths, ANSC, Radius, BC, and Eccentricities assuming that MWC cannot be computed in sub-mn time.*

In [66], Lincoln et al. showed that MWC is hard under the Min-Wt-k-Clique Conjecture. This result in conjunction with Theorem 2.3.8 shows that the problems in the $\tilde{O}(mn)$ class including 2-SiSP, 2-SiSC, Replacement Paths, Radius, BC, Eccentricities and ANSC are also Min-Wt-k-Clique hard.

We now discuss the SETH and k-DSH hardness of Eccentricities and BC, which are also MWC and Min-Wt-k-Clique hard.

SETH and k-DSH hardness for Diameter, Eccentricities and BC. Another fundamental graph problem in the $\tilde{O}(mn)$ class is Diameter, which has a trivial sub-mn reduction to Eccentricities. Even though Diameter is in the $\tilde{O}(mn)$ class, we do not have an MWC-hardness result for it (nor is a sub-cubic reduction from MWC known). However computing Diameter in sub-m^2 time in graphs with $m = O(n)$ edges was shown to be SETH-hard in [78] for both directed and undirected graphs. The *Strong Exponential Time Hypothesis (SETH)* [51] states that that for every $\delta < 1$ there exists a k such that there is no $2^{\delta \cdot n}$ time algorithm for k-SAT. On the other hand, a SETH-based conditional hardness is not known for either MWC or APSP. The construction in [78] that gives sub-m^2 hardness for Diameter on very sparse graphs also gives a sub-mn hardness under SETH listed below.

Theorem 2.3.9 (**Sub-mn Hardness Under SETH**). *Under SETH, Eccentricities does not have a sub-mn time algorithm in an unweighted or weighted graph, either directed or undirected.*

The k *Dominating Set Hypothesis (k-DSH)* [72] states that there exists k_0 such that for all $k \geq k_0$, a dominating set of size k in an undirected graph on n vertices cannot be found in $O(n^{k-\epsilon})$ for any constant $\epsilon > 0$. It is known that k-DSH hardness implies SETH-hardness, and a sub-m^2 hardness result for Diameter under k-DSH *for even values of* k was shown in [78] for graphs with $\tilde{O}(n)$ edges. We show that Diameter is k-DSH hard for sub-mn time for all values of k, both odd and even, thus strengthening the SETH and k-DSH hardness results for Diameter and Eccentricities.

Eccentricities as a Central Problem for mn. We observe that directed Eccentricities is a central problem in the mn class: If a sub-mn time algorithm is obtained for directed Eccentricities, not only would it refute k-DSH, SETH and MWC hardness (Definition 2.3.6) it would also imply sub-mn time algorithms for several MWC-hard problems: 2-SiSP, 2-SiSC, s-t Replacement Paths, ANSC and Radius in directed graphs as well as Radius and Eccentricities in undirected graphs.

APSP. APSP has a special status in the mn class. Since its output size is n^2, it has near-optimal algorithms [85, 42, 76, 77] for graphs with $m = O(n)$. Also, the n^2 size for the APSP output means that any inference made through sparse reductions to APSP will not be based on a sub-mn time bound but instead on a sub-$mn+n^2$ time bound. It also turns out that the SETH and k-DSH hardness results for Diameter depend crucially on staying with a purely sub-mn bound, and hence even though Diameter has a simple sparse n^2 reduction to APSP, we do not have SETH or k-DSH hardness for computing APSP in sub-$mn + n^2$ time.

Betweenness Centrality (BC). We discuss this problem in Section 2.6. Sparse reductions for several variants of BC were given in [2] but none established MWC-hardness. We give nontrivial sparse reductions to establish MWC hardness for some important variants of BC. Our results show that BC and Eccentricities are key problems in the mn class that are MWC-hard and SETH/k-DSH hard while being

sub-cubic equivalent to APSP. Fig. 2.8 in Section 2.6 is an enhancement of Fig. 2.1 that includes our results for BC variants.

Separation of Time Bounds for Sparse Graphs. It is readily seen that the $\tilde{O}(m^{3/2})$ bound for triangle finding problems is a better bound than the $\tilde{O}(mn)$ bound for the mn class. But imposing a total ordering on functions of two variables requires some care. For example, maximal 2-connected subgraphs of a given directed graph can be computed in $O(m^{3/2})$ time [24] as well as in $O(n^2)$ time [43]. with $m^{3/2}$ a better bound for very sparse graphs and n^2 for very dense graphs, In Section 2.8 we motivate a natural definition of what it means for for one time bound to be smaller than another time bound for sparse graphs. By our definitions, $m^{3/2}$ is a smaller time bound than both mn and n^2 for sparse graphs. Our definitions establish that the problems related to triangle listing must have *provably smaller* time bounds for sparse graphs than the mn class under the hardness conjectures and fine-grained reductions for the MWC class.

Overview of the Chapter. Sections 2.4 and 2.5 present our sparse reductions for undirected and directed graphs. Section 2.6 present our sparse reductions for centrality problems in directed graphs. In Sections 2.7 and 2.8 we present SETH and k-DSH hardness results, and the resulting provable split of the sub-cubic equivalence class under these hardness results for sparse time bounds.

2.4 Weighted Undirected Graphs

2.4.1 Reducing MWC to APSD

In undirected graphs, the only known sub-cubic reduction from MWC to APSD [79] uses a dense reduction to Min-Wt-Δ. Described in [79] for integer edge weights of value at most W_{max}, it first uses an algorithm in [67] to compute, in $O(n^2 \cdot \log n \log nW_{max})$ time, a 2-approximation W to the weight of a minimum weight

cycle as well as shortest paths between all pairs of vertices with pathlength at most $W/2$. The reduced graph for Min-Wt-Δ is constructed as a (dense) bipartite graph with edges to represent all of these shortest paths, together with the edges of the original graph in one side of the bipartition. This results in each triangle in the reduced graph corresponding to a cycle in the original graph, and with a minimum weight cycle guaranteed to be present as a triangle. An MWC is then constructed using a version of Color Coding [13] with 2 colors.

The approach in [79] does not work in our case, as we are dealing with sparse reductions. Instead, we give a sparse reduction directly from MWC to APSD. In contrast to [79], where finding a minimum weight 3-edge triangle gives the MWC in the original graph, in our reduction the MWC is constructed as a path P in a reduced graph followed by a shortest path in the original graph.

One may ask if we can sparsify the dense reduction from MWC to Min-Wt-Δ in [79] but such a reduction, though very desirable, would immediately refute MWCC and would achieve a major breakthrough by giving an $\tilde{O}(m^{3/2})$ time algorithm for undirected MWC.

We now sketch our sparse reduction from undirected MWC to APSP. We start with stating from [79] the notion of a 'critical edge' and then present some additional properties we will use.

Lemma 2.4.1 ([79])**.** *Let $G = (V, E, w)$ be a weighted undirected graph, where $w : E \to \mathcal{R}^+$, and let $C = \langle v_1, v_2, \ldots, v_l \rangle$ be a cycle in G. There exists an edge (v_i, v_{i+1}) on C such that $\lceil \frac{w(C)}{2} \rceil - w(v_i, v_{i+1}) \leq d_C(v_1, v_i) \leq \lfloor \frac{w(C)}{2} \rfloor$ and $\lceil \frac{w(C)}{2} \rceil - w(v_i, v_{i+1}) \leq d_C(v_{i+1}, v_1) \leq \lfloor \frac{w(C)}{2} \rfloor$. The edge (v_i, v_{i+1}) is called the* critical edge *of C with respect to the start vertex v_1.*

Lemma 2.4.2. *Let C be a minimum weight cycle in weighted undirected graph G. Let x and y be two vertices on C and let $\pi^1_{x,y}$ and $\pi^2_{x,y}$ be the paths from x to y in C. W.l.o.g. assume that $w(\pi^1_{x,y}) \leq w(\pi^2_{x,y})$. Then $\pi^1_{x,y}$ is a shortest path from x to*

y and $\pi^2_{x,y}$ is a second simple shortest path from x to y, i.e. a path from x to y that is shortest among all paths from x to y that are not identical to $\pi^1_{x,y}$.

Proof. Assume to the contrary that $\pi^3_{x,y}$ is a second simple shortest path from x to y of weight less than $w(\pi^2_{x,y})$. Let $\pi^3_{x,y}$ deviate from $\pi^1_{x,y}$ at vertex u and then merge back at vertex v. Then the subpaths from u to v in $\pi^1_{x,y}$ and $\pi^3_{x,y}$ together form a cycle of weight strictly less than $w(C)$, resulting in a contradiction as C is a minimum weight cycle in G. □

Observation 2.4.3. *Let $G = (V, E, w)$ be a weighted undirected graph. Let $C = \langle v_1, v_2, \ldots, v_l \rangle$ be a minimum weight cycle in G, and let (v_p, v_{p+1}) be its critical edge with respect to v_1. W.l.o.g. assume that $d_G(v_1, v_p) \geq d_G(v_1, v_{p+1})$. If G' is obtained by removing edge (v_{p-1}, v_p) from G, then the path $P = \langle v_1, v_l, \ldots, v_{p+1}, v_p \rangle$ is a shortest path from v_1 to v_p in G'.*

Observation 2.4.3 holds since the path P there must be either a shortest path or a second simple shortest path in G by the above lemma, so in G' it must be a shortest path.

In our reduction we construct a collection of graphs $G_{i,j,k}$, each with $2n$ vertices (containing 2 copies of V) and $O(m)$ edges, with the guarantee that, for the minimum weight cycle C, in at least one of the graphs the edge (v_{p-1}, v_p) (in Observation 2.4.3) will not connect across the two copies of V and the path P of Observation 2.4.3 will be present. Then, if a call to APSP computes P as a shortest path from v_1 to v_p (across the two copies of V), we can verify that edge (v_p, v_{p+1}) is not the last edge on the computed shortest path from v_1 to v_p in G, and so we can form the concatenation of these two paths as a possible candidate for a minimum weight cycle. The challenge is to construct a small collection of graphs where we can ensure that the path we identify in one of the derived graphs is in fact the simple path P in the input graph. We overcome this challenge by using our new

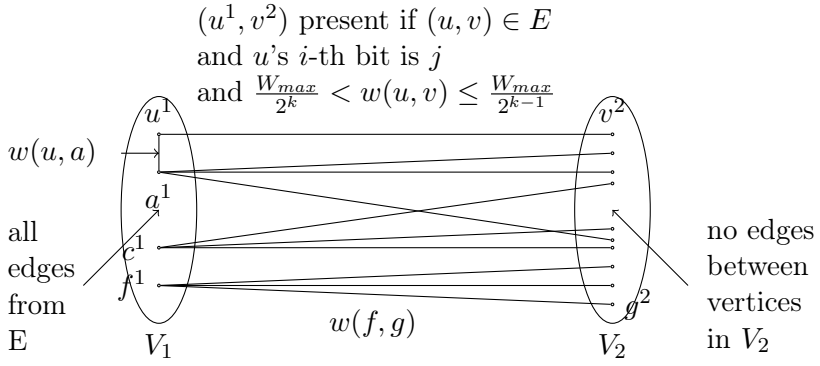


Figure 2.2: Construction of $G_{i,j,k}$.

bit-sampling technique which allows us to selectively sample edges from G to place in these constructed graphs $G_{i,j,k}$ such that P is guaranteed to be identified in one of these graphs.

Each $G_{i,j,k}$ has two copies of each vertex $u \in V$, $u^1 \in V_1$ and $u^2 \in V_2$. All edges in G are present on the vertex set V_1, but there is no edge that connects any pair of vertices within V_2. In $G_{i,j,k}$ there is an edge from $u^1 \in V_1$ to $v^2 \in V_2$ iff there is an edge from u to v in G, and u's i-th bit is j, and $\frac{W_{max}}{2^k} < w(u, v) \leq \frac{W_{max}}{2^{k-1}}$. All the edges in $G_{i,j,k}$ retain their weights from G. Thus, the edge (u^1, v^2) is present in $G_{i,j,k}$ with weight w only if (u, v) is an edge in G with the same weight w and further, certain conditions (as described above) hold for the indices i, j, k. This is our bit-sampling method. Here, $1 \leq i \leq \lceil \log n \rceil$, $j \in \{0, 1\}$ and $k \in \{1, 2, \ldots, \lceil \log \rho \rceil\}$, so we have $2 \cdot \log n \cdot \log \rho$ graphs. Figure 2.2 depicts the construction of graph $G_{i,j,k}$.

The first condition for an edge (u^1, v^2) to be present in $G_{i,j,k}$ is that u's i-th bit must be j. This ensures that there exist a graph where the edge (v^1_{p-1}, v^2_p) absent and the edge (v^1_{p+1}, v^2_p) is present (as v_{p-1} and v_{p+1} differ on at least 1 bit). To contrast with a similar step in [79], we need to find the path P in the sparse derived graph, while in [79] it suffices to look for the 2-edge path that represents P in a triangle in their dense reduced graph.

The second condition — that an edge (u^1, v^2) is present only if $\frac{W_{max}}{2^k} < w(u,v) \leq \frac{W_{max}}{2^{k-1}}$ — ensures that there is a graph $G_{i,j,k}$ in which, not only is edge (v^1_{p+1}, v^2_p) present and edge (v^1_{p-1}, v^2_p) absent as noted by the first condition, but also the shortest path from v^1_1 to v^2_p is in fact the path P in Observation 2.4.3, and does not correspond to a false path where an edge in G is traversed twice. In particular, we show that this second condition allows us to exclude a shortest path from v^1_1 to v^2_p of the following form: take the shortest path from v_1 to v_p in G on vertices in V_1, then take an edge (v^1_p, x^1), and then the edge (x^1, v^2_p). Such a path, which has weight $d_C(v_1, v_p) + 2w(x, v_p)$, could be shorter than the desired path, which has weight $d_C(v_1, v_{p+1}) + w(v_{p+1}, v_p)$. In our reduction we avoid selecting this ineligible path by requiring that the weight of the selected path should not exceed $d_G(v_1, v_p)$ by more than $W_{max}/2^{k-1}$. We show that these conditions suffice to ensure that P is identified in one of the $G_{i,j,k}$, and no spurious path of shorter length is identified. Notice that, in contrast to [79], we do not estimate the MWC weight by computing a 2-approximation. Instead, this second condition allows us to identify the critical edge in the appropriate graph.

MWC-to-APSP(G); ▷ this gives a simpler $\tilde{O}(W_{max} \cdot n^\omega)$ time algorithm for MWC with small integer weights than [79]

1: $wt \leftarrow \infty$
2: **for** $1 \leq i \leq \lceil \log n \rceil$, $j \in \{0, 1\}$, and $1 \leq k \leq \lceil \log \rho \rceil$ **do**
3: Compute APSP on $G_{i,j,k}$
4: **for** $y, z \in V$ **do**
5: **if** $d_{G_{i,j,k}}(y^1, z^2) \leq d_G(y, z) + \frac{W_{max}}{2^{k-1}}$ **then**
6: Check if $Last_{G_{i,j,k}}(y^1, z^2) \neq Last_G(y, z)$
7: **if** both checks in Steps 5-6 hold **then**
8: $wt \leftarrow min(wt, d_{G_{i,j,k}}(y^1, z^2) + d_G(y, z))$
9: **return** wt

In the following two lemmas we identify three key properties of a path π from y^1 to z^2 ($y \neq z$) in a $G_{i,j,k}$ that (I) will be satisfied by the path P in Observation 2.4.3

for $y^1 = v_1^1$ and $z^2 = v_p^2$ in some $G_{i,j,k}$ (Lemma 2.4.4), and (II) will cause a simple cycle in G to be contained in the concatenation of π with the shortest path from y to z computed by APSP (Lemma 2.4.5). Once we have these two Lemmas in hand, it gives us a method to find a minimum weight cycle in G (described in Algorithm MWC-to-APSP) by calling APSP on each $G_{i,j,k}$ and then identifying all pairs y^1, z^2 in each graph that satisfy these properties. Since the path P is guaranteed to be one of the pairs, and no spurious path will be identified, the minimum weight cycle can be identified. We now fill in the details.

Lemma 2.4.4. *Let $C = \langle v_1, v_2, \ldots, v_l \rangle$ be a minimum weight cycle in G and let (v_p, v_{p+1}) be its critical edge with respect to the start vertex v_1. W.l.o.g. assume that $d_G(v_1, v_p) \geq d_G(v_1, v_{p+1})$. Then there exists an $i \in \{1, \ldots, \lceil \log n \rceil\}$, $j \in \{0, 1\}$ and $k \in \{1, 2, \ldots, \lceil \log \rho \rceil\}$ such that the following conditions hold:*

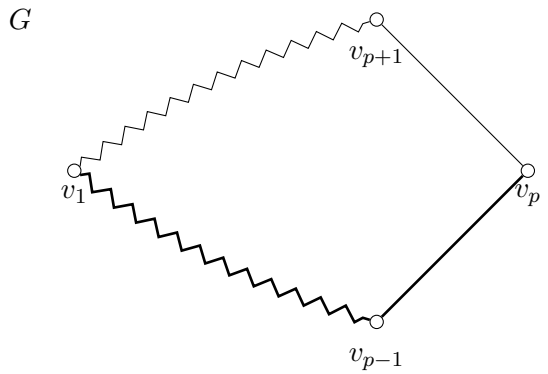
(i) $d_{G_{i,j,k}}(v_1^1, v_p^2) + d_G(v_1, v_p) = w(C)$

(ii) $Last_{G_{i,j,k}}(v_1^1, v_p^2) \neq Last_G(v_1, v_p)$

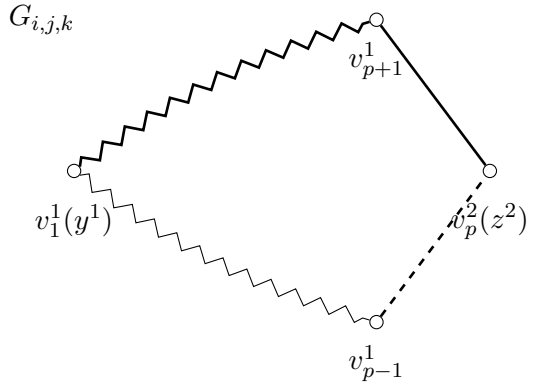
(iii) $d_{G_{i,j,k}}(v_1^1, v_p^2) \leq d_G(v_1, v_p) + \frac{W_{max}}{2^{k-1}}$

Proof. Let i, j and k be such that: v_{p-1} and v_{p+1} differ on i-th bit and j be the i-th bit of v_{p+1} and k be such that $\frac{W_{max}}{2^k} < w(v_p, v_{p+1}) \leq \frac{W_{max}}{2^{k-1}}$. Hence, edge (v_{p-1}^1, v_p^2) is not present and the edge (v_{p+1}^1, v_p^2) is present in $G_{i,j,k}$ and so $Last_{G_{i,j,k}}(v_1^1, v_p^2) \neq Last_G(v_1, v_p)$, satisfying part 2 of the lemma.

Let us map the path P in Observation 2.4.3 to the path P' in $G_{i,j,k}$, such that all vertices except v_p are mapped to V_1 and v_p is mapped to V_2 (bold path from v_1^1 to v_p^2 in Figure 2.3b). Then, if P' is a shortest path from v_1^1 to v_p^2 in $G_{i,j,k}$, both parts 1 and 3 of the lemma will hold. So it remains to show that P' is a shortest path. But if not, an actual shortest path from v_1^1 to v_p^2 in $G_{i,j,k}$ would create a shorter cycle in G than C, and if that cycle were not simple, one could extract from it an even shorter cycle, contradicting the fact that C is a minimum weight cycle in G. □



(a) MWC C in G. Shortest path from v_1 to v_p is highlighted.



- i, j, k are such that:
  1. i-th bit of $v_{p+1}(v_{p-1})$ is $j(\bar{j})$
  2. $W_{max}/2^k < w(v_p, v_{p+1}) \leq W_{max}/2^{k-1}$.
- Alg. MWC-to-APSP' will compute $w(C)$ in this $G_{i,j,k}$ in wt (line 6)

(b) Shortest path from y^1 to z^2 in $G_{i,j,k}$.

Figure 2.3: Here Figure (a) represent the MWC C in G. The path $\pi_{y,z}$ (in bold) is the shortest path from y to z in G. The path π_{y^1,z^2} (in bold) in Figure (b) is the shortest path from y^1 to z^2 in $G_{i,j,k}$: where the edge (v^1_{p-1}, v^2_p) is absent due to i, j bits. The paths $\pi_{y,z}$ in G and π_{y^1,z^2} in $G_{i,j,k}$ together comprise the MWC C.

Lemma 2.4.5. *If there exists* $i \in \{1, \ldots, \lceil \log n \rceil\}$, $j \in \{0, 1\}$, $k \in \{1, 2, \ldots, \lceil \log \rho \rceil\}$, *and* $y, z \in V$ *such that the following conditions hold:*

(i) $d_{G_{i,j,k}}(y^1, z^2) + d_G(y, z) = wt$ *for some* wt

(ii) $Last_{G_{i,j,k}}(y^1, z^2) \neq Last_G(y, z)$

(iii) $d_{G_{i,j,k}}(y^1, z^2) \leq d_G(y, z) + \frac{W_{max}}{2^{k-1}}$

Then G *has a simple cycle of weight at most* wt *that contains* z.

Proof. Let $\pi_{y,z}$ be a shortest path from y to z in G (see Figure 2.3a) and let π_{y^1,z^2} be a shortest path from y^1 to z^2 in $G_{i,j,k}$ (Figure 2.3b). Let $\pi'_{y,z}$ be the path corresponding to π_{y^1,z^2} in G.

Now we need to show that the path $\pi'_{y,z}$ is simple. Assume that $\pi'_{y,z}$ is not simple. It implies that the path π_{y^1,z^2} must contain x^1 and x^2 for some $x \in V$. Now if $x \neq z$, then we can remove the subpath from x^1 to x^2 (or from x^2 to x^1) to obtain an even shorter path from y^1 to z^2.

It implies that the path π_{y^1,z^2} contains z^1 as an internal vertex. Let π_{z^1,z^2} be the subpath of π_{y^1,z^2} from vertex z^1 to z^2. If π_{z^1,z^2} contains at least 2 internal vertices then this would be a simple cycle of weight less than wt, and we are done. Otherwise, the path π_{z^1,z^2} contains exactly one internal vertex (say x^1). Hence path π_{z^1,z^2} corresponds to the edge (z, x) traversed twice in graph G. But the weight of the edge (x, z) must be greater than $\frac{W_{max}}{2^k}$ (as the edge (x^1, z^2) is present in $G_{i,j,k}$). Hence $w(\pi_{z^1,z^2}) > \frac{W_{max}}{2^{k-1}}$ and hence $d_{G_{i,j,k}}(y^1, z^2) \geq d_G(y, z) + w(\pi_{z^1,z^2}) > d_G(y, z) + \frac{W_{max}}{2^{k-1}}$, resulting in a contradiction as condition 3 states otherwise. (It is for this property that the index k in $G_{i,j,k}$ is used.) Thus path π_{y^1,z^2} does not contain z^1 as an internal vertex and hence $\pi'_{y,z}$ is simple.

If the paths $\pi_{y,z}$ and $\pi'_{y,z}$ do not have any internal vertices in common, then $\pi_{y,z} \circ \pi'_{y,z}$ corresponds to a simple cycle C in G of weight wt that passes through y

and z. Otherwise, we can extract from $\pi_{y,z} \circ \pi'_{y,z}$ a cycle of weight smaller than wt. This establishes the lemma. □

Proof of Theorem 2.3.2: To compute the weight of a minimum weight cycle in G in $\tilde{O}(n^2 + T_{APSP})$, we use procedure MWC-to-APSP. By Lemmas 2.4.4 and 2.4.5, the value wt returned by this algorithm is the weight of a minimum weight cycle in G. □

An $\tilde{O}(W_{max} \cdot n^\omega)$ time algorithm for MWC with small integer weights is given in [79]. By using the $\tilde{O}(W_{max}.n^\omega)$ algorithm for APSP with small integer edge weights [83], Alg. MWC-to-APSP gives a simpler $\tilde{O}(W_{max} \cdot n^\omega)$ time algorithm for this problem.

Sparse Reduction to APSD: We now describe how to avoid using the $Last$ matrix in the reduction. A 2-approximation algorithm for finding a cycle of weight at most $2t$, where t is such that the minimum-weight cycle's weight lies in the range $(t, 2t]$, as well as distances between pairs of vertices within distance at most t, was given by Lingas and Lundell [67]. This algorithm can also compute the last edge on each shortest path it computes, and its running time is $\tilde{O}(n^2 \log(n\rho))$. For a minimum weight cycle $C = \langle v_1, v_2, \ldots, v_l \rangle$ where the edge (v_p, v_{p+1}) is a critical edge with respect to the start vertex v_1, the shortest path length from v_1 to v_p or to v_{p+1} is at most t. Thus using this algorithm, we can compute the last edge on a shortest path for such pair of vertices in $\tilde{O}(n^2 \log(n\rho))$ time.

In our reduction to APSD, we first run the 2-approximation algorithm on the input graph G to obtain the $Last(y, z)$ for certain pairs of vertices. Then, in Step 5 we check if $Last_{G_{i,j,k}}(y^1, z^2) \neq Last_G(y, z)$ *only if* $Last_G(y, z)$ has been computed (otherwise the current path is not a candidate for computing a minimum weight cycle). It appears from the algorithm that the $Last$ values are also needed in the $G_{i,j,k}$. However, instead of computing the $Last$ values in each $G_{i,j,k}$, we check for the

shortest path from y to z only in those $G_{i,j,k}$ graphs where the $Last_G(y,z)$ has been computed, and the edge is not present in $G_{i,j,k}$. In other words, if $Last(y,z) = q$, we will only consider the shortest paths from y^1 to z^2 in those graphs $G_{i,j,k}$ where q's i-th bit is not equal to j. Thus our reduction to APSD goes through without needing APSP to output the *Last* matrix. This gives rise to an improved algorithm for MWC with small integer weights.

2.4.2 Reducing ANSC to APSP in Unweighted Undirected Graphs

For our sparse $\tilde{O}(n^2)$ reduction from ANSC to APSP in unweighted undirected graphs, we use the graphs from the previous section, but we do not use the index k, since the graph is unweighted.

Our reduction exploits the fact that in unweighted graphs, every edge in a cycle is a critical edge with respect to some vertex. Thus we construct $2\lceil \log n \rceil$ graphs $G_{i,j}$, and in order to construct a shortest cycle through vertex z in G, we will set $z = v_p^2$ in the reduction in the previous section. Then, by letting one of the two edges incident on z in the shortest cycle through z be the critical edge for the cycle, the construction from the previous section will allow us to find the length of a minimum length cycle through z, for each $z \in V$, with the post-processing algorithm ANSC-to-APSP.

ANSC-to-APSP

1: **for** each vertex $z \in V$ **do** $wt[z] \leftarrow \infty$
2: **for** $1 \leq i \leq \lceil \log n \rceil$, $j \in \{0,1\}$ **do**
3: Compute APSP′ on $G_{i,j,}$
4: **for** $y, z \in V$ **do**
5: **if** $d_{G_{i,j,}}(y^1, z^2) \leq d_G(y,z) + 1$ **then**
6: Check if $Last_{G_{i,j,}}(y^1, z^2) \neq Last_G(y,z)$
7: **if** both checks in Steps 5-6 hold **then**
8: $wt[z] \leftarrow min(wt[z], d_{G_{i,j,}}(y^1, z^2) + d_G(y,z))$
9: **return** wt array

Correctness of the above sparse reduction follows from the following two lemmas, which are similar to Lemmas 2.4.4 and 2.4.5.

Lemma 2.4.6. *Let $C = \langle z, v_2, v_3, \ldots, v_q \rangle$ be a minimum length cycle passing through vertex $z \in V$. Let (v_p, v_{p+1}) be its critical edge such that $p = \lfloor \frac{q}{2} \rfloor + 1$. Then there exists an $i \in \{1, \ldots, \lceil \log n \rceil\}$ and $j \in \{0, 1\}$ such that the following conditions hold:*

(i) $d_{G_{i,j}}(v_p^1, z^2) + d_G(v_p, z) = len(C)$

(ii) $Last_{G_{i,j}}(v_p^1, z^2) \neq Last_G(v_p, z)$

(iii) $d_{G_{i,j}}(v_p^1, z^2) \leq d_G(v_p, z) + 1$

Lemma 2.4.7. *If there exists an $i \in \{1, \ldots, \lceil \log n \rceil\}$ and $j \in \{0, 1\}$ and $y, z \in V$ such that the following conditions hold:*

(i) $d_{G_{i,j}}(y^1, z^2) + d_G(y, z) = q$ *for some* q *where* $d_G(y, z) = \lfloor \frac{q}{2} \rfloor$

(ii) $Last_{G_{i,j}}(y^1, z^2) \neq Last_G(y, z)$

(iii) $d_{G_{i,j}}(y^1, z^2) \leq d_G(y, z) + 1$

Then there exists a simple cycle C passing through z of length at most q in G.

Proof of Theorem 2.3.3: We now show that the entries in the wt array returned by the above algorithm correspond to the ANSC output for G. Let $z \in V$ be an arbitrary vertex in G and let $q = wt[z]$. Let y' be the vertex in Step 5 for which we obtain this value of q. Hence by Lemma 2.4.7, there exists a simple cycle C passing through z of length at most q in G. If there were a cycle through z of length $q' < q$ then by Lemma 2.4.6, there exists a vertex y'' such that conditions in Step 5 hold for q', and the algorithm would have returned a smaller value than $wt[z]$, which is a contradiction. This is a sparse $\tilde{O}(n^2)$ reduction since it makes $O(\log n)$ calls to APSP, and spends $\tilde{O}(n^2)$ additional time. □

It would be interesting to see if we can obtain a reduction from *weighted* ANSC to APSD or APSP. The above reduction does not work for the weighted case since it exploits the fact that for any cycle C through a vertex z, an edge in C that is incident on z is a critical edge for some vertex in C. However, this property need not hold in the weighted case.

2.4.3 Bit-Sampling

We use the bit-sampling technique in our reductions for undirected graphs: from weighted MWC to APSP (Section 2.4.1), unweighted ANSC to unweighted APSP (Section 2.4.2) and from weighted k-SiSC to k-SiSP (Section 2.9.1). This technique is crucial to all of these reductions. Using this technique, we obtain a new near-linear time algorithm for undirected k-SiSC, a new $\tilde{O}(n^\omega)$ algorithm for unweighted undirected ANSC and a simpler $\tilde{O}(W_{max} \cdot n^\omega)$ algorithm for weighted MWC. Here we describe how this technique is different from the 'bit-encoding' technique in [2] and how it gives an explicit construction for Color Coding for 2 colors.

2.4.3.1 Bit-sampling and Color Coding

Color Coding is a method introduced by Alon, Yuster and Zwick [13]. For the special case of 2 colors, the method constructs a collection C of $O(\log n)$ different 2-coloring on an n-element set V, such that for every pair $\{x, y\}$ in V, there is a 2-coloring in C that assigns different colors to x and y. When the elements of V have unique $\log n$-bit labels, e.g., by numbering them from 0 to $n-1$, our bit-sampling method on index i (ignoring indices j and k) can be viewed as an explicit construction of exactly $\lceil \log n \rceil$ hash functions for the 2-perfect hash family: the i-th hash function assigns to each element the i-th bit in its label as its color.

In our construction we actually use $2 \log n$ functions (using both i and j) since we need a stronger version of color coding where, for any pair of vertices x,

y, there is a hash function that assigns color 0 to x and 1 to y and another that assigns 1 to x and 0 to y. This is needed in order to ensure that when $x = v_{p-1}$ and $y = v_{p+1}$, the edge (v^1_{p-1}, v^2_p) is absent and the edge (v^1_{p+1}, v^2_p) is present. A different variant of Color Coding with 2 colors is used in [79] in their dense reduction from undirected MWC to Min-Wt-Δ, and we do not immediately see how to apply our bit-sampling technique there.

Our bit-sampling method differs from a 'bit-encoding' technique used in some reductions in [2, 1], where the objective is to preserve sparsity in the constructed graph while also preserving paths from the original graph $G = (V, E)$. This technique creates paths between two copies of V by adding $\Theta(\log n)$ new vertices with $O(\log n)$ bit labels, and using the $O(\log n)$ bit labels on these new vertices to induce the desired paths in the constructed graph. The bit-encoding technique (from [2]) is useful for certain types of reductions, and we use it in our sparse reduction from 2-SiSP to Radius in Section 2.5, and from 2-SiSP to BC in Section 2.6.

The bit-sampling technique we use in our reduction here is different from this bit-encoding technique. Here the objective is to selectively sample the edges from the original graph to be placed in the reduced graph, based on the bit-pattern of the end points and the edge weight. In our construction, we create $\Theta(\log n)$ different graphs, where in each graph the copies of V are connected by single-edge paths, without requiring additional intermediate vertices.

In Section 2.9.1, we give another application of our *bit-sampling technique* to obtain a new near-linear time algorithm for k-SiSC in undirected graphs (see definition in Section 2.2). Note that this problem is not in the mn class and this result is relevant here as an application of our new bit-sampling technique.

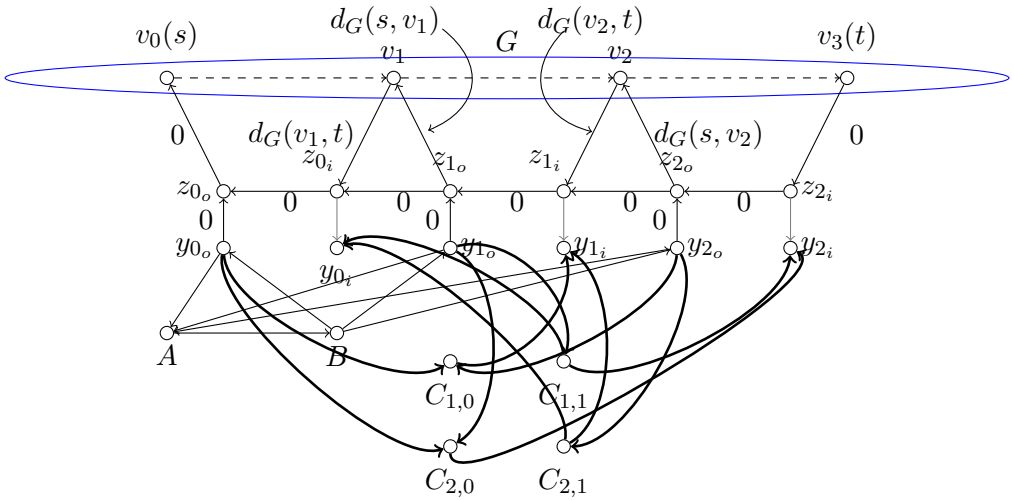


Figure 2.4: G'' for $l = 3$. The gray and the bold edges have weight $\frac{11}{9}M'$ and $\frac{1}{3}M'$ respectively. All the outgoing (incoming) edges from (to) A have weight 0 and the outgoing edges from B have weight M'.

2.5 Weighted Directed Graphs

2.5.1 Reducing 2-SiSP to Radius and s-t Replacement Paths to Eccentricities

A sparse $O(n^2)$ reduction from 2-SiSP to APSP was given in [41]. Our sparse reduction from 2-SiSP to Radius refines this result and the sub-mn partial order by plugging the Radius and Eccentricities problems within the sparse reduction chain from 2-SiSP to APSP. Also, in Section 2.5.2 we show MWC $\leq^{\textbf{\textit{sprs}}}_{m+n}$ 2-SiSP, thus establishing MWC-hardness for both 2-SiSP and Radius. Our 2-SiSP to Radius reduction here is unrelated to the sparse reduction in [41] from 2-SiSP to APSP.

The input is $G = (V, E, w)$, with source s and sink t in V, and a shortest path P ($s = v_0 \to v_1 \rightsquigarrow v_{l-1} \to v_l = t$). We need to compute a second simple s-t shortest path. Figure 2.4 gives an example of our reduction to an input G'' to the Radius problem for $l = 3$. Our reduction differs from a sub-cubic reduction from Min-Wt-Δ to Radius in [2] which transforms minimum weight triangle to Radius by

creating a 4-partite graph. However, since Min-Wt-Δ can be solved in $O(m^{3/2})$ time this is not relevant to us. Instead, we give a more complex reduction from 2-SiSP where the second shortest path can have $\Theta(n)$ edges and hence we cannot start with a k-partite graph for some constant k.

In G'' we first map every edge (v_j, v_{j+1}) lying on P to the vertices z_{j_o} and z_{j_i} such that the shortest path from z_{j_o} to z_{j_i} corresponds to the shortest path from s to t avoiding the edge (v_j, v_{j+1}). We then add vertices y_{j_o} and y_{j_i} in the graph and connect them to vertices z_{j_o} and z_{j_i} by adding edges (y_{j_o}, z_{j_o}) and (y_{j_i}, z_{j_i}), and then additional edges from y_{j_o} to other y_{k_o} and y_{k_i} vertices such that the longest shortest path from y_{j_o} is to the vertex y_{j_i}, which in turn corresponds to the shortest path from z_{j_o} to z_{j_i}. In order to preserve sparsity, we have an interconnection from each y_{j_o} vertex to all y_{k_i} vertices (except for $k = j$) with a sparse construction by using $2 \log n$ additional vertices $C_{r,s}$ in a manner similar to a bit-encoding technique used in [2] in their reduction from Min-Wt-Δ to Betweenness Centrality (this technique however, is different from the new 'bit-sampling' technique used in Section 2.4), and we have two additional vertices A, B with suitable edges to induce connectivity among the y_{j_o} vertices. In our construction, we ensure that the center is one of the y_{j_o} vertices and hence computing the Radius in the reduced graph gives the minimum among all the shortest paths from z_{j_o} to z_{j_i}. This corresponds to a shortest replacement path from s to t.

Lemma 2.5.1. *In weighted directed graphs, 2-SiSP $\leq^{\boldsymbol{sprs}}_{m+n}$ Radius and s-t Replacement Paths $\leq^{\boldsymbol{sprs}}_{m+n}$ Eccentricities*

Proof. We are given an input graph $G = (V, E)$, a source vertex s and a sink/target vertex t and we wish to compute the second simple shortest path from s to t. Let P ($s = v_0 \to v_1 \rightsquigarrow v_{l-1} \to v_l = t$) be the shortest path from s to t in G.

Constructing the reduced graph G'': We first create the graph G', which contain G and l additional vertices z_0,z_1,...,z_{l-1}. We remove the edges lying on P from G'.

For each $0 \leq i \leq l-1$, we add an edge from z_i to v_i of weight $d_G(s, v_i)$ and an edge from v_{i+1} to z_i of weight $d_G(v_{i+1}, t)$. Also for each $1 \leq i \leq l-1$, we add a zero weight edge from z_i to z_{i-1}.

Now form G'' from G'. For each $0 \leq j \leq l-1$, we replace vertex z_j by vertices z_{j_i} and z_{j_o} and we place a directed edge of weight 0 from z_{j_i} to z_{j_o}, and we also replace each incoming edge to (outgoing edge from) z_j with an incoming edge to z_{j_i} (outgoing edge from z_{j_o}) in G'.

Let W_{max} be the largest edge weight in G and let $M' = 9nW_{max}$. For each $0 \leq j \leq l-1$, we add additional vertices y_{j_i} and y_{j_o} and we place a directed edge of weight 0 from y_{j_o} to z_{j_o} and an edge of weight $\frac{11}{9}M'$ from z_{j_i} to y_{j_i}.

We add 2 additional vertices A and B, and we place a directed edge from A to B of weight 0. We also add l incoming edges to A (outgoing edges from B) from (to) each of the $y'_{j_o}s$ of weight 0 (M').

We also add edges of weight $\frac{2M'}{3}$ from y_{j_o} to y_{k_i} (for each $k \neq j$). But due to the addition of $O(n^2)$ edges, graph G' becomes dense. To solve this problem, we add a gadget in our construction that ensures that $\forall 0 \leq j \leq l-1$, we have at least one path of length 2 and weight equal to $\frac{2M'}{3}$ from y_{j_o} to y_{k_i} (for each $k \neq j$) (similar to [2]). In this gadget, we add $2\lceil \log n \rceil$ vertices of the form $C_{r,s}$ for $1 \leq r \leq \lceil \log n \rceil$ and $s \in \{0, 1\}$. Now for each $0 \leq j \leq l-1$, $1 \leq r \leq \lceil \log n \rceil$ and $s \in \{0, 1\}$, we add an edge of weight $\frac{M'}{3}$ from y_{j_o} to $C_{r,s}$ if $j's$ r-th bit is equal to s. We also add an edge of weight $\frac{M'}{3}$ from $C_{r,s}$ to y_{j_i} if j's r-th bit is not equal to s. So overall we add $2n \log n$ edges that are incident to $C_{r,s}$ vertices; for each y_{j_o} we add $\log n$ outgoing edges to $C_{r,s}$ vertices and for each y_{j_i} we add $\log n$ incoming edges from $C_{r,s}$ vertices.

We can observe that for $0 \leq j \leq l-1$, there is at least one path of weight $\frac{2M'}{3}$ from y_{j_o} to y_{k_i} (for each $k \neq j$) and the gadget does not add any new paths from y_{j_o} to y_{j_i}. The reason is that for every distinct j, k, there is at least one bit (say r) where j and k differ and let s be the r-th bit of j. Then there must be an

edge from y_{j_o} to $C_{r,s}$ and an edge from $C_{r,s}$ to y_{k_i}, resulting in a path of weight $\frac{2M'}{3}$ from y_{j_o} to y_{k_i}. And by the same argument we can also observe that this gadget does not add any new paths from y_{j_o} to y_{k_i}.

We call this graph as G''. Figure 2.4 depicts the full construction of G'' for $l = 3$. We now establish the following three properties.

(i) For each $0 \leq j \leq l-1$, the longest shortest path in G'' from y_{j_o} is to the vertex y_{j_i}. It is easy to see that the shortest path from y_{j_o} to any of the vertices in G or any of the z's has weight at most nW_{max}. And the shortest paths from y_{j_o} to the vertices A and B have weight 0. For $k \neq j$, the shortest path from y_{j_o} to y_{k_o} and y_{k_i} has weight M' and $\frac{2}{3}M'$ respectively. Whereas the shortest path from y_{j_o} to y_{j_i} has weight at least $10nW_{max}$ as it includes the last edge (z_{j_i}, y_{j_i}) of weight $\frac{11}{9}M' = 11nW_{max}$. It is easy to observe that the shortest path from y_{j_o} to y_{j_i} corresponds to the shortest path from z_{j_o} to z_{j_i}.

(ii) The shortest path from z_{j_o} to z_{j_i} corresponds to the replacement path for the edge (v_j, v_{j+1}) lying on P. Suppose not and let P_j $(s \rightsquigarrow v_h \rightsquigarrow v_k \rightsquigarrow t)$ (where v_h is the vertex where P_j separates from P and v_k is the vertex where it joins P) be the replacement path from s to t for the edge (v_j, v_{j+1}). But then the path π_j $(z_{j_o} \rightarrow z_{j-1_i} \rightsquigarrow z_{h_o} \rightarrow v_h \circ P_j(v_h, v_k) \circ v_k \rightarrow z_{k_i} \rightarrow z_{k_o} \rightsquigarrow z_{j_i})$ (where $P_j(v_h, v_k)$ is the subpath of P_j from v_j to v_k) from z_{j_o} to z_{j_i} has weight equal to $wt(P_j)$, resulting in a contradiction as the shortest path from z_{j_o} to z_{j_i} has weight greater than that of P_j.

(iii) One of the vertices among y_{j_o}'s is a center of G''. It is easy to see that none of the vertices in G could be a center of the graph G'' as there is no path from any $v \in V$ to any of the y_{j_o}'s in G''. Using a similar argument, we can observe that none of the z's, or the vertices y_{j_i}'s could be a potential candidate for the center of G''. For vertices A and B, the shortest path to any of the y_{j_i}'s has weight exactly $\frac{5}{3}M' = 15nW_{max}$, which is strictly greater than the weight of the largest shortest

path from any of the y_{j_o}'s. Thus one of the vertices among y_{j_o}'s is a center of G''.

Thus by computing the radius in G'', from (i), (ii), and (iii), we can compute the weight of the shortest replacement path from s to t, which by definition of 2-SiSP, is the second simple shortest path from s to t. This completes the proof of 2-SiSP $\lesssim^{\boldsymbol{sprs}}_{m+n}$ Radius.

Now, if instead of computing Radius in G'' we compute the Eccentricities of all vertices in G'', then from (i) and (ii) we can compute the weight of the replacement path for every edge (v_j, v_{j+1}) lying on P, thus solving the replacement paths problem. So s-t Replacement Paths $\lesssim^{\boldsymbol{sprs}}_{m+n}$ Eccentricities.

Constructing G'' takes $O(m + n \log n)$ time since we add $O(n)$ additional vertices and $O(m + n \log n)$ additional edges, and given the output of Radius (Eccentricities), we can compute 2-SiSP (s-t Replacement Paths) in $O(1)$ ($O(n)$) time and hence the cost of both reductions is $O(m + n \log n)$. □

2.5.2 Directed ANSC and Replacement Paths

We first describe a sparse reduction from directed MWC to 2-SiSP, which we will use for reducing ANSC to the s-t replacement paths problem. This reduction is adapted from a sub-cubic non-sparse reduction from Min-Wt-Δ to 2-SiSP in [91]. The reduction in [91] reduces Min-Wt-Δ to 2-SiSP by creating a tripartite graph. Since starting from Min-Wt-Δ is not appropriate for our results (as discussed in our sparse reduction to directed Radius), we start instead from MWC, and instead of the tripartite graph used in [91] we use the original graph G with every vertex v replaced with 2 copies, v_i and v_o.

In this reduction, as in [91], we first create a path of length n with vertices labeled from p_0 to p_n, which will be the initial shortest path. We then map every edge (p_i, p_{i+1}) to the vertex i in the original graph G such that the replacement path from p_0 to p_n for the edge (p_i, p_{i+1}) corresponds to the shortest cycle passing

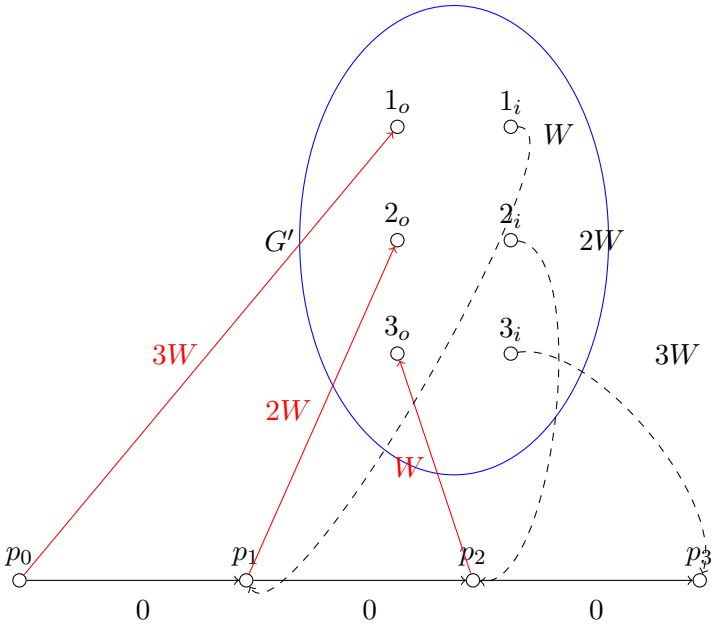


Figure 2.5: G'' for $n = 3$ in the reduction: MWC $\leq^{\textbf{sprs}}_{m+n}$ 2-SiSP

through i in G. Thus computing 2-SiSP (i.e., the shortest replacement path) from p_0 to p_n in the constructed graph corresponds to the minimum weight cycle in the original graph. Figure 2.5 gives an example of the constructed graph for $n = 3$.

Lemma 2.5.2. *In weighted directed graphs, MWC* $\leq^{\textbf{sprs}}_{m+n}$ *2-SiSP*

Proof. To compute MWC in G, we first create the graph G', where we replace every vertex z by vertices z_i and z_o, and we place a directed edge of weight 0 from z_i to z_o, and we replace each incoming edge to (outgoing edge from) z with an incoming edge to z_i (outgoing edge from z_o). We also add a path P $(p_0 \to p_1 \rightsquigarrow p_{n-1} \to p_n)$ of length n and weight 0.

Let $Q = n \cdot W_{max}$, where W_{max} is the maximum weight of any edge in G. For each $1 \leq j \leq n$, we add an edge of weight $(n - j + 1)Q$ from p_{j-1} to j_o and an edge of weight jQ from j_i to p_j in G' to form G''. Figure 2.5 depicts the full construction of G'' for $n = 3$. This is an $(m + n)$ reduction, and it can be seen that the second

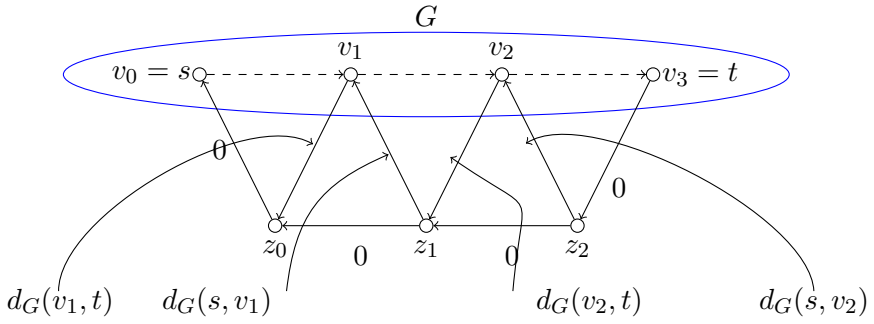


Figure 2.6: G' for $l = 3$ in the reduction: directed s-t Replacement Paths $\leq^{\boldsymbol{sprs}}_{m+n}$ ANSC

simple shortest path from p_0 to p_n in G'' corresponds to a minimum weight cycle in G. □

We now establish the equivalence between ANSC and the s-t replacement paths problem under $(m+n)$-reductions by first showing an $(m+n)$-sparse reduction from s-t replacement paths problem to ANSC. We then describe a sparse reduction from ANSC to the s-t replacement paths problem, which is similar to the reduction from MWC to 2-SiSP.

Lemma 2.5.3. *In weighted directed graphs, s-t replacement paths $\equiv^{\boldsymbol{sprs}}_{m+n}$ ANSC*

Proof. We are given an input graph $G = (V, E)$, a source vertex s and a sink vertex t and we wish to compute the replacement paths for all the edges lying on the shortest path from s to t. Let $P(s = v_0 \to v_1 \rightsquigarrow v_{l-1} \to v_l = t)$ be the shortest path from s to t in G.

(i) Constructing G': We first create the graph G', as described in the proof of Lemma 2.5.1. Figure 2.6 depicts the full construction of G' for $l = 3$.

(ii) We now show that for each $0 \leq i \leq l-1$, the replacement path from s to t for the edge (v_i, v_{i+1}) lying on P has weight equal to the shortest cycle passing through z_i. If not, assume that for some i $(0 \leq i \leq l-1)$, the weight of the replacement

path from s to t for the edge (v_i, v_{i+1}) is not equal to the weight of the shortest cycle passing through z_i.

Let P_i ($s \rightsquigarrow v_j \rightsquigarrow v_k \rightsquigarrow t$) (where v_j is the vertex where P_i separates from P and v_k is the vertex where it joins P) be the replacement path from s to t for the edge (v_i, v_{i+1}) and let C_i ($z_i \rightsquigarrow z_p \rightarrow v_p \rightsquigarrow v_q \rightarrow z_q \rightsquigarrow z_i$) be the shortest cycle passing through z_i in G'.

If $wt(P_i) < wt(C_i)$, then the cycle C_i' ($z_i \rightarrow z_{i-1} \rightsquigarrow z_j \rightarrow v_j \circ P_i(v_j, v_k) \circ v_k \rightarrow z_k \rightarrow z_{k-1} \rightsquigarrow z_i$) (where $P_i(v_j, v_k)$ is the subpath of P_i from v_j to v_k) passing through z_i has weight equal to $wt(P_i) < wt(C_i)$, resulting in a contradiction as C_i is the shortest cycle passing through z_i in G'.

Now if $wt(C_i) < wt(P_i)$, then the path P_i' ($s \rightsquigarrow v_p \circ C_i(v_p, v_q) \circ v_q \rightsquigarrow v_l$) where $C_i(v_p, v_q)$ is the subpath of C_i from v_p to v_q, is also a path from s to t avoiding the edge (v_i, v_{i+1}), and has weight equal to $wt(C_i) < wt(P_i)$, resulting in a contradiction as P_i is the shortest replacement path from s to t for the edge (v_i, v_{i+1}).

We then compute ANSC in G'. And by (ii), the shortest cycles for each of the vertices $z_0, z_1, \ldots, z_{l-1}$ gives us the replacement paths from s to t. This leads to an $(m + n)$ sparse reduction from s-t replacement paths problem to ANSC.

Now for the other direction, we are given an input graph $G = (V, E)$ and we wish to compute the ANSC in G. We first create the graph G'', as described in Lemma 2.5.2. We can see that the shortest path from p_0 to p_n avoiding edge (p_{j-1}, p_j) corresponds to a shortest cycle passing through j in G. This gives us an $(m + n)$-sparse reduction from ANSC to s-t replacement paths problem. □

2.6 Betweenness Centrality: Reductions

In this section, we consider sparse reductions for Betweenness Centrality and related problems. In its full generality, the Betweenness Centrality of a vertex v is the sum, across all pairs of vertices s, t, of the fraction of shortest paths from s to t that

contain v as an internal vertex. This problem has a $\tilde{O}(mn)$ time algorithm due to Brandes [21]. Since there can be an exponential (in n) number of shortest paths from one vertex to another, this general problem can deal with very large numbers. In [2], a simplified variant was considered, where it is assumed that there is a unique shortest path for each pair of vertices, and the Betweenness Centrality of vertex v, $BC(v)$, is defined as the number of vertex pairs s, t such that v is an internal vertex on the unique shortest path from s to t. We will also restrict our attention to this variant here.

A number of sparse reductions relating to the following problems were given in [2].

- *Betweenness Centrality (BC)* of a vertex v, $BC(v)$.
- *Positive Betweenness Centrality (Pos BC) of v:* determine whether $BC(v) > 0$.
- *All Nodes Betweenness Centrality (ANBC):* compute, for each v, the value of $BC(v)$.
- *Positive All Nodes Betweenness Centrality (Pos ANBC):* determine, for each v, whether $BC(v) > 0$.
- *Reach Centrality (RC)* of v: compute $\max_{s,t \in V: d_G(s,v)+d_G(v,t)=d_G(s,t)} \min(d_G(s,v), d_G(v,t))$.

Figure 2.7 gives an overview of the previous fine-grained results given in [2] for Centrality problems. In this figure, BC is the only centrality problem that is known to be sub-cubic equivalent to APSP, and hence is shaded in the figure (along with Min-Wt-Δ). None of these sparse reductions in [2] imply MWC hardness for any of the centrality problems since Diameter is not MWC-hard (or even sub-cubic equivalent to APSP), and Min-Wt-Δ has an $\tilde{O}(m^{3/2})$ time algorithm, and so will give a sub-mn algorithm for MWC if it is MWC-hard. On other hand, Diameter

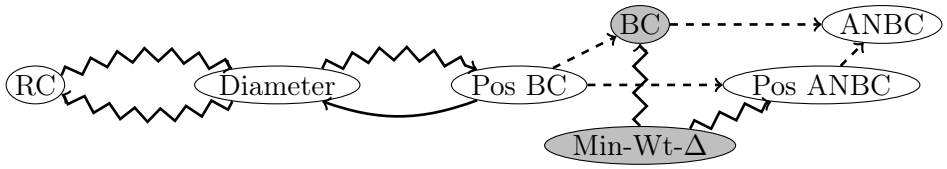


Figure 2.7: Known sparse reductions for centrality problems, all from [2]. The regular edges represent sparse $O(m+n)$ reductions, the squiggly edges represent tilde-sparse $O(m+n)$ reductions, and the dashed edges represent reductions that are trivial. BC and Min-Wt-Δ (shaded with gray) are known to be sub-cubic equivalent to APSP [2, 91].

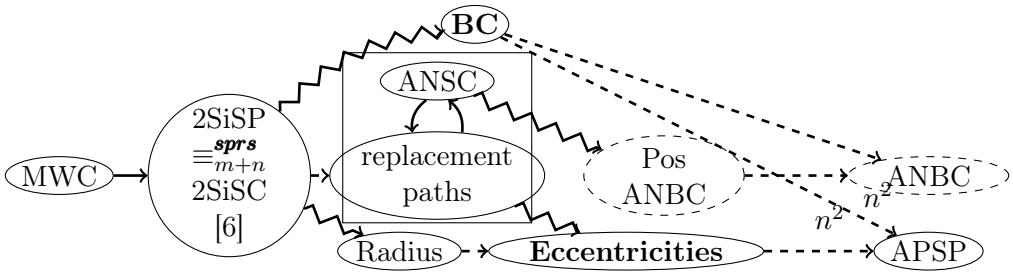


Figure 2.8: Sparse reductions for weighted directed graphs. The regular edges represent sparse $O(m+n)$ reductions, the squiggly edges represent tilde-sparse $O(m+n)$ reductions, and the dashed edges represent reductions that are trivial. All problems except APSP are MWCC-hard. Eccentricities, BC, ANBC and Pos ANBC are also SETH/ k-DSH hard. ANBC and Pos ANBC (problems inside the dashed circles) are both not known to be subcubic equivalent to APSP. BC and Eccentricities (in bold) are MWC-hard, sub-cubic equivalent to APSP and SETH/k-DSH hard.

is known to be both SETH-hard [78] and k-DSH Hard (Section 2.7) and hence all these problems in Figure 2.7 (except Min-Wt-Δ) are also SETH and k-DSH hard.

In this section, we give a sparse reduction from 2-SiSP to BC, establishing MWC-hardness for BC. We also give a tilde-sparse reduction from ANSC to Pos ANBC, and thus we have MWC-hardness for both Pos ANBC and for ANBC, though neither problem is known to be in the sub-cubic equivalence class. (Both have $\tilde{O}(mn)$ time algorithms, and have APSP-hardness under sub-cubic reductions.)

Figure 2.8 gives an updated partial order of our sparse reductions for weighted directed graphs; this figure augments Figure 2.1 by including the sparse reductions

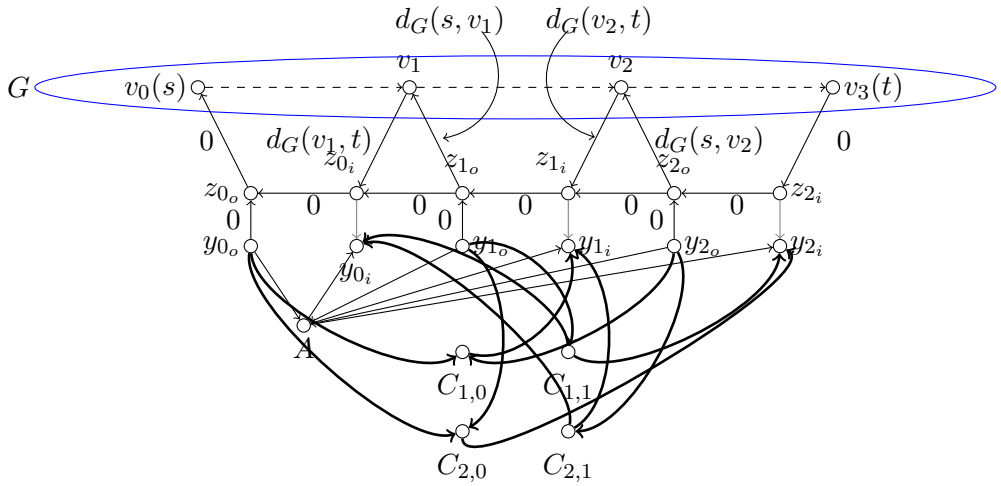


Figure 2.9: G'' for $l = 3$ in the reduction: directed 2-SiSP $\lesssim^{\boldsymbol{sprs}}_{m+n}$ Betweenness Centrality. The gray and the bold edges have weight M' and $\frac{1}{3}M'$ respectively. All the outgoing (incoming) edges from (to) A have weight $M' + q$ (0). Here $M' = 9nW_{max}$ where W_{max} is the largest edge weight in G and q is some value in the range from 0 to nW_{max}.

for BC problems given in this section.

2.6.1 2-SiSP to BC

Our sparse reduction from 2-SiSP to BC is similar to the reduction from 2-SiSP to Radius described in Section 2.5. The input is $G = (V, E, w)$, with source s and sink t in V, and a shortest path P ($s = v_0 \to v_1 \rightsquigarrow v_{l-1} \to v_l = t$). We need to compute a second simple s-t shortest path. Figure 2.9 gives an example of our reduction to an input G'' to the BC problem for $l = 3$.

In our reduction, we first map every edge (v_j, v_{j+1}) to new vertices y_{j_o} and y_{j_i} such that the shortest path from y_{j_o} to y_{j_i} corresponds to the replacement path from s to t for the edge (v_j, v_{j+1}). We then add an additional vertex A and connect it to vertices y_{j_o}'s and y_{j_i}'s. We also ensure that the only shortest paths passing through A are from y_{j_o} to y_{j_i}. We then do binary search on the edge weights for the edges

going from A to y_{j_i}'s with oracle calls to the Betweenness Centrality problem, to compute the weight of the shortest replacement path from s to t, which by definition of 2-SiSP, is the second simple shortest path from s to t.

Lemma 2.6.1. *In weighted directed graphs, 2-SiSP* $\leq^{\textbf{sprs}}_{m+n}$ *BC*

Proof. We are given an input graph $G = (V, E)$, a source vertex s and a sink/target vertex t and we wish to compute the second simple shortest path from s to t. Let P $(s = v_0 \to v_1 \rightsquigarrow v_{l-1} \to v_l = t)$ be the shortest path from s to t in G.

(i) Constructing G'': We first construct the graph G'', as described in the proof of Lemma 2.5.1, without the vertices A and B. For each $0 \leq j \leq l-1$, we change the weight of the edge from z_{j_i} to y_{j_i} to M' (where $M' = 9nW_{max}$ and W_{max} is the largest edge weight in G).

We add an additional vertex A and for each $0 \leq j \leq l-1$, we add an incoming (outgoing) edge from (to) y_{j_o} (y_{j_i}). We assign the weight of the edges from y_{j_o}'s to A as 0 and from A to y_{j_i}'s as $M' + q$ (for some q in the range 0 to nM).

Figure 2.9 depicts the full construction of G'' for $l = 3$.

We observe that for each $0 \leq j \leq l-1$, a shortest path from y_{j_o} to y_{j_i} with (z_{j_i}, y_{j_i}) as the last edge has weight equal to $M' + d_{G''}(z_{j_o}, z_{j_i})$.

(ii) We now show that the Betweenness Centrality of A, i.e. $BC(A)$, is equal to l iff $q < d_{G''}(z_{j_o}, z_{j_i})$ for each $0 \leq j \leq l-1$. The only paths that passes through the vertex A are from vertices y_{j_o}'s to vertices y_{j_i}'s. For $j \neq k$, as noted in the proof of Lemma 2.5.1, there exists some r, s such that there is a path from y_{j_o} to y_{k_i} that goes through $C_{r,s}$ and has weight equal to $\frac{2}{3}M'$. However a path from y_{j_o} to y_{k_i} has weight $M' + q$, which is strictly greater than $\frac{2}{3}M'$ and hence the pairs (y_{j_o}, y_{k_i}) does not contribute to the Betweenness Centrality of A.

Now if $BC(A)$, is equal to l, it implies that the shortest paths for all pairs (y_{j_o}, y_{j_i}) passes through A and there is exactly one shortest path for each such pair. Hence for each $0 \leq j \leq l-1$, $M' + q < M' + d_{G''}(z_{j_o}, z_{j_i})$. Thus $q < d_{G''}(z_{j_o}, z_{j_i})$

for each $0 \leq j \leq l-1$.

On the other hand if $q < d_{G''}(z_{j_o}, z_{j_i})$ for each $0 \leq j \leq l-1$, then the path from y_{j_o} to y_{j_i} with (z_{j_i}, y_{j_i}) as the last edge has weight $M' + d_{G''}(z_{j_o}, z_{j_i})$. However the path from y_{j_o} to y_{j_i} passing through A has weight $M' + q < M' + d_{G''}(z_{j_o}, z_{j_i})$. Hence every such pair contributes 1 to the Betweenness Centrality of A and thus $BC(A) = l$.

Thus using (ii), we just need to find the minimum value of q such that $BC(A) < l$ in order to compute the value $\min_{0 \leq j \leq l-1} d_{G''}(z_{j_o}, z_{j_i})$. We can find such q by performing a binary search in the range 0 to nM and computing $BC(A)$ at every layer. Thus we make $O(\log nW_{max})$ calls to the Betweenness Centrality algorithm.

As observed in the proof of Lemma 2.5.3, we know that the shortest path from z_{j_o} to z_{j_i} corresponds to the replacement path for the edge (v_j, v_{j+1}) lying on P. Thus by making $O(\log nW_{max})$ calls to the Betweenness Centrality algorithm, we can compute the second simple shortest path from s to t in G. This completes the proof.

The cost of this reduction is $O((m + n \log n) \cdot \log nW_{max})$. □

2.6.2 ANSC to Positive ANBC

We now describe a tilde-sparse reduction from the ANSC problem to the All Nodes Positive Betweenness Centrality problem (Pos ANBC). Sparse reductions from Min-Wt-Δ and from Diameter to Pos ANBC are given in [2]. However, Min-Wt-Δ can be solved in $O(m^{3/2})$ time, and Diameter is not known to be MWC-hard, hence neither of these reductions can be used to show hardness of the All Nodes Positive Betweenness Centrality problem relative to MWC hardness. (Recall that Pos ANBC is not known to be subcubic equivalent to APSP.)

Our reduction is similar to the reduction from 2-SiSP to the Betweenness

Centrality problem, but instead of computing betweenness centrality through one vertex, it computes the positive betweenness centrality values for n different nodes. The input is $G = (V, E, w)$ and we wish to compute the ANSC in G.

In this reduction, we first split every vertex x into vertices x_o and x_i such that the shortest path from x_o to x_i corresponds to the shortest cycle passing through x in the original graph. We then add additional vertices z_x for each vertex x in the original graph and connect it to the vertices x_o and x_i such that the only shortest path passing through z_x is from x_o to x_i. We then perform binary search on the edge weights for the edges going from z_x to x_i with oracle calls to the Positive Betweenness Centrality problem, to compute the weight of the shortest cycle passing through x in G.

Lemma 2.6.2. *In weighted directed graphs, ANSC* $\lesssim^{\boldsymbol{sprs}}_{m+n}$ *Pos ANBC*

Proof. We are given an input graph $G = (V, E)$ and we wish to compute the ANSC in G. Let W_{max} be the largest edge weight in G.

(i) Constructing G': Now we construct a graph G' from G. For each vertex $x \in V$, we replace x by vertices x_i and x_o and we place a directed edge of weight 0 from x_i to x_o, and we also replace each incoming edge to (outgoing edge from) x with an incoming edge to x_i (outgoing edge from x_o) in G'. We can observe that the shortest path from x_o to x_i in G' corresponds to the shortest cycle passing through x in G.

For each vertex $x \in V$, we add an additional vertex z_x in G' and we add an edge of weight 0 from x_o to z_x and an edge of weight q_x (where q_x lies in the range from 0 to nW_{max}) from z_x to x_i.

Figure 2.10 depicts the full construction of G' for $n = 3$.

We observe that the shortest path from x_o to x_i for some vertex $x \in V$ passes through z_x only if the shortest cycle passing through x in G has weight greater than q_x.

(ii) We now show that for each vertex $x \in V$, Positive Betweenness Centrality of z_x

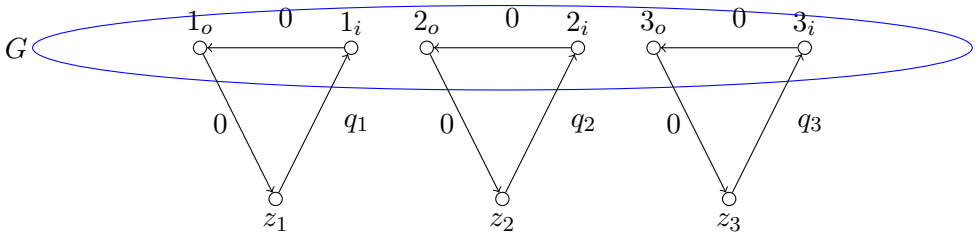


Figure 2.10: G' for $n = 3$ in the reduction: directed ANSC $\lesssim^{\boldsymbol{sprs}}_{m+n}$ Pos ANBC.

is true, i.e., $BC(z_x) > 0$ *iff the shortest cycle passing through* x *has weight greater than* q_x. It is easy to see that the only path that pass through vertex z_x is from x_o to x_i (as the only outgoing edge from x_i is to x_o and the only incoming edge to x_o is from x_i).

Now if $BC(z_x) > 0$, it implies that the shortest path from x_o to x_i passes through z_x and hence the path from x_o to x_i corresponding to the shortest cycle passing through x has weight greater than q_x.

On the other hand, if the shortest cycle passing through x has weight greater than q_x, then the shortest path from x_o to x_i passes through z_x. And hence $BC(z_x) > 0$.

Then using (ii), we just need to find the maximum value of q_x such that $BC(z_x) > 0$ in order to compute the weight of the shortest cycle passing through x in the original graph. We can find such q_x by performing a binary search in the range 0 to nW_{max} and computing Positive Betweenness Centrality for all nodes at every layer. Thus we make $O(\log nW_{max})$ calls to the Pos ANBC algorithm. This completes the proof.

The cost of this reduction is $O((m+n) \cdot \log nW_{max})$. □

2.7 Conditional Hardness Under k-DSH

Here we improve on a result shown in [78] that a sub-m^2 algorithm for Diameter would refute k-DSH for *even values* of k by showing sub-mn hardness for Diameter for both odd and even values of k. Since Diameter trivially reduces to Eccentricities and BC [2] and k-DSH hardness implies SETH hardness, this result also holds for Eccentricities and BC, and relative to both k-DSH and SETH.

Lemma 2.7.1. *Suppose for some constant α there is an $O(m^\alpha \cdot n^{2-\alpha-\epsilon})$ time algorithm, for some $\epsilon > 0$, for solving Diameter in an unweighted m-edge n-node graph, either undirected or directed. Then there exists a $k' > 0$ such that for all $k \geq k'$, the k-Dominating Set problem can be solved in $O(n^{k-\epsilon})$ time.*

Proof. When k is even we use a construction in [78]. To determine if undirected graph $G = (V, E)$ has a k-dominating set we form $G' = (V', E')$, where $V' = V_1 \cup V_2$, with V_1 containing a vertex for each subset of V of size $k/2$ and $V_2 = V$. We add an edge from a vertex $v \in V_1$ to a vertex $x \in V_2$ if the subset corresponding to v does not dominate x. We induce a clique in the vertex partition V_2. As shown in [78], G' has diameter 3 if G has a dominating set of size k and has diameter 2 otherwise, and this gives the reduction when k is even.

If k is odd, so $k = 2r+1$, we make n calls to graphs derived from $G' = (V', E')$ as follows, where now each vertex in V_1 represents a subset of r vertices in V. For each $x \in V$ let V_x be the set $\{x\} \cup \{\text{neighbors of } x \text{ in } G\}$, and let G_x be the subgraph of G' induced on $V - V_x$. If G has a dominating set D of size k that includes vertex x then consider any partition of the remaining $2r$ vertices in D into two subsets of size r each, and let u and v be the vertices corresponding to these two sets in V_1. Since all paths from u to v in G_x pass through $V_2 - V_x$, there is no path of length 2 from u to v since every vertex in $V_2 - V_x$ is covered by either u or v. Hence the diameter of G_x is greater than 2 in this case. But if there is no dominating set of

size k that includes x in G, then for any $u, v \in V_1$, at least one vertex in $V_2 - V_x$ is not covered by both u and v and hence there is a path of length 2 from u to v. If we now compute the diameter in each of graphs G_x, $x \in V$, we will detect a graph with diameter greater than 2 if and only if G has a dominating set of size k.

Each graph G_x has $N = O(n^r)$ vertices and $M = O(n^{r+1})$ edges. If we now assume that Diameter can be computed in time $O(M^\alpha \cdot N^{2-\alpha-\epsilon})$, then the above algorithm for k Dominating Set runs in time $O(n \cdot M^\alpha \cdot N^{2-\alpha-\epsilon}) = O(n^{2r+1-\epsilon r+\alpha})$, which is $O(n^{k-\epsilon})$ time when $k \geq 3 + \frac{2\alpha}{\epsilon}$. The analysis is similar for k even. In the directed case, we get the same result by replacing every edge in G' with two directed edges in opposite directions. □

2.8 Time Bounds for Sparse Graphs

Let $T(m, n)$ be a function which is defined for $m \geq n - 1$. We will interpret $T(m, n)$ as a time for an algorithm on a connected graph and we will refer to $T(m, n)$ as a *time bound* for a graph problem. We now focus on formalizing the notion of a time bound $T(m, n)$ being smaller than another time bound $T'(m, n)$ for sparse graphs.

If the time bounds $T(m, n)$ and $T'(m, n)$ are of the form $m^\alpha n^\beta$, then one possible way to check if $T(m, n)$ is smaller than $T'(m, n)$ is to check if the exponents of m and n in $T(m, n)$ are individually smaller than the corresponding exponents in $T'(m, n)$. But using this approach, we would not be able to compare between time bounds $m^{1/2}n$ and $mn^{1/2}$. Another possible way is to use a direct extrapolation from the single variable case and define $T(m, n)$ to be (polynomially) smaller than $T'(m, n)$ if $T(mn) = O((T'(m, n))^{1-\epsilon})$ for some constant $\epsilon > 0$. But such a definition would completely ignore the dependence of the functions on each of their two variables. We would want our definition to take into account the sparsity of the graph, i.e., as a graph becomes sparser, the smaller time bound has smaller running time. To incorporate this idea, our definition below asks for $T(m, n)$ to be a factor of m^ϵ

smaller than $T'(m,n)$, for some $\epsilon > 0$. Further, this requirement is placed only on *sufficiently sparse graphs* (and for a weakly smaller time bound, we also require a certain minimum edge density). The consequence of this definition is that when one time bound is not dominated by the other for all values of m, the domination needs to hold for sufficiently sparse graphs in order for the dominated function to be a smaller time bound for sparse graphs.

Definition 2.8.1 (**Comparing Time Bounds for Sparse Graphs**). *Given two time bounds $T(m,n)$ and $T'(m,n)$,*

(i) $T(m,n)$ *is a* smaller time bound than $T'(m,n)$ for sparse graphs *if there exist constants $\gamma, \epsilon > 0$ such that $T(m,n) = O\left(\frac{1}{m^\epsilon} \cdot T'(m,n)\right)$ for all values of $m = O(n^{1+\gamma})$.*

(ii) $T(m,n)$ *is a* weakly smaller time bound than $T'(m,n)$ for sparse graphs *if there exists a positive constant γ such that for any constant δ with $\gamma > \delta > 0$, there exists an $\epsilon > 0$ such that $T(m,n) = O\left(\frac{1}{m^\epsilon} \cdot T'(m,n)\right)$ for all values of m in the range $m = O(n^{1+\gamma})$ and $m = \Omega(n^{1+\delta})$.*

Part (i) in above definition requires a polynomially smaller (in m) bound for $T(m,n)$ relative to $T'(m,n)$ for sufficiently sparse graphs. For example, $\frac{m^3}{n^2}$ is a smaller time bound than $m^{3/2}$, which in turn is a smaller time bound than mn; m^2 is a smaller time bound than n^3. A time bound of $n\sqrt{m}$ is a weakly smaller bound than $m\sqrt{n}$ for sparse graphs by part (ii) but not a smaller bound since the two bounds coincide when $m = O(n)$.

Our definition for comparing time bounds for sparse graphs is quite strong as it allows us to compare a wide range of time bounds. For example, using this definition we can say that $n\sqrt{m}$ is a weakly smaller bound than $m\sqrt{n}$ for sparse graphs. Whereas if we use the possible approaches that we discussed before then we would not be able to compare these two time bounds.

With Definition 2.8.1 in hand, the following lemma is straightforward.

Lemma 2.8.2. *Let $T_1(m,n) = O(m^{\alpha_1}n^{\beta_1})$ and $T_2(m,n) = O(m^{\alpha_2}n^{\beta_2})$ be two time bounds, where $\alpha_1, \beta_1, \alpha_2, \beta_2$ are constants.*

(i) $T_1(m,n)$ is a smaller time bound than $T_2(m,n)$ for sparse graphs if $\alpha_2 + \beta_2 > \alpha_1 + \beta_1$.

(ii) $T_1(m,n)$ is a weakly smaller time bound than $T_2(m,n)$ for sparse graphs if $\alpha_2 + \beta_2 = \alpha_1 + \beta_1$, and $\alpha_2 > \alpha_1$.

Definition 2.8.1, in conjunction with Lemma 2.8.2 and Theorem 2.3.9, lead to the following *provable separation* of time bounds for sparse graph problems in the sub-cubic equivalence class:

Theorem 2.8.3 (**Split of Time Bounds for Sparse Graphs.**). *Under either SETH or k-DSH, triangle finding problems in the sub-cubic equivalence class have algorithms with a smaller time bound for sparse graphs than* any algorithm *we can design for Eccentricities.*

2.9 Additional Results

2.9.1 k-SiSC Algorithm : Undirected Graphs

This section deals with an application of our *bit-sampling technique* to obtaining a new near-linear time algorithm for k-SiSC in undirected graphs (see definition below). Note that this problem is not in the mn class and this result is included here as an application of the *bit-sampling technique*.

k-SiSC is the problem of finding k simple shortest cycles passing through a vertex v. Here the output is a sequence of k simple cycles through v in non-decreasing order of weights such that the i-th cycle in the output is different from the previous

$i-1$ cycles. The corresponding path version of this problem is known as k-SiSP and is solvable in near linear time in undirected graphs [57].

We now use our *bit-sampling technique* (described in Section 2.4) to get a near-linear time algorithm for k-SiSC, which was not previously known. We obtain this k-SiSC algorithm by giving a tilde-sparse $\tilde{O}(m+n)$ time reduction from k-SiSC to k-SiSP. This reduction uses our *bit-sampling technique* for sampling the edges incident to v and creates $\lceil \log n \rceil$ different graphs. Here we only use index i of our bit-sampling method.

Lemma 2.9.1. *In undirected graphs, k-SiSC $\leq^{\boldsymbol{sprs}}_{\sim(m+n)}$ k-SiSP.*

Proof. Let the input be $G = (V, E)$ and let $x \in V$ be the vertex for which we need to compute k-SiSC. Let $\mathcal{N}(x)$ be the neighbor-set of x. We create $\lceil \log n \rceil$ graphs $G_i = (V_i, E_i)$ such that $\forall 1 \leq i \leq \lceil \log n \rceil$, G_i contains two additional vertices $x_{0,i}$ and $x_{1,i}$ (instead of the vertex x) and $\forall y \in \mathcal{N}(x)$, the edge $(y, x_{0,i}) \in E_i$ if y's i-th bit is 0, otherwise the edge $(y, x_{1,i}) \in E_i$. This is our bit-sampling method.

The construction takes $O((m+n) \cdot \log n)$ time and we observe that every cycle through x will appear as a path from $x_{0,i}$ to $x_{1,i}$ in at least one of the G_i. Hence, the k-th shortest path in the collection of k-SiSPs from $x_{0,i}$ to $x_{1,i}$ in $\log n$ G_i, $1 \leq i \leq \lceil \log n \rceil$ (after removing duplicates), corresponds to the k-th SiSC passing through x. □

Using the undirected k-SiSP algorithm in [57] that runs in $O(k \cdot (m + n \log n))$, we obtain an $O(k \log n \cdot (m + n \log n))$ time algorithm for k-SiSC in undirected graphs.

2.10 Conclusion and Open Problems

We have given an extensive collection of sparse reductions for path problems in $\tilde{O}(mn)$ class and have established MWC as a key problem for this class. Several

open problems remain of which we mention two.

Our reduction from ANSC to APSP in undirected graphs only works for the unweighted case. An open question here is to extend this reduction to the weighted case or to come up with an altogether different reduction.

The directed and undirected versions of most of these problems are known to be subcubic equivalent [91]. However such an equivalence is not known for the sparse case. For path problems, the undirected versions trivially reduces to their directed versions though nothing is known for the reverse case. Nothing is known in either direction for cycle problems. It will be interesting to see if one can establish sparse reductions for these problems.

Chapter 3

k-Simple Shortest Paths and Cycles

3.1 Introduction

In this work we study a related problem to computing shortest paths, known as k simple shortest paths (k-SiSP). In Chapter 2 we showed that the 2-SiSP problem is MWC-Hard and it cannot be solved in sub-mn time unless Minimum Weight Cycle can be solved in sub-mn time. In this section we describe our new algorithms and fundamentally new techniques for several problems related to finding multiple simple shortest paths and cycles in a graph.

In the *k simple shortest paths (k-SiSP)* problem, given a pair of vertices s, t, the output is a sequence of k simple paths from s to t, where the i-th path in the collection is a shortest simple path in the graph that is not identical to any of the $i-1$ paths preceding it in the output. (Note that these k simple shortest paths need not have the same weight.) It is noted in [30] that the k-SiSP problem is more common than the version where a path can contain cycles.

In this work we consider the problem of generating multiple simple shortest

paths (SiSP) and cycles (SiSC) in a weighted directed graph under the following set-ups: the k simple shortest paths for all pairs of vertices (k-APSiSP), k simple shortest paths in the overall graph (k-All-SiSP), and the corresponding problem of finding simple shortest cycles in the overall graph (k-All-SiSC). We obtained significantly faster algorithms for k-APSiSP for small values of k, and fast algorithms, that also appear to be the first nontrivial algorithms, for the remaining two problems for all $k \geq 1$. Implicit in our method for k-All-SiSC are new algorithms for finding k simple shortest cycles through a specified vertex (k-SiSC) and through every vertex (k-ANSiSC) in weighted directed graphs.

The techniques we use in our algorithms are of special interest: We use two *path extension* techniques, a new method for k-APSiSP, and another for k-All-SiSP that is related to a method used in [25] for fully dynamic APSP, but which is still new for the context in which we use it.

Related Work For the case when the k shortest paths need not be simple, the all-pairs version (k-APSP) was considered in the classical papers of Lawler [61, 62] and Minieka [69]. The most efficient current algorithm for k-APSP runs the k-SSSP algorithm in [30] on each of the n vertices in turn, leading to a bound of $O(mn + n^2 \log n + kn^2)$. It was noted in Minieka [69] that the all-pairs version of k shortest paths becomes significantly harder when simple paths are required, i.e., that the problem we study here, k-APSiSP, appears to be significantly harder than k-APSP.

Even for a single source-sink pair, the problem of generating k simple shortest paths (k-SiSP) is considerably more challenging than the unrestricted version considered in [30]. Yen's algorithm [92] finds the k simple shortest paths for a specific pair of vertices in $O(k \cdot (mn + n^2 \log n))$. This time bound was improved slightly [41], using Pettie's faster APSP algorithm [76], to $O(k(mn + n^2 \log \log n))$. On the other hand, it is shown in [91] that if the *second* simple shortest path for a single source-

sink pair (i.e., $k = 2$ in k-SiSP) can be found in $O(n^{3-\delta})$ time for some $\delta > 0$, then APSP can also be computed in $O(n^{3-\alpha})$ time for some $\alpha > 0$; the latter is a major open problem. Thus, for dense graphs, where $m = \Theta(n^2)$, we cannot expect to improve the $\tilde{O}(mn)$ bound, even for 2-SiSP, unless we solve a major and long-standing open problem for APSP.

The k-SiSP problem is much simpler in the undirected case and is known to be solvable in $O(k(m + n \log n))$ time [57]. For unweighted directed graphs, Roditty and Zwick [80] gave an $\tilde{O}(km\sqrt{n})$ randomized algorithm for directed k-SiSP. They also showed that k-SiSP can be solved with $O(k)$ executions of an algorithm for the 2-SiSP problem.

A problem related to 2-SiSP is the *replacement paths* problem. In the s-t version of this problem, we need to output a shortest path from s to t when an edge on the shortest path p is removed; the output is a collection of $|p|$ paths, each a shortest path from s to t when an edge on p is removed. Clearly, given a solution to the s-t replacement paths problem, the second shortest path from s to t can be computed as the path of minimum weight in this solution. This is essentially the method used in all prior algorithms for 2-SiSP (and with modifications, for k-SiSP), and thus the current fastest algorithms for 2-SiSP and replacement paths have the same time bound. For the all-pairs case that is of interest to us, the output for the replacement paths problem would be $O(n^3)$ paths, where each path is shortest for a specific vertex pair, when a specific edge in its shortest path is removed. In view of the large space needed for this output, in the all-pairs version of replacement paths, the problem of interest is *distance sensitivity oracles (DSO).* Here, the output is a compact representation from which any specific replacement path can be found with $O(1)$ time. The first such oracle was developed in Demetrescu et. al. [26], and it has size $O(n^2 \log n)$. The current best construction time for an oracle of this size is $O(mn \log n + n^2 \log^2 n)$ time for a randomized algorithm, and a log factor slower for

a deterministic algorithm, given in Bernstein and Karger [17]. Given such an oracle, the output to 2-APSiSP can be computed with $O(n)$ queries for each source-sink pair, i.e., with $O(n^3)$ queries to the DSO.

To the best of our knowledge, for $k > 1$ the problem of generating k simple shortest cycles in the overall graph in non-decreasing order of their weights (k-All-SiSC) has not been studied before, and neither has k-SiSC (k Simple Shortest Cycles through a given node) or k-ANSiSC (k All Nodes Simple Shortest Cycles); for $k = 1$, 1-All-SiSC asks for a minimum weight cycle and 1-ANSiSC is the ANSC problem [93], both of which can be found in $\tilde{O}(mn)$ time, and 1-SiSC can be solved in $\tilde{O}(m + n)$ time. On the other hand, enumerating simple (or *elementary*) cycles in no particular order — which is thus a special case of k-All-SiSC — has been studied extensively [86, 89, 84, 53]. The first polynomial time algorithm was given by Tarjan [84], and ran in $O(kmn)$ time for k cycles. This result was improved to $O(k \cdot m + n)$ by Johnson [53]. We do not expect to match this linear time result for k-All-SiSC since it includes the minimum weight cycle problem for $k = 1$.

3.2 Our Results

A summary of our results is given in Table 2.1.

Computing k simple shortest paths for all pairs (k-APSiSP) in G. We came up with a new approach to the k-APSiSP problem, which computes the sets $P_k^*(x, y)$ as defined below. Our method introduces the key notion of a 'nearly k SiSP set', $Q_k(x, y)$, defined as follows.

Definition 3.2.1. *Let $G = (V, E)$ be a directed graph with non-negative edge weights. For $k \geq 2$, and a vertex pair x, y, let $k^* = \min\{r, k\}$, where r is the number of simple paths from x to y in G. Then,*

1. *$P_k^*(x, y)$ is the set of k^** simple shortest paths from x to y *in G*

Problem	Known Results	Our Results
2-APSiSP	$O(n^3 + mn \log^2 n)$ (using DSO [17])	$\boldsymbol{O(mn + n^2 \log n)}$
3-APSiSP	$\tilde{O}(mn^3)$ [92]	$\boldsymbol{O(mn^2 + n^3 \log n)}$
k-SiSC	—	$\boldsymbol{O(k \cdot (mn + n^2 \log\log n))}$
k-ANSiSC	—	$\boldsymbol{O(mn + n^2 \log n)}$ **if** $\boldsymbol{k = 2}$ **and** $\boldsymbol{O(k \cdot (mn^2 + n^3 \log\log n))}$ **if** $\boldsymbol{k > 2}$
k-All-SiSC	—	$\boldsymbol{\tilde{O}(kmn)}$
k-All-SiSP	—	$\boldsymbol{\tilde{O}(k)}$ **if** $\boldsymbol{k < n}$ **and** $\boldsymbol{\tilde{O}(n)}$ **if** $\boldsymbol{k \geq n}$ **per path amortized, after a startup cost of** $\boldsymbol{O(m)}$

Table 3.1: Our results for directed graphs. All algorithms are deterministic. (DSO stands for Distance Sensitivity Oracles).

2. $Q_k(x, y)$ *is the set of* k nearly simple shortest paths from x to y, *defined as follows. If* $k^* = k$ *and the* $k - 1$ *simple shortest paths from* x *to* y *share the same first edge* (x, a) *then* $Q_k(x, y)$ *contains these* $k - 1$ *simple shortest paths, together with the simple shortest path from* x *to* y *that does not start with edge* (x, a)*, if such a path exists. Otherwise (i.e, if either the former or latter condition does not hold),* $Q_k(x, y) = P_k^*(x, y)$.

Our algorithm for k-APSiSP first constructs $Q_k(x, y)$ for all pairs of vertices x, y, and then uses these sets in an efficient algorithm, Compute-APSiSP, to compute the $P_k^*(x, y)$ for all x, y. The latter algorithm runs in time $O(k \cdot n^2 + n^2 \log n)$ for any k, while our method for constructing the $Q_k(x, y)$ depends on k. For $k = 2$ we present an $O(mn + n^2 \log n)$ time method to compute the $Q_2(x, y)$ sets; this gives a 2-APSiSP algorithm that matches Yen's bound of $O(mn + n^2 \log n)$ for 2-*SiSP for a single pair of vertices.* It is also faster (by a poly-logarithmic factor) than the best algorithm for DSO (distance sensitivity oracles) for the all-pairs replacement paths

problem [17]. In fact, we also show that the $Q_2(x, y)$ sets can be computed in $O(n^2)$ time using a DSO, and hence 2-APSiSP can be computed in $O(n^2 \log n)$ time plus the time to construct the DSO.

For $k \geq 3$ our algorithm to compute the Q_k sets makes calls to an algorithm for $(k-1)$-APSiSP, so we combine the two components together in a single recursive method, APSiSP, that takes as input G and k, and outputs the P_k^* sets for all vertex pairs. The time bound for APSiSP increases with k: it is faster than Yen's method for $k = 3$ by a factor of n (and hence is faster than the current fastest method by almost a factor of n), it matches Yen for $k = 4$, and its performance degrades for larger k.

If a faster algorithm can be designed to compute the Q_k sets, then we can run Compute-APSiSP on its output and hence compute k-APSiSP in additional $O(k \cdot n^2 + n^2 \log n)$ time. Thus, a major open problem left by our results is the design of a faster algorithm to compute the Q_k sets for larger values of k.

New Approach: Computing simple shortest paths without finding detours. Our method for computing k-APSiSP (using the $Q_k(x, y)$ sets) extends an existing simple path in the data structure to create a new simple path by adding a single incoming edge. This approach differs from all previous approaches to finding k simple paths and replacement paths. All known previous algorithms for 2-SiSP compute replacement paths for every edge on the shortest path (by computing suitable 'detours'). In fact, Hershberger et al. [46] present a lower bound for k-SiSP, exclusively for the class of algorithms that use detours, by pointing out that all known algorithms for k-SiSP compute replacement paths, and all known replacement path algorithms use detours. In contrast, our method may enumerate and inspect paths that are not detours, including paths with cycles. Thus our method is fundamentally new.

Generating k simple shortest cycles and paths (k-All-SiSC, k-SiSC, k-

ANSiSC) and k-All-SiSP. We consider the problem of generating the k simple shortest cycles in the graph G in non-increasing order of their weight (k-All-SiSC). In Section 3.4 we came up with an algorithm for k-All-SiSC that runs in $\tilde{O}(k \cdot mn)$ time by generating each successive simple shortest cycle in G in $\tilde{O}(mn)$ time. The same algorithm can be used to enumerate all simple cycles in G in non-decreasing order of their weights. Recall that the related problem of simply enumerating simple cycles in a graph in no particular order was a very well-studied classical problem [86, 89, 84, 53] until an algorithm that generates successive cycles in linear time was obtained [53]. Our algorithm does not match the linear time bound per successive cycle, but it is to be noted that 1-All-SiSC (i.e., the problem of generating a minimum weight cycle) is a very fundamental and well-studied problem for which the current best bound is $\tilde{O}(mn)$.

Our algorithm for k-All-SiSC creates a auxiliary graph on which suitable SiSP computation can be performed to generate the desired output. Using the same auxiliary graph, we came up with fast algorithms for k-SiSC and k-ANSiSC.

Complementing our result for k-All-SiSC, we present in Section 3.5 an algorithm for k-All-SiSP that generates each successive simple path in $\tilde{O}(k)$ time if $k < n$, and in $\tilde{O}(n)$ time if $k > n$, after an initial start-up cost of $O(m)$ to find the first path. This time bound is considerably faster than that for k-All-SiSC. Our method, All-SiSP, is again one of extending existing paths by an edge (as is Compute-APSiSP); it is, however, a different path extension method.

Path Extensions. We use two different path extension methods, one for k-APSiSP and the other for k-All-SiSP. Path extensions have been used before in the hidden paths algorithm for APSP [55] and more recently, for fully dynamic APSP [25]. Our path extension method for k-All-SiSP is inspired by a method in [25] to compute 'locally shortest paths' for fully dynamic APSP. Our path extension method for k-APSiSP appears to be new.

Here are the main theorems we establish for our algorithmic results. In all cases, the input is a directed graph $G = (V, E)$ with nonnegative edge weights.

Theorem 3.2.2. *Given an integer $k > 1$, and the nearly simple shortest paths sets $Q_k(x, y)$ (Definition 3.2.1) for all $x, y \in V$, Algorithm* Compute-APSiSP *produces the k simple shortest paths for every pair of vertices in $O(k \cdot n^2 + n^2 \log n)$ time.*

Theorem 3.2.3. *(i) Algorithm* 2-APSiSP *correctly computes 2-APSiSP in $O(mn + n^2 \log n)$ time.*

(ii) For $k > 2$, Algorithm APSiSP *correctly computes k-APSiSP in $T(m, n, k)$ time, where $T(m, n, k) \leq n \cdot T(m, n, k-1) + O(mn + n^2 \cdot (k + \log n))$.*

(iii) $T(m, n, 3)$, the time bound for algorithm APSiSP *for $k = 3$, is $O(m \cdot n^2 + n^3 \cdot \log n)$.*

Theorem 3.2.4. *(k-All-SiSC) After an initial start-up cost of $O(mn + n^2 \log n)$ time, we can compute each successive simple shortest cycle in $O(mn + n^2 \log \log n)$ time. This computes k-All-SiSC.*

Theorem 3.2.5. *(k-All-SiSP) After an initial start-up cost of $O(m)$ time to generate the first path, Algorithm* All-SiSP *computes each succeeding simple shortest path with the following bounds:*

(i) amortized $O(k + \log n)$ time if $k = O(n)$ and $O(n + \log k)$ time if $k = \Omega(n)$;

(ii) worst-case $O(k \cdot \log n)$ time if $k = O(n)$, and $O(n \cdot \log k)$ time if $k = \Omega(n)$.

3.3 The k-APSiSP Algorithm

In this section, we present our algorithm to compute k-APSiSP on a directed graph $G = (V, E)$ with non-negative edge-weight function wt. The algorithm has two main steps. In the first step it computes the nearly k-SiSP sets $Q_k(x, y)$ for all pairs x, y.

In the second step it computes the exact k-SiSP sets $P_k^*(x, y)$ for all x, y using the $Q_k(x, y)$ sets. This second step is the same for any value of k, and we describe this step first in Section 3.3.1. We then present efficient algorithms to compute the Q_k sets for $k = 2$ and $k > 2$.

In our algorithms we maintain the paths in each $P_k^*(x, y)$ and $Q_k(x, y)$ set in an array in non-decreasing order of edge-weights.

3.3.1 The Compute-APSiSP Procedure

In this section we present an algorithm, COMPUTE-APSISP, to compute k-APSiSP. This algorithm takes as input, the graph G, together with the nearly k-SiSP sets $Q_k(x, y)$, for each pair of distinct vertices x, y, and outputs the k^* simple shortest paths from x to y in the set $P_k^*(x, y)$ for each pair of vertices $x, y \in V$ (note that k^*, which is defined in Definition 3.2.1, can be different for different vertex pairs x, y). As noted above, the construction of the $Q_k(x, y)$ sets will be described in the next section.

The *right (left) subpath* of a path π is defined as the path obtained by removing the first (last) edge on π. If π is a single edge (x, y) then this path is the vertex y (x).

Lemma 3.3.1. *Suppose there are k simple shortest paths from x to y, all having the same first edge (x, a). Then $\forall i$, $1 \leq i \leq k$, the right subpath of the i-th simple shortest path from x to y has weight equal to the weight of the i-th simple shortest path from a to y.*

Proof. By induction on k. Since subpaths of shortest paths are shortest paths, the statement holds for $k = 1$. Assume the statement is true for all $h \leq k$, and consider the case when the $h + 1$ simple shortest paths from x to y all share the same first edge (x, a). Inductively, the right subpath of each of the first h simple shortest paths have the weight equal to the corresponding simple shortest paths from a to y.

Suppose the weight of the right subpath $\pi_{a,y}$ of the $(h+1)$-th simple shortest path from x to y is not equal to the weight of the $(h+1)$-th simple shortest path from a to y. Hence, if $\pi^{'}_{a,y}$ is the $(h+1)$-th simple shortest path from a to y, we must have $wt(\pi_{a,y}) > wt(\pi^{'}_{a,y})$.

Since $\pi_{xa,y}$ is the $(h+1)$-th simple shortest path from x to y and $wt(\pi_{a,y}) > wt(\pi^{'}_{a,y})$, there exists at least one path from a to y that contains x and is also the j-th simple shortest path from a to y, where $j \leq h+1$. Let this path be $\pi^{''}_{a,y}$. Let the subpath of $\pi^{''}_{a,y}$ from x to y be $\pi^{''}_{xa',y}$. But then $wt(\pi^{''}_{xa',y}) < wt(\pi^{''}_{a,y}) \leq wt(\pi^{'}_{a,y}) < wt(\pi_{a,y}) < wt(\pi_{xa,y})$. But this is a contradiction to our assumption that all the first $h+1$ simple shortest paths from x to y contains (x,a) as the first edge. This contradiction establishes the induction step and the lemma. □

Algorithm COMPUTE-APSISP computes the $P^*_k(x,y)$ sets by extending an existing path by an edge. In particular, if the k-SiSPs from x to y all use the same first edge (x,a), then it computes the k-th SiSP by extending the k-th SiSP from a to y (otherwise, the sets $P^*_k(x,y)$ are trivially computed from the sets $Q_k(x,y)$). The algorithm first initializes the $P^*_k(x,y)$ sets with the corresponding $Q_k(x,y)$ sets in Step 4. In Step 5, it checks whether the shortest $k-1$ paths in $P^*_k(x,y)$ have the same first edge and if so, by definition of $Q_k(x,y)$, this $P^*_k(x,y)$ may not have been correctly initialized, and may need to update its k-th shortest path to obtain the correct output. In this case, the common first edge (x,a) is added to the set $Extensions(a,y)$ in Step 7. We explain this step below.

We define the *k-Left Extended Simple Path (k-LESiP)* $\pi_{xa,y}$ from x to y as the path $\pi_{xa,y} = (x,a) \circ \pi_{a,y}$, where the path $\pi_{a,y}$ is the k-th shortest path in $Q_k(a,y)$, and $\circ$ denotes the concatenation operation. In our algorithm we will construct k-LESiPs for those pairs x,y for which the $k-1$ simple shortest paths all start with the edge (x,a). The algorithm also maintains a set $Extensions(a,y)$ for each pair of distinct vertices a,y; this set contains those edges (x,a) incoming to a which are

Algorithm 1 COMPUTE-APSiSP($G = (V, E), wt, k, \{Q_k(x, y), \forall x, y\}$)

Initialize:
$H \leftarrow \phi$ {H is a priority queue.}
for all $x, y \in V, x \neq y$ **do**
    $P_k^*(x, y) \leftarrow Q_k(x, y)$
    if the $k-1$ shortest paths in $P_k^*(x, y)$ have the same first edge **then**
        Let (x, a) be the first edge in the $(k-1)$ shortest paths in $P_k^*(x, y)$
        Add (x, a) to the set $Extensions(a, y)$
        if $|Q_k(a, y)| = k$ **then**
            $\pi \leftarrow$ the path of largest weight in $Q_k(a, y)$
            $\pi' \leftarrow (x, a) \circ \pi$
            Add π' to H with weight $wt(x, a) + wt(\pi)$
Main Loop:
while $H \neq \phi$ **do**
    $\pi \leftarrow$ EXTRACT-MIN(H)
    Let $\pi = (xa, y)$ and π' a path of largest weight in $P_k^*(x, y)$
    if $|P_k^*(x, y)| = k - 1$ **then**
        add π to $P_k^*(x, y)$ and set *update* flag
    else if $wt(\pi) < wt(\pi')$ **then**
        Replace π' with π in $P_k^*(x, y)$ and set *update* flag
    if *update* flag is set **then**
        for all $(x', x) \in Extensions(x, y)$ **do**
            Add $(x', x) \circ \pi$ to H with weight $wt(x', x) + wt(\pi)$

the first edge on all $k-1$ SiSPs from x to y. In addition to adding the common first edge (x,a) in the $(k-1)$ SiSPs in $P_k^*(x,y)$ to $Extensions(a,y)$ in Step 7, the algorithm creates the k-LESiP with start edge (x,a) and end vertex y using the k-th shortest path in the set $P_k^*(a,y)$, and adds it to heap H in Steps 8-11. Let $\mathcal{U}$ denote the set of $P_k^*(x,y)$ sets which may need to be updated; these are the sets for which the if condition in Step 5 holds.

In the main while loop in Steps 13-22, a min-weight path is extracted in each iteration. We establish below that this min-weight path is added to the corresponding P_k^* in Step 17 or 19 only if it is the k-th SiSP; in this case, its left extensions are created and added to the heap H in Step 22, and we note that some of these paths could be cyclic.

Lemma 3.3.2. *Let $G=(V,E)$ be a directed graph with non-negative edge weight function wt, and $\forall x,y \in V$, let the set $Q_k(x,y)$ contain the nearly k-SiSPs from x to y. Then, algorithm* Compute-APSiSP *correctly computes the sets $P_k^*(x,y)$ $\forall x,y \in V$.*

Proof. First, we need to show that the paths in sets $P_k^*(x,y)$ are indeed simple. Clearly, the paths added to P_k^* from sets Q_k in Step 4 are already simple (from the definition of Q_k). So we only need to show that the paths added to P_k^* in Steps 17 and 19 are simple. To the contrary assume that some of the paths that are added to P_k^* are non-simple. Clearly these paths must be of length greater than 1. Let $\pi_{xa,y} = x \rightarrow a \rightsquigarrow y$ be the first minimum weight path extracted from H that contains a cycle and was added to P_k^* in Step 17 or 19. Clearly, $P_k^*(x,y) \in \mathcal{U}$ and $(x,a) \in Extensions(a,y)$ and the right subpath $\pi_{a,y}$ must be in P_k^* (otherwise the path $\pi_{xa,y}$ would never have been added to heap H in Step 11 or 22). The right subpath $\pi_{a,y}$ must also be simple (as $wt(\pi_{a,y}) < wt(\pi_{xa,y})$), and it must contain x in order to create a cycle in $\pi_{xa,y}$. Let $\pi_{xa',y}$ $(a' \neq a)$ be the subpath of $\pi_{a,y}$ from x to y. Now there are two cases depending on whether $\pi_{xa,y}$ was added to P_k^* in Step

17 or 19.

If $\pi_{xa,y}$ was added to $P_k^*(x,y)$ in Step 17 and as $P_k^*(x,y) \in \mathcal{U}$, it implies that all $k-1$ paths in $Q_k(x,y)$ have same first edge (x,a) and there is no simple path from x to y in $Q_k(x,y)$ with some first edge $(x,a'') \neq (x,a)$. This is a contradiction as the subpath $\pi_{xa',y}$ of $\pi_{a,y}$ contains $(x,a') \neq (x,a)$ as its first edge.

Otherwise, let $\pi_{xa'',y} \in Q_k(x,y)$ $(a'' \neq a)$ be the path that was removed from P_k^* in Step 19 to accommodate $\pi_{xa,y}$. Thus, we have $wt(\pi_{xa',y}) < wt(\pi_{xa,y}) < wt(\pi_{xa'',y})$, which is a contradiction as $\pi_{xa'',y} \in Q_k(x,y)$ and is the shortest path from x to y avoiding edge (x,a) (as the other $k-1$ shortest paths in $Q_k(x,y)$ have (x,a) as the first edge). As path $\pi_{xa,y}$ is arbitrary, hence all paths in P_k^* are simple.

Now we need to show that $P_k^*(x,y)$ indeed contains the k^* SiSPs from x to y.

From the definition of $Q_k(x,y)$, it is evident that $P_k^*(x,y)$ indeed contains the $k-1$ SiSPs from x to y. We now need to show that the k-th shortest path in each of the sets P_k^* is indeed the corresponding k-th SiSP. To the contrary assume that there exists a P_k^* set that does not contain the correct k-th SiSP. Let $\pi_{xa,y} = x \rightarrow a \rightsquigarrow y$ be the minimum weight k-th SiSP that is not present in P_k^*. Clearly, $\pi_{xa,y} \notin Q_k(x,y)$ (otherwise it would have been added to $P_k^*(x,y)$ in Step 4). This implies that $\pi_{xa,y}$ has the same first edge as that of the $k-1$ SiSPs from x to y and hence $P_k^*(x,y) \in \mathcal{U}$ and $(x,a) \in \mathit{Extensions}(a,y)$. By Lemma 3.3.1, the right subpath of $\pi_{xa,y}$ must have weight equal to the k-th SiSP from a to y. Thus, there are at least k SiSPs from a to y and the set $P_k^*(a,y)$ contains all the k SiSPs from a to y. And as $(x,a) \in \mathit{Extensions}(a,y)$, a path $\pi'_{xa,y}$ with the k-th SiSP from a to y as the right subpath and weight equal to $wt(\pi_{xa,y})$ must have been added to H either in Step 11 or 22 and would have been added to $P_k^*(x,y)$ in Step 17 or 19, resulting in a contradiction to our assumption that $P_k^*(x,y)$ does not contain all the k SiSPs. Thus, $P_k^*(x,y)$ does contain the k^* SiSPs from x to y. □

The time bound for Algorithm COMPUTE-APSISP in Theorem 3.2.2 is established with the following sequence of simple lemmas.

Lemma 3.3.3. *There are $O(kn^2)$ paths in P_k^*, and $O(n^2)$ elements across all Extensions sets.*

Proof. $|P_k^*(x,y)| = O(kn^2)$ since there are at most k paths in each of the $n \cdot (n-1)$ sets $P_k^*(x,y)$. For the second part, exactly one edge is contributed to a Extensions set by each $P_k^*(x,y) \in \mathcal{U}$ in Step 7. □

Lemma 3.3.4. *Each $P_k^*(x,y)$ set is updated at most once in the main while loop.*

Proof. A path can be added to $P_k^*(x,y)$ at most once in Step 17 since its size will increase to k after the addition. Also, a path is added at most once in either Step 17 or Step 19 since paths are extracted from H in nondecreasing order of their weights. □

Lemma 3.3.5. *The number of k-LESiPs added to heap H is $O(n^2)$.*

Proof. For each k-LESiP, the right subpath must be the k-th shortest path in P_k^*. For each pair of vertices $x, y \in V$, there is at most one entry across the Extensions sets (say edge $(x,a) \in \mathit{Extensions}(a,y)$) and hence at most one k-LESiP will be added to heap H in Step 11 for pair (x,y). By lemma 3.3.4, we know that the set $P_k^*(a,y)$ is updated at most once and hence at most one k-LESiP will be added to heap H for pair (x,y) in Step 22. Thus, there are only $O(n^2)$ k-LESiPs that were added to the heap H in the algorithm. □

Lemma 3.3.6. *Algorithm* COMPUTE-APSISP *runs in $O(kn^2 + n^2 \log n)$ time.*

Proof. A binary heap suffices for H. The initialization for loop in Steps 3-11 takes $O(kn^2)$ time to initialize and inspect the P_k^* sets. It is executed at most n^2 times and, outside of the inspection of $P_k^*(x,y)$ an iteration costs $\Theta(\log n)$ time (cost for

insertion in heap), thus contributing $O(n^2 \log n)$ to the running time. The while loop is executed $O(n^2)$ times as by lemma 3.3.5, $O(n^2)$ elements are added to the heap. The extract-min operation takes $\Theta(\log n)$ time and hence Step 14 contributes $O(n^2 \log n)$ to the running time. Steps 15-19 takes constant time per iteration and hence add $O(n^2)$ to the total running time. By lemma 3.3.3, Step 22 is executed $O(n^2)$ times and contributes $O(n^2 \log n)$ to the running time. Thus, the total running time of the algorithm is $O(kn^2 + n^2 \log n)$. □

3.3.2 Computing the Q_k Sets

3.3.2.1 Computing Q_k for $k = 2$

We now give an $O(mn + n^2 \log n)$ time algorithm to compute $Q_2(x, y)$ for all pairs x, y. This method uses the procedure FAST-EXCLUDE from Demetrescu et al. [26], which we now describe (full details of this algorithm can be found in [26]).

Given a rooted tree T, edges (u_1, v_1) and (u_2, v_2) on T are *independent*[26] if the subtree of T rooted at v_1 and the subtree of T rooted at v_2 are disjoint. Given the weighted directed graph $G = (V, E)$, the SSSP tree T_s rooted at a source vertex $s \in V$, and a set S of independent edges in T_s, algorithm FAST-EXCLUDE in [26] computes, for each edge $e \in S$, a shortest path from s to every other vertex in $G - \{e\}$. This algorithm runs in time $O(m + n \log n)$.

We will compute the second path in each $Q_2(x, y)$ set, for a given $x \in V$, by running FAST-EXCLUDE with x as source, and with the set of outgoing edges from x in the shortest path tree rooted at x, T_x, as the set S. Clearly, this set S is independent, and hence algorithm FAST-EXCLUDE will produce its specified output. Now consider any vertex $y \neq x$, and let (x, a) be the first edge on the shortest path from x to y in T_x. By its specification, FAST-EXCLUDE will compute a shortest path from x to y that avoids edge (x, a) in its output, which is the second path needed for $Q_2(x, y)$. This holds for every vertex $y \in V - \{x\}$. Thus we have:

Lemma 3.3.7. *The $Q_2(x, y)$ sets for pairs x, y can be computed in $O(mn+n^2 \log n)$ time.*

This leads to the following algorithm for 2-APSiSP. Its time bound in Theorem 3.2.3, part (i) follows from Lemma 3.3.7 and the time bound for COMPUTE-APSiSP given in Section 3.3.1.

Algorithm 2 2-APSiSP$(G = (V, E); wt)$

1: **for** each $x \in V$ **do**
2: Compute a shortest path in each $Q_2(x, y)$, $y \in V - \{x\}$ (Dijkstra with source x)
3: Compute the second path in each $Q_2(x, y)$, $y \in V - \{x\}$, using FAST-EXCLUDE with source x and $S = \{(x, a) \in T_x\}$
4: COMPUTE-APSiSP(G, wt, 2, $\{Q_2(x, y), \forall x, y\}$)

The space bound is $O(n^2)$ since the Q_2 sets contain $O(n^2)$ paths and the call to COMPUTE-APSiSP takes $O(n^2)$ space.

Computing the Q_2 sets from distance sensitivity oracle. Let a DSO D with constant query time be given. For each $x, y \in V$, let π_{xy} be the shortest path from x to y. The second SiSP in $Q_2(x, y)$ is the shortest path from x to y avoiding the first edge on π_{xy}, so we can compute the second SiSP in $Q_2(x, y)$ by making $O(1)$ queries to D. Thus, $O(n^2)$ queries suffice to compute the second SiSP in all $Q_2(x, y)$ sets. A DSO with constant query time can be computed by a randomized algorithm in $O(n \log n \cdot (m + n \log n))$ time, and deterministically in $O(n \log^2 n \cdot (m + n \log n))$ time [17]. Since COMPUTE-APSiSP runs in $O(n^2 \log n)$, this gives a $\tilde{O}(mn)$ time algorithm for 2-APSiSP. It is not clear if we can efficiently compute 2-APSiSP directly from a DSO, without using the Q_2 sets and COMPUTE-APSiSP.

3.3.2.2 Computing Q_k for $k \geq 3$

Our algorithm will use the following types of sets. For each vertex $x \in V$, let I_x be the set of incoming edges to x. Also, for a vertex $x \in V$, and vertices $a, y \in V - \{x\}$, let $P_k^{*x}(a, y)$ be the set of k simple shortest paths from a to y in $G - I_x$, the graph obtained after removing the incoming edges to x. Recall that we maintain all P^* and Q sets as sorted arrays.

Algorithm APSiSP(G, k) first computes the sets $P_{k-1}^{*x}(a, y)$, for all vertices $a, y \in V$. Then it computes each $Q_k(x, y)$ as the set of all paths in the set $P_{k-1}^*(x, y)$, together with a shortest path in $\bigcup_{\{(x,a) \text{ outgoing from } x\}} \{(x, a) \circ p \mid p \in P_{k-1}^{*x}(a, y)\}$ (which is not present in $P_{k-1}^*(x, y)$).

Algorithm 3 APSiSP$(G = (V, E),\ wt,\ k)$

1: **if** $k = 2$ **then**
2: compute Q_2 sets using algorithm in Section 3.3.2.1
3: **else**
4: **for** each $x \in V$ **do**
5: $I_x \leftarrow$ set of incoming edges to x
6: Call APSiSP$(G - I_x, wt, k-1)$ to compute $P_{k-1}^{*x}(u, v)\ \forall u, v \in V$
7: **for** each $y \in V - \{x\}$ **do**
8: $Q_k(x, y) \leftarrow P_{k-1}^{*x}(x, y)$
9: **for all** $(x, a) \in E$ **do**
10: $count_a \leftarrow$ number of paths in $Q_k(x, y)$ with (x, a) as the first edge
11: Let $Z(x, y) = \bigcup_{\{(x,a) \text{ outgoing from } x\}} (x, a) \circ P_{k-1}^{*x}(a, y)[count_a + 1]$
12: $Q_k(x, y) \leftarrow Q_k(x, y) \cup \{$ a shortest path in $Z(x, y)\}$
13: Compute-APSiSP$(G, wt, k, \{Q_k(x, y)\ \forall x, y \in V\})$

To compute the P_{k-1}^{*x} sets, APSiSP(G, wt, k) recursively calls APSiSP$(G - I_x, wt, k-1)$ n times, for each vertex $x \in V$. Once we have computed the P_{k-1}^{*x} sets, the $Q_k(x, y)$ sets are readily computed as described in steps 8 - 12. After the computation of $Q_k(x, y)$ sets, APSiSP(G, wt, k) calls Compute-APSiSP$(G, wt, k, \{Q_k(x, y) \forall x, y \in V\})$ to compute the P_k^* sets, which is the output of the k-APSiSP problem. This establishes the following lemma and part (ii) of Theorem 3.2.3.

Lemma 3.3.8. *Algorithm* APSiSP (G, wt, k) *correctly computes the sets* $P_k^*(x, y)$ $\forall x, y \in V$.

Proof of Theorem 3.2.3, part (iii). The for loop starting in Step 4 is executed n times, and for $k = 3$ the cost of each iteration is dominated by the call to Algorithm 2-APSiSP in Step 6, which takes $O(mn + n^2 \log n)$ time. This contributes $O(mn^2 + n^3 \log n)$ to the total running time. The inner for loop starting in Step 7 is executed n times per iteration of the outer for loop, and the cost of each iteration is $O(k + d_x)$. Summing over all $x \in V$, this contributes $O(kn^2 + mn)$ to the total running time. Step 13 runs in $O(n^2 \log n)$ time as shown in Section 3.3.1. Thus, the total running time is $O(mn^2 + n^3 \log n)$. □

The space bound for APSiSP is $O(k^2 \cdot n^2)$, as the P_{k-1}^* and Q_k sets contain $O(kn^2)$ paths, and the recursive call to APSiSP$(G - I_x, wt, k-1)$ needs to maintain the P_{r-1}^* and Q_r sets at each level of recursion. The call to Compute-APSiSP takes $O(kn^2)$ space as noted earlier.

k-APSiSP. The performance of Algorithm APSiSP degrades by a factor of n with each increase in k. Thus, it matches Yen's algorithm (applied to all-pairs) for $k = 4$, and for larger values of k its performance is worse than Yen.

Since finding the P_k^* sets is at least as hard as finding the Q_k sets (as long as the running time is $\Omega(k \cdot n^2 + n^2 \log n)$), it is possible that the for loop starting in Step 4 could be replaced by a faster algorithm for finding the Q_k sets, which in turn would lead to a faster algorithm for k-APSiSP.

3.3.3 Generating k Simple Shortest Cycles

k-SiSC. This is the problem of generating the k simple shortest cycles through a specific vertex z in G. We can reduce this problem to k-SiSP by forming G'_z, where we replace vertex z by vertices z_i and z_o in G'_z, we place a directed edge of weight 0

from z_i to z_o, and we replace each incoming edge to (outgoing edge from) z with an incoming edge to z_i (outgoing edge from z_o) in G'_z. Then the k-th simple shortest path from z_o to z_i in G'_z can been seen to correspond to the k-th simple shortest cycle through z in G. This gives an $O(k \cdot (mn + n^2 \log\log n))$ time algorithm for computing k-SiSC using [41]. We also observe that we can solve k-SiSP from s to t in G if we have an algorithm for k-SiSC: create G' by adding a new vertex x^* and zero weight edges (x^*, s), (t, x^*), and then call k-SiSC for vertex x^*. Thus k-SiSP and k-SiSC are equivalent in complexity in weighted directed graphs.

k-ANSiSC. This is the problem of generating k simple shortest cycles that pass through a given vertex x, for every vertex $x \in V$. For $k = 1$ this problem can be solved in $O(mn + n^2 \log\log n)$ time by computing APSP [93]. For $k = 2$, we can reduce this problem to k-APSiSP by forming the graph G' where for each vertex x, we replace vertex x in G by vertices x_i and x_o in G', we place a directed edge of weight 0 from x_i to x_o, and we replace each edge (u, x) in G by an edge (u_o, x_i) in G' (and hence we also replace each edge (x, v) in G by an edge (x_o, v_i) in G'). For $k > 2$, k-ANSiSC can be computed in $O(k \cdot n \cdot (mn + n^2 \log\log n))$ time by computing k-SiSC for each vertex.This leads to the following theorem.

Theorem 3.3.9. *Let G be a directed graph with non-negative edge weights. Then,*

(i) k-SiSC can be computed in $O(k \cdot (mn + n^2 \log\log n))$ time, the same time as k-SiSP.

(ii) 2-ANSiSC can be computed in $O(mn + n^2 \log n)$ time, and for $k > 2$, k-ANSiSC can be computed in $O(k \cdot n \cdot (mn + n^2 \log\log n))$ time, the same time as n applications of k-SiSP.

3.4 Enumerating Simple Shortest Cycles (k-All-SiSC)

In this section we give a method to generate each successive simple shortest cycle in G (k-All-SiSC) in $\tilde{O}(m \cdot n)$ time. For enumerating simple paths in non-decreasing order of weight (k-All-SiSP), we give a faster method in Section 3.5 that uses again a path extension method, different from the one used in Section 3.3.1.

Let the input graph be $G = (V, E)$. From G we form the graph $G' = (V', E')$ as in the construction for k-ANSiSC in Section 3.3.3. We then proceed as follows. We assume the vertices are numbered 1 through n. Our algorithm for k-All-SiSC maintains an array $A[1..n]$, where each $A[j]$ contains a triple (ptr_j, w_j, k_j); here ptr_j is a pointer to the shortest cycle, not yet generated, that contains j as the minimum vertex (if such a cycle exists), w_j is the weight of this cycle, and k_j is the number of shortest simple cycles through vertex j that have already been generated. (Note that any given cycle is assigned to exactly one position in array A.)

Initially, we compute the entry for each $A[j]$ by running Dijkstra's algorithm with source j_o on the subgraph G'_j of G' induced on $V'_j = \{x_i, x_o \mid x \geq j\}$, to find a shortest path p from j_o to j_i; we then initialize $A[j]$ with a pointer to the cycle in G associated with p, and with its weight, and with $k_j = 0$.

For each $k \geq 1$, we generate the k-th simple shortest cycle in G by choosing a minimum weight cycle in array A. Let this entry be in $A[r]$ and let $\kappa = k_r$. We then compute the $(\kappa + 1)$-th shortest cycle through vertex r by computing the $(\kappa + 1)$-th shortest simple path from vertex r_o to vertex r_i in G' using a k-SiSP algorithm. The entry for $A[r]$ is now updated to a pointer to this newly computed simple cycle and its weight, and k_r is updated to $\kappa + 1$.

The time bound in Theorem 3.2.4 is seen by noting that the initialization takes $O(mn+n^2 \log n)$ for the n calls to Dijkstra's algorithm. Thereafter, we generate each new cycle in the slightly faster APSP time bound of $O(mn + n^2 \log\log n)$ with the k-SiSP algorithm in [41], by maintaining the relevant information from

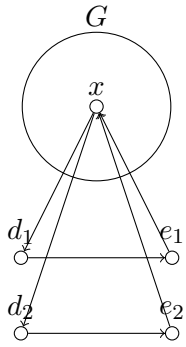


Figure 3.1: Construction of G' for $k = 3$ for Lemma 3.4.1.

the computation of earlier cycles. We now show that this time bound is optimal with respect to the Minimum Weight Cycle problem by showing that the problem of generating the k-th simple shortest cycle in a graph after the first $k-1$ cycles have been generated is at least as hard as the Min-Wt-Cyc problem.

Hardness Result. We establish a hardness result for the *k-th-All-SiSC* problem, which is the problem of computing the k-th simple shortest cycle in G after the $k-1$ simple shortest cycles in G have been computed (for any constant $k > 1$).

Lemma 3.4.1. *Min-Wt-Cyc* $\leq_{(m+n)}$ *k-th-All-SiSC.*

Proof. Suppose we are given an instance of the Min-Wt-Cyc, a directed graph $G = (V, E)$. Now we'll reduce this instance of the problem to that of computing k-th-All-SiSC in a weighted directed graph.

Now create a directed graph $G' = (V', E')$ such that it contains G as its subgraph and $2(k-1)$ additional vertices coming from the vertex partitions $D = \{d_i\}_{i=1}^{k-1}$ and $E = \{e_i\}_{i=1}^{k-1}$. Fix some $x \in V$. For each $1 \leq i \leq k-1$, add edges of weight 0 from x to d_i, from d_i to e_i and from e_i to x.

Figure 3.1 depicts the full construction of G' for $k = 3$.

Now the first $(k-1)$ min-weight cycles in G' correspond to the cycles involving vertices x, d_i and e_i (for each $1 \leq i \leq k-1$). And the k-th min-weight cycle in G'

corresponds to the minimum weight cycle in G.

As the number of vertices and edges in G' are linear in the number of vertices and edges, respectively, in G, we get the desired result. □

A similar algorithm can generate successive simple shortest paths. But in the next section, we present a faster algorithm for this problem.

3.5 Generating Simple Shortest Paths (k-All-SiSP)

Our algorithm for k-All-SiSP is inspired by the method in [25] for fully dynamic APSP. With each path π we will associate two sets of paths $L(\pi)$ and $R(\pi)$ as described below. Similar sets are used in [25] for 'locally shortest paths' but here they have a different use.

Left and right extensions. Let $\mathcal{P}$ be a collection of simple paths. For a simple path π_{xy} from x to y in $\mathcal{P}$, its left extension set $L(\pi_{xy})$ is the set of simple paths $\pi' \in \mathcal{P}$ such that $\pi' = (x', x) \circ \pi_{xy}$, for some $x' \in V$. Similarly, the right extension set $R(\pi_{xy})$ is the set of simple paths $\pi'' = \pi_{xy} \circ (y, y')$ such that $\pi'' \in \mathcal{P}$. For a trivial path $\pi = \langle v \rangle$, $L(\pi)$ is the set of incoming edges to v, and $R(\pi)$ is the set of outgoing edges from v.

Algorithm All-SiSP, generates all simple shortest paths in G in non-decreasing order of weight. To generate the k shortest simple paths in G, we can terminate the while loop after k iterations. Algorithm All-SiSP initializes a priority queue H with the edges in G, and it initializes the extension sets for the vertices in G. In each iteration of the main loop, the algorithm extracts the minimum weight path π in H as the next simple path in the output sequence. It then generates suitable extensions of π to be added to H as follows. Let the first edge on π be (x, a) and the last edge (b, y). Then, All-SiSP left extends π along those edges (x', x) such that there is a path $\pi_{x'b}$ in $L(l(\pi))$; it also requires that $x' \neq y$, since extending to

x' would create a cycle in the path. It forms similar extensions to the right in the for loop starting at Step 15.

Algorithm 4 ALL-SISP$(G = (V, E); wt)$

1: Initialization:
2: $H \leftarrow \phi$ $\{H$ is a priority queue.$\}$
3: **for all** $(x, y) \in E$ **do**
4: Add (x, y) to priority queue H with $wt(x, y)$ as key
5: Add (x, y) to $L(\langle y \rangle)$ and $R(\langle x \rangle)$
6: Main loop:
7: **while** $H \neq \phi$ **do**
8: $\pi \leftarrow$ EXTRACT-MIN(H)
9: Add π to the output sequence of simple paths
10: Let $\pi_{xb} = \ell(\pi)$ and $\pi_{ay} = r(\pi)$ (so (x, a) $((b, y))$ is the first (last) edge on π)
11: **for all** $\pi_{x'b} \in L(\pi_{xb})$ with $x' \neq y$ **do**
12: Form $\pi_{x'y} \leftarrow (x', x) \circ \pi$ and add $\pi_{x'y}$ to H with $wt(\pi_{x'y})$ as key
13: Add $\pi_{x'y}$ to $L(\pi_{xy})$ and to $R(\pi_{x'b})$
14: **for all** $\pi_{ay'} \in R(\pi_{ay})$ with $y' \neq x$ **do**
15: Perform steps complementary to Steps 12 and 13

We establish correctness in the following two lemmas, and then we prove Theorem 3.2.5.

Lemma 3.5.1. *Every path generated by Algorithm* ALL-SISP *is a simple path.*

Proof. Since edge weights are non-negative, the first path generated by Algorithm 4 is a minimum weight edge inserted in Step 4, which is a simple path. Assume the algorithm generates a path with a cycle, and let σ be the first path extracted in Step 8 that contains a cycle. Let (x', a) and (b, y) be the first and last edges on σ. Since σ contains a cycle, it contains at least two edges so (x', a) and (b, y) are distinct edges.

Consider the step when the non-simple path σ is placed on H. This does not occur in Step 4 since σ contains at least two edges. So σ is placed on H in some iteration of the while loop. Let π be the path extracted from H in this iteration; π

is a simple path by assumption since it was extracted from H before σ. Then σ is added to H either as a left extension of π (in Step 12) or as a right extension of π in a step complementary to Step 12 in the for loop in Step 15.

Consider the left extension case, and let σ be formed when processing path $\pi_{x'b} \in L(l(\pi))$ with $x' \neq y$ in Step 11. Thus σ is formed as $(x', x) \circ \pi$ in Step 12. But $(x', x) \circ \pi = (x', x) \circ \ell(\pi) \circ (b, y) = \pi_{x'b} \circ (b, y)$. Since $\pi_{x'b} \in L(l(\pi))$, it was also placed in H in either Step 4 or Step 12. And as $wt(\pi_{x'b}) < wt(\sigma)$, the path $\pi_{x'b}$ is simple. Since $\pi_{x'b}$ is simple, a cycle can be formed in σ only if $x' = y$. But this is specifically forbidden in the condition in Step 11. A similar argument applies to right extensions added to H in Step 15. Hence σ is a simple path, and Algorithm 4 does not generate any path containing a cycle. □

Lemma 3.5.2. *Algorithm* All-SiSP *generates all simple paths in G in non-decreasing order of their weights.*

Proof. Clearly the algorithm correctly generates the minimum weight edge in G as the minimum weight simple path in the output in the first iteration of the while loop. By Lemma 3.5.1 all generated paths are simple. Also, these simple paths are generated in non-decreasing order of weight since any path added to H in Steps 12 and 15 has weight at least as large as the weights of the paths that have been extracted at that time, due to non-negative edge-weights. It remains to show that no simple path in G is omitted in the sequence of simple paths generated.

Suppose the algorithm fails to generate all simple shortest paths in G and let π be a simple path of smallest weight that is not generated by Algorithm 4. Let π be a path with first edge (x, a) and last edge (b, y); $(x, a) \neq (b, y)$ since all single edge paths is added to H in Step 4, and will be extracted in a future iteration. Let π_{ab} be the subpath of π from a to b. By assumption, the paths $\pi_{xb} = \ell(\pi)$ and $\pi_{ay} = r(\pi)$ are placed in the output by Algorithm 4 since they are simple paths with

weight smaller than the weight of π. Without loss of generality assume that π_{xb} was extracted from H before π_{ay}.

Clearly, π_{xb} was inserted in H before π_{ay} was extracted. In the iteration of the while loop when π_{xb} was added to H, π_{xb} was added to $L(\pi_{ab})$ in Step 13 since $r(\pi_{xb}) = \pi_{ab}$. In the later iteration when π_{ay} was extracted from H, the paths in $L(\ell(\pi_{ay}))$ are considered in Step 12. But $\ell(\pi_{ay}) = \pi_{ab}$. When the paths in $L(\ell(\pi_{ay})) = L(\pi_{ab})$ are considered in Step 11 during the processing of π_{ay}, the path π_{xb} will be one of the paths processed, and in Step 12 the path $(x, a) \circ \pi_{ay} = \pi$ will be formed and added to H. Thus π will be added to H, and hence will be extracted and added to the output sequence. □

Proof of Theorem 3.2.5. We will maintain paths with pointers to their left and right subpaths, so each path takes $O(1)$ space. For the amortized bound we will implement H as a Fibonacci heap. The initialization takes $O(m)$ time. Each L and R set can contain at most $n-2$ paths, and further, since extensions are formed only with paths already in H, each of these sets has size $\min\{k, n-2\}$. The k-th iteration of the while loop takes time $O(\log |H|)$ for the extract-min operation, and $O(\min\{k, n\})$ time for the processing of the L and R sets. At the start of the k-th iteration, the number of paths in H is at most $O(m+k\cdot\min\{k, n\})$, and since $m = O(n^2)$, $\log |H| = O(\log(n+k))$. Hence the amortized time for the k-th iteration is $O(\min\{k, n\} + \log(n + k))$.

For the worst-case bound we will use a binary heap. Then, the initialization takes $O(m)$ time to build a heap on the m edges, and the k-th iteration costs $O(\min\{k, n\} \cdot \log(n + k))$ for the heap operations. □

3.6 Conclusion and Open Problems

We have presented a new algorithm for the problem of generating k simple shortest paths for every pair of vertices in a weighted directed graph (k-APSiSP). This

algorithm is of special interest since it is the first algorithm that does not use the 'detour finding' technique for computing multiple simple shortest paths. In fact, all previous algorithms known for finding multiple simple shortest paths, replacements paths, and distance sensitivity oracles find the solution by computing 'detours'. In contrast, we have introduced a novel path extension method. We have then considered the problem of enumerating simple cycles in the graph in non-decreasing order of their weights (k-All-SiSC), and we have given an algorithm that generates each successive simple cycle in $\tilde{O}(mn)$ time. Finally, we have used a different path extension technique to obtain a very efficient algorithm to generate the k simple shortest paths in the entire graph (k-All-SiSP).

Our k-All-SiSP algorithm is nearly optimal if the paths need to be output. It is also not difficult to see that our bounds for 2-APSiSP and k-All-SiSC (for constant k) are the best possible to within a polylog factor for sparse graphs unless the long-standing $\tilde{O}(mn)$ bounds for APSP and minimum weight cycles are improved. In recent work [6] we give several fine-grained reductions that demonstrate that the minimum weight cycle problem holds a central position for a class of problems that currently have $\tilde{O}(mn)$ time bound on sparse graphs, both directed and undirected.

For undirected graphs, our k-All-SiSP algorithm gives an algorithm with the same bound. Also, our k-APSiSP algorithm works for undirected graphs, and this gives a faster algorithm for $k = 2$ and matches the previous best bound for $k = 3$ (using [57]). However, our algorithms for the three variants of finding simple shortest cycles do not work for undirected graphs. This is addressed in the work presented in Chapter 2, where the fine-grained reductions also give new algorithms for finding shortest cycles in undirected graphs.

We conclude with two avenues for further research.

1. The main open question for k-APSiSP is to come up with faster algorithms to compute the $Q_k(x, y)$ sets for larger values of k. This is the key to a faster k-

APSiSP algorithm using our approach, for $k > 2$.

2. The space requirements of algorithms are high. Can we come up with space-efficient algorithms that match our time bounds?

Part II

Distributed Results

Chapter 4

Distributed Weighted All Pairs Shortest Paths

4.1 Introduction

In the previous chapters (Chapter 2 and 3) we studied the shortest path problems in the sequential setting. In Chapters 5, 6 and 7, we study the shortest paths problem in the distributed setting, specifically the weighted all pairs shortest path (APSP) problem.

The design of distributed algorithms for various network (or graph) problems such as shortest paths [65, 70, 28, 50] and minimum spanning tree [36, 75, 38, 59] is a well-studied area of research. The most widely considered model for studying distributed algorithms is the CONGEST model [73] (also see [28, 50, 49, 65, 70, 39]), described in more detail below in Section 4.2. In Chapters 5, 6 and 7, we consider the problem of computing all pairs shortest paths (APSP) in a weighted directed (or undirected) graph in this model.

The problem of computing all pairs shortest paths (APSP) in distributed networks is a very fundamental problem, and there has been a considerable line of

work for the CONGEST model as described later. However, for a weighted graph no deterministic algorithm was known in this model other than a trivial method that runs in n^2 rounds. In Chapter 5 we present the first algorithm for this problem in the CONGEST model that computes weighted APSP deterministically in less than n^2 rounds. Our algorithm computes APSP deterministically in $O(n^{3/2} \cdot \sqrt{\log n})$ rounds in this model in both directed and undirected graphs. We follow up on this result with an improved $\tilde{O}(n^{4/3})$ rounds deterministic APSP algorithm described in Chapter 7. In Chapter 6 we present a deterministic APSP algorithm that improves the round complexity for moderate integer edge weights.

Our APSP algorithms in Chapters 5, 6 and 7 follows the general 3-phase strategy initiated by Ullman and Yannakakis [87] for parallel computation of path problems in directed graphs:

1. Compute h-hop shortest paths for each source for a suitable value of h. (An h-hop path is a path that contains at most h edges.)

2. Find a small *blocker set* Q that intersects all paths computed in Step 1. (With randomization, this step is very simple: a random sample of the vertices of size $O((n/h) \cdot \log n)$ satisfies this property w.h.p. in n.)

3. Compute shortest paths between all pairs of vertices in Q, and using this information and the h-hop trees from Step 1, compute the APSP output at each node in V.

CONGEST directed APSP algorithms that fall in this framework include the randomized algorithm in Huang et al. [50] that runs in $\tilde{O}(n^{5/4})$ rounds for polynomial integer edge-weights, the deterministic algorithm presented in this chapter for arbitrary edge-weights, and the deterministic algorithm in Chapter 6 that improves on the result in this chapter for moderate integer edge-weights, and the algorithm in Chapter 7 that improves the APSP round complexity for arbitrary edge weights.

We now describe the CONGEST model for which we propose our APSP algorithms.

4.2 The CONGEST Model

In the CONGEST model [73], there are n independent processors interconnected in a network. We refer to these processors as nodes. These nodes are connected in the network by bounded-bandwidth links which we refer to as edges. The network is modeled by a graph $G = (V, E)$ where V is the set of processors and E is the set of edges or links between these processors. Here $|V| = n$ and $|E| = m$.

Each node is assigned a unique ID between 1 and $poly(n)$ and has infinite computational power. Each node has limited topological knowledge and only knows about its incident edges. For the weighted APSP problem we consider, each edge has a positive integer weight bounded by $poly(n)$. Also if the edges are directed, the corresponding communication channels are bidirectional and hence the communication network can be represented by the underlying undirected graph U_G of G (this model is also used in [50, 47, 40]).

The computation proceeds in rounds. In each round each processor can send a message of size $O(\log n)$ along edges incident to it, and it receives the messages sent to it in the previous round. The model allows a node to send different message along different edges though we do not need this feature in our algorithm. The performance of an algorithm in the CONGEST model is measured by its round complexity, which is the worst-case number of rounds of distributed communication. Hence the goal is to minimize the round complexity of an algorithm.

4.3 Related Work

We compare our results for distributed APSP with other results in Table 4.1.

Table 4.1: Table comparing our results for non-negative edge-weighted graphs (including zero edge weights) with previous known results. Here W is the maximum edge weight and Δ is the maximum weight of a shortest path in G. Arb. stands for arbitray edge weights and Int. stands for integer edge weights. Rand. stands for randomized algorithm and Det. stands for deterministic algorithm. Dir. stands for directed graphs and Undir. stands for undirected graphs.

Problem: Exact Weighted APSP				
Author	Arb./ Int. weights	Rand. / Det.	Undir. / (Dir. & Undir.)	Round Complexity
Huang et al. [50]	Int.	Rand.	Dir. & Undir.	$\tilde{O}(n^{5/4})$
Elkin [28]	Arb.	Rand.	Undir.	$\tilde{O}(n^{5/3})$
Our Result [10] (Ch. 5)	**Arb.**	**Det.**	**Dir. & Undir.**	$\tilde{\mathbf{O}}(\mathbf{n^{3/2}})$
Our Result [8] (Ch. 6)	**Int.**	**Det.**	**Dir. & Undir.**	$\tilde{\mathbf{O}}(\mathbf{n^{3/2-\epsilon/4}})$ **(when** $\mathbf{W \leq n^{1-\epsilon}}$**)** $\tilde{\mathbf{O}}(\mathbf{n^{3/2-\epsilon/3}})$ **(when** $\mathbf{\Delta \leq n^{3/2-\epsilon}}$**)**
	Arb.	**Rand.**	**Dir. & Undir.**	$\tilde{\mathbf{O}}(\mathbf{n^{4/3}})$
Bernstein & Nanongkai [18]	Arb.	Rand.	Dir. & Undir.	$\tilde{O}(n)$
Our Result [9] (Ch. 7)	**Arb.**	**Det.**	**Dir. & Undir.**	$\tilde{\mathbf{O}}(\mathbf{n^{4/3}})$

Prior Work.

Unweighted APSP. For APSP in unweighted undirected graphs, $O(n)$-round algorithms were given independently in [49, 74]. An improved $n+O(D)$-round algorithm was then given in [65], where D is the diameter of the undirected graph. Although this latter result was claimed only for undirected graphs, the algorithm in [65] is also a correct $O(n)$-round APSP algorithm for directed unweighted graphs. The message complexity of directed unweighted APSP was reduced to $mn + O(m)$ in a recent algorithm [47] that runs in $\min\{2n, n + O(D)\}$rounds (where D is now the directed diameter of the graph). A lower bound of $\Omega(n/\log n)$ for the number of rounds needed to compute the diameter of the graph in the CONGEST model is given in [33].

Weighted APSP. While unweighted APSP is well-understood in the CONGEST model much remains to be done in the weighted case. For deterministic algorithms, weighted SSSP for a single source can be computed in n rounds using the classic Bellman-Ford algorithm [15, 32], and this leads to a simple deterministic weighted APSP algorithm that runs in $O(n^2)$ rounds. Nothing better was known for the number of rounds for deterministic weighted APSP until our current results.

Exact Randomized APSP Algorithms. Even with randomization, nothing better than n^2 rounds was known for exact weighted APSP until recently, when Elkin [28] gave a randomized weighted APSP algorithm that runs in $\tilde{O}(n^{5/3})$ rounds for graphs with arbitrary edge weights and this was further improved to $\tilde{O}(n^{5/4})$ rounds in Huang et al. [50] for integer edge weights. Recently Bernstein and Nanongkai [19] gave a $\tilde{O}(n)$ rounds randomized APSP algorithm for graphs with arbitrary edge weights. This result subsumes both of these previous results. All of these results hold with high probability in n.

Derandomizing Distributed Algorithms. Censor-Hillel et al. [23] semi-formalized a template of combining bounded independence with the method of conditional ex-

pectation for derandomizing an algorithm for computing Maximal Independent Set (MIS) in the distributed setting. In Section 7.3.2 we instead use a linear-sized sample space for generating pairwise independent random variables and then use an aggregration of suitable parameters of sample point values to derandomize our randomized blocker set algorithm.

Deterministic Approximation Algorithms for APSP. There are deterministic algorithms for approximating weighted all pairs shortest path problem, and these run in $\tilde{O}(n)$ rounds for both directed [63] and undirected graphs [63, 44, 29].

Chapter 5

Deterministic Distributed All Pairs Shortest Paths in $\tilde{O}(n^{3/2})$ Rounds

5.1 Introduction

In this Chapter we describe our $\tilde{O}(n^{3/2})$ rounds deterministic weighted APSP algorithm in the CONGEST model. Our distributed APSP algorithm is quite simple and we give an overview in Section 5.2. It uses the notion of a blocker set introduced by King [58] in the context of sequential fully dynamic APSP computation. Our deterministic distributed algorithm for computing a blocker set is the most nontrivial component of our algorithm, and is described in Section 5.3.

5.2 Overview of the APSP Algorithm

Let $G = (V, E)$ be an edge-weighted graph (directed or undirected) with weight function w and with $|V| = n$ and $|E| = m$. The CONGEST model assumes that every

message is of $O(\log n)$-bit size, which restricts $w(e)$ to be an $O(\log n)$ size integer value. However, outside of this restriction imposed by the CONGEST model, our algorithm works for arbitrary edge-weights (even negative edge-weights as long as there is no negative-weight cycle). Given a path p we will use *weight* or *distance* to denote the sum of the weights of the edges on the path and *length* (or sometimes *hops*) to denote the number of edges on the path. We denote the shortest path distance from a vertex x to a vertex y in G by $\delta(x, y)$. In the following we will assume that G is directed, but the same algorithm works for undirected graphs as well.

An h-hop shortest path from a source s to a vertex v is the minimum weight path from s to v with at most h hops. In the case of multiple paths with the same weight from s to v we assume that v chooses the path with its parent vertex of minimum id. We will use $h = \sqrt{n \cdot \log n}$ in our algorithm.

Our overall APSP algorithm is given in Algorithm 1.

Algorithm 1 Overall APSP algorithm

Input: set of sources S, number of hops h

1: **For each** $x \in S$ **in sequence:** Compute h-hop shortest paths starting from source x.
2: Compute a blocker set Q of size $\Theta(\frac{n \log n}{h})$ for the h-hop shortest paths computed in Step 1 (described in Section 5.3).
3: **for each** $c \in Q$ **in sequence:** compute SSSP tree rooted at c.
4: **for each** $c \in Q$ **in sequence:** broadcast $ID(c)$ and the shortest path distance values $\delta_h(x, c)$ for each $x \in S$.
5: **Local Step at node** $v \in V$**:** for each $x \in S$ compute the shortest path distance $\delta(x, v)$ using the received values.

In Step 1 the h-hop SSSPs along with the h-hop shortest path distances, $\delta_h(x, v)$, are computed at every vertex v for each source $x \in V$. These paths can be easily converted to form a rooted tree at x by first computing $2h$-hop shortest paths and then just extracting out the first h-hop paths.

Step 2 computes a *blocker set* Q of $q = \Theta((n \log n)/h)$ nodes for the collection

of h-hop SSSPs constructed in Step 1. This step is described in detail in Section 5.3, where we describe a distributed implementation of King's sequential method [58]. Our method computes the blocker set Q in $O(nh + (n^2 \log n)/h)$ rounds. We now give the definition of a blocker set for a collection of rooted h-hop trees.

Definition 5.2.1 (**Blocker Set** [58])**.** *Let H be a collection of rooted h-hop trees in a graph $G = (V, E)$. A set $Q \subseteq V$ is a* blocker set *for H if every root to leaf path of length h in every tree in H contains a vertex in Q. Each vertex in Q is called a* blocker vertex *for H.*

In Step 3 of Algorithm 1 we compute $\delta(c, v)$ for each $c \in Q$ and for all $v \in V$. In Step 4 each blocker vertex c broadcasts all of the $\delta_h(x, c)$ values, for each source $x \in S$, it computed in Step 1. Finally, in Step 5 each node v computes $\delta(x, v)$ for each $x \in S$ using the values it computed or received in the earlier steps. More specifically, v computes $\delta(x, v)$ as:

$$\delta(x, v) = \min \left\{ \delta_h(x, v), \ \min_{c \in Q} (\delta_h(x, c) + \delta(c, v)) \right\} \tag{5.1}$$

Lemma 5.2.2. *The $\delta(x, v)$ values computed at each v in Step 5 of Algorithm 1 are the correct shortest path distances.*

Proof. Fix vertices x, v and consider a shortest path p from x to v. If p has at most h edges then $w(p) = \delta_h(x, v)$ and this value is directly computed at v in Step 1. Otherwise by the property of the blocker set Q we know that there is a vertex $c \in Q$ which lies along p within the h-hop SSSP tree rooted at x that is constructed in Step 1. Let p_1 be the portion of p from x to c and let p_2 be the portion from c to v. So $w(p_1) = \delta_h(x, c)$, $w(p_2) = \delta(c, v)$ and $w(p) = w(p_1) + w(p_2)$.

The value $\delta_h(x, c)$ is received by v in the broadcast step for center c in Step 4. The value $\delta(c, v)$ is computed at v when SSSP with root c is computed in Step 3. Hence v has the information needed to compute $\delta(x, v)$ in Step 5 for each x using

Equation 5.1. □

We now bound the number of rounds needed for each step in Algorithm 1 (other than Step 2). For this we first state bounds for some simple primitives that will be used to execute these steps.

Lemma 5.2.3. *Given a source $s \in V$, using the Bellman-Ford algorithm:*

(a) the shortest path distance $\delta(s, v)$ can be computed at each $v \in V$ in n rounds.

(b) the h-hop shortest path distance $\delta_h(s, v)$ can be computed at each $v \in V$ in h rounds.

Lemma 5.2.4. *A node v can broadcast k local values to all other nodes reachable from it deterministically in $O(n + k)$ rounds.*

Proof. We construct a BFS tree rooted at v in at most n rounds and then we pipeline the broadcast of the k values. The root v sends the i-th value to all its children in round i for $1 \leq i \leq k$. In a general round, each node x that received a value in the previous round sends that value to all its children. It is readily seen that the i-th value reaches all nodes at hop-length d from v in the BFS tree in round $i + d - 1$, and this is the only value that node x receives in this round. □

Lemma 5.2.5. *All $v \in V$ can broadcast a local value to every other node they can reach in $O(n)$ rounds deterministically.*

Proof. This broadcast can be done in $O(n)$ rounds in many ways, for example by piggy-backing on an $O(n)$ round unweighted APSP algorithm [65, 47] (and also [49, 74] for undirected graphs) where now each message contains the value sent by source s in addition to the current shortest path distance estimate for source s. □

Lemma 5.2.6. *Algorithm 1 runs in $O(n \cdot h + (n^2/h) \cdot \log n)$ rounds assuming Step 2 can be implemented to run within this bound.*

Proof. Let the size of the blocker set be $q = \frac{n}{h} \cdot \log n$. Using part (b) of Lemma 5.2.3 and Lemma 6.4.2, Step 1 can be computed in $O(n \cdot h)$ rounds. Step 3 can be computed in $O(n \cdot q) = O((n^2/h) \cdot \log n)$ rounds by part (a) of Lemma 5.2.3. Step 4 can be computed in $O(n \cdot q) = O((n^2/h) \cdot \log n)$ rounds by Lemma 5.2.4 (using $k = n$). Finally, Step 5 involves only local computation and no communication. This establishes the lemma. □

In Section 5.3 we give a description of our deterministic algorithm to compute Step 2 in $O(n \cdot h + (n^2/h) \cdot \log n)$ rounds, which leads to our main theorem (by using $h = \sqrt{n \cdot \log n}$).

Theorem 5.2.7. *Algorithm 1 is a deterministic distributed algorithm for weighted APSP in directed or undirected graphs that runs in $O(n^{3/2} \cdot \sqrt{\log n})$ rounds in the* Congest *model.*

5.3 Computing Blocker Set Deterministically

The simplest method to find a blocker set is to chose the vertices randomly. An early use of this method for path problems in graphs was in Ullman and Yannakakis [87] where a random set of $O(\sqrt{n} \cdot \log n)$ distinguished nodes was picked. It is readily seen that some vertex in this set will intersect any path of $O(\sqrt{n})$ vertices in the graph (and so this set would serve as a blocker set of size $O((n \log n)/h)$for our algorithm if $h = \sqrt{n}$). Using this observation an improved randomized parallel algorithm (in the PRAM model) was given in [87] to compute the transitive closure. Since then this method of using random sampling to choose a suitable blocker set has been used extensively in parallel and dynamic computation of transitive closure and shortest paths, and more recently, in distributed computation of APSP [50].

It is not clear if the above simple randomized strategy can be derandomized in its full generality. However, for our purposes a blocker set only needs to intersect

all paths in the set of hop trees we construct in Step 1 of Algorithm 1. For this, a deterministic sequential algorithm for computing a blocker set was given in King [58] in order to compute fully dynamic APSP. This algorithm computes a blocker set of size $O((n/h)\ln p)$ for a collection F of h-hop trees with a total of p leaves across all trees (and hence p root to leaf paths) in an n-node graph. In our setting $p \leq n^2$ since we have n trees and each tree could have up to n leaves.

King's sequential blocker set algorithm uses the following simple observation: Given a collection of p paths each with exactly h nodes from an underlying set V of n nodes, there must exist a vertex that is contained in at least ph/n paths. The algorithm adds one such vertex v to the blocker set, removes all paths that are covered by this vertex and repeats this process until no path remains in the collection. The number of paths is reduced from p to at most $(1 - h/n) \cdot p$ when the blocker vertex v is removed, hence after $O((n/h)\ln p)$ removals of vertices, all paths are removed. Since p is at most n^2 the size of the blocker set is $O((n\log n)/h)$. King's sequential algorithm for finding a blocker set runs in $O(n^2 \log n)$ deterministic time.

We now describe our distributed algorithm to compute a blocker set. As in King [58], for each vertex v in a tree T_x in the collection of trees H we define:

- $score_x(v)$ is the number of leaves at depth h in T_x that are in the subtree rooted at v in T_x;
- $score(v) = \sum_x score_x(v)$.

Thus, $score(v)$ is the number of root-to-leaf length paths of length h in the collection of trees H that contain vertex v. Initially, our distributed algorithm computes all $score_x(v)$ and $score(v)$ for all vertices $v \in V$ and all h-hop trees T_x in $O(n \cdot h)$ rounds. Then through an all-to-all broadcast of $score(v)$ to all other nodes for all v, all nodes identify the vertex c with maximum score as the next blocker vertex to be removed from the trees and added to the blocker set Q. (In case there are multiple vertices

with the maximum score the algorithm chooses the vertex of minimum id having this maximum score. This ensures that all vertices will locally choose the same vertex as the next blocker vertex once they have received the scores of all vertices.) We repeat this process until all scores are zeroed out. By the discussion above (and as observed in [58]) we will identify all the vertices in Q in $O((n \cdot \log n)/h)$ repeats of this process.

What remains is to obtain an $O(n)$ round procedure to update the *score* and $score_x$ values at all nodes each time a vertex c is removed so that we have the correct values at each node for each tree when the leaves covered by c are removed from the tree.

If a vertex v is a descendant of the removed vertex c in T_x then all paths in T_x that pass through v are removed when c is removed and hence $score_x(v)$ needs to go down to zero for each such tree T_x where v is a descendant of the chosen blocker node c. In order to facilitate an $O(n)$-round computation of these updated $score_x$ values in each tree at all nodes that are descendants of c, we initially precompute at every node v a list $Anc_x(v)$ all of its ancestors in each tree T_x. This is computed in $O(n \cdot h)$ rounds using our ANCESTORS algorithm (Algorithm 4). Thereafter, each time a new blocker vertex c is selected to be removed from the trees and added to Q, it is a local computation at each node v to determine which of the $Anc_x(v)$ sets at v contain c and to zero out $score_x(v)$ for each such x.

The other type of vertices whose scores change after a vertex c is removed are the ancestors of c in each tree. If v is an ancestor of c in T_x then after c is removed $score_x(v)$ needs to be reduced by $score_x(c)$ (i.e., c's score before it was removed and added to Q) since these paths no longer need to be covered by v. For these ancestor updates we give an $O(n)$-round algorithm that runs after the addition of each new blocker node to Q and correctly updates the scores for these ancestors in every tree. (Algorithm 6). These algorithms together give the overall deterministic algorithm

(Algorithm 2) for the computation of the blocker set Q in $O(n \cdot h + (n^2 \log n)/h)$ rounds. We present an improved blocker set algorithm that runs in $O(nq + \sqrt{\Delta hk})$ rounds in Section 6.3 and another one that runs in $\tilde{O}(nh)$ rounds in Section 7.3.

We now give the details of our algorithms. Recall that we use the h-hop CSSSP algorithm (described in Section 6.4) for Step 1 in Algorithm 1. Hence after that step, for each tree T_s rooted at s every node v in the tree knows its shortest path distance from s, $\delta(s, v)$, its hop length $h_s(v)$ and its parent node in T_s. We also determine for each node its children in T_s. We can compute this in one round for each T_s by have each node send its child status to its parent. Thus after n rounds all nodes know all their children in every tree T_s.

5.3.1 The Blocker Set Algorithm

Algorithm 2 COMPUTE-BLOCKER

Input: h-hop CSSSP Collection of all h-hop trees T_x; Output: set Q

1: **Initialization [lines 2-6]:**
2: Run Algorithm 3 to compute scores for all $v \in V$
3: For each T_x compute the ancestors of each vertex v in T_x in $Anc_x(v)$ using Algorithm 4
4: **for** each $v \in V$ **do**
5: **Local Step:** $score(v) \leftarrow \sum_{x \in V} score_x(v)$
6: broadcast $score(v)$ to all nodes in V (using Lemma 5.2.5)
7: **Add blocker vertices to blocker set Q [lines 8-12]:**
8: **while** there is a node c with $score(c) > 0$ **do**
9: **for** each $v \in V$ **do**
10: **Local Step:** select the node c with max score as next vertex in Q
11: Run Algorithms 5 and 6 to update $score_x(v)$ for each $x \in V$ and $score(v)$
12: broadcast $score(v)$ to all nodes in V and receive $score(x)$ from all other nodes x

Algorithm 2 gives our distributed deterministic method to compute a blocker set. It uses a collection of helper algorithms that are described in the next section. This blocker set algorithm is at the heart of our main algorithm (Algorithm 1, Step 2)

for computing the exact weighted APSP.

Step 2 of Algorithm 2 executes Algorithm 3 to compute all the initial scores at all nodes v. Step 3 involves running Algorithm 4 for pre-computing ancestors of each node in every T_x. Step 5 is a local computation (no communication) where all nodes v compute their total score by summing up the scores for all trees T_x to which they belong. And in Step 6, each node v broadcasts its score value to all other nodes.

The while loop in Steps 8-12 of Algorithm 2 runs as long as there is a node with positive score. In Step 10, the node with maximum score is selected as the vertex c to be added to Q (and if there are multiple nodes with the maximum score, then among them the node with the minimum ID is selected, so that the same node is selected locally at every vertex). In Step 11, after blocker vertex c is selected, each node v checks whether it is a descendant of c in each T_x and if so update its score for that tree using Algorithm 5. This is followed by an execution of Algorithm 6 which updates the scores at each node v for each tree T_x in which v is an ancestor of c. Then in Step 12, all the nodes broadcast their score to all other nodes so that they can all select the next vertex to be added to Q. This leads to the following lemma, assuming the results shown in the next section.

Lemma 5.3.1. *Algorithm 2 correctly computes the blocker set Q in $O(n \cdot h + n \cdot |Q|)$ rounds.*

Proof. Step 2 runs in $O(n \cdot h)$ rounds (by Lemma 5.3.2) and so does Step 3 (see Lemma 5.3.3). Step 5 is a local computation and the broadcast in Step 6 runs in $O(n)$ rounds by Lemma 5.2.5.

The while loop starting in Step 8 runs for $|Q|$ iterations since a new blocker vertex is added to Q in each iteration. In each iteration, Step 10 is a local computation as is the execution of Algorithm 5 in Step 11. Algorithm 6 in Step 11 runs in $O(n)$ rounds (Lemma 5.3.6). The all-to-all broadcast in Step 12 is the same as the

initial all-to-all broadcast in Step 6 and runs in $O(n)$ rounds. Hence each iteration of the while loop runs in $O(n)$ rounds giving the desired bound. □

5.3.2 Algorithms for Computing and Updating Scores

In this section we give the details of our algorithms for computing initial scores (Algorithm 3) and for updating these scores values once a blocker vertex c is selected and added to the blocker set Q (Algorithms 4-6).

Algorithm 3 Compute Initial scores for a node v in T_x

1: **Initialization [Local Step]: if** $h_x(v) = h$ **then** $score_x(v) \leftarrow 1$ **else** $score_x(v) \leftarrow 0$
2: **In round** $r > 0$**:**
3: **send: if** $r = h - h_x(v) + 1$ **then** send $\langle score_x(v) \rangle$ to $parent_x(v)$
4: **receive [lines 5-9]:**
5: **if** $r = h - h_x(v)$ **then**
6: let $\mathcal{I}$ be the set of incoming messages to v
7: **for each** $M \in \mathcal{I}$ **do**
8: let $M = \langle score^- \rangle$ and let the sender be w
9: **if** w is a child of v in T_x **then** $score_x(v) \leftarrow score_x(v) + score^-$

Algorithm 3 gives the procedure for computing the initial scores for a node v in a tree T_x. In Step 1 each leaf node at depth h initializes its score for T_x to 1 and all other nodes set their initial score to 0. In a general round $r > 0$, nodes with $h_x(v) = h+1-r$ send out their scores to their parents and nodes with $h_x(v) = h-r$ will receive all the scores from its children in T_x and set its score equal to the sum of these received scores (Steps 5-9).

Lemma 5.3.2. *Algorithm 3 computes the initial scores for every node v in T_x in $O(h)$ rounds.*

Proof. The leaves at depth h correctly initialize their score to 1 locally in Step 1. Since we only consider paths of length h from the root x to a leaf, it is readily seen that a node v that is $h_x(v)$ hops away from x in T_x will receive scores from its

children in round $h - h_x(v)$ and thus will have the correct $score_x(v)$ value to send in Step 3. □

For every $x \in V$, every node $v \in T_x$ will run this algorithm to compute their score in T_x. Since every run of Algorithm 3 for a given x takes h rounds, all the initial scores can be computed in $O(n \cdot h)$ rounds.

Algorithm 4 ANCESTORS (v, x): Algorithm for computing ancestors of node v in T_x at round r

```
1: Initialization [Local Step]: Anc_x(v) ← φ
2: In round r > 0:
3: send [lines 4-8]:
4: if r = 1 then
5:     send ⟨v⟩ to v's children in T_x
6: else
7:     let ⟨y⟩ be the message v received in round r − 1
8:     send ⟨y⟩ to v's children in T_x
9: receive [lines 10-11]:
10: let ⟨y⟩ be the message v received in this round
11: add y to Anc_x(v)
```

Algorithm 4 describes our algorithm for precomputing the ancestors of each node v in a tree T_x of height h. In round 1, every node v sends its ID to its children in T_x as described in Step 5. And in a general round r, v sends the ID of the ancestor that it received in round $r-1$ (Steps 7-8). If a node v receives the ID of an ancestor y, then it immediately adds it to its ancestor set, $Anc_x(v)$ (Steps 10-11).

Lemma 5.3.3. *For a tree T_x of height h rooted at vertex x, Algorithm 4 correctly computes the set of ancestors for all nodes v in T_x in $O(h)$ rounds.*

Proof. We show that all nodes v correctly computes all their ancestors in T_x in the set $Anc_x(v)$ using induction on round r. We show that by round r, every node v has added all its ancestors that are at most r hops away from v.

If $r = 1$, then v's parent in T_x (say y) would have send out its ID to v in Step 5 and v would have added it to $Anc_x(v)$ in Step 11.

Assume that every node v has already added all ancestors in the set $Anc_x(v)$ that are at most $r-1$ hops away from v.

Let u be the ancestor of v in T_x that is exactly r hops away from v. Then by induction, $u \in Anc_x(y)$ since u is exactly $r-1$ hops away from y and thus y must have send u's ID to v in round r in Step 8 and hence v would have added u to its set $Anc_x(v)$ in round r in Step 11. □

Once we have pre-computed the $Anc_x(v)$ sets for all vertices v and all trees T_x using Algorithm 4, updating the scores at each node for all trees in which it is a descendant of the newly chosen blocker node c becomes a purely local computation. Algorithm 5 describes the algorithm at node v that updates its scores after a vertex c is added as a blocker node to Q. At node v for each given T_x, v checks if $c \in Anc_x(v)$ and if so update its score values in Steps 4-5.

Algorithm 5 Algorithm for updating scores at v when v is a descendant of new blocker node c

Input: blocker vertex c added to Q.
There is no communication in this algorithm, it is entirely a **local computation** at v.

```
1: if score(v) ≠ 0 then
2:     for each x ∈ V do
3:         if c ∈ Anc_x(v) then
4:             score(v) ← score(v) − score_x(v)
5:             score_x(v) ← 0
```

Lemma 5.3.4. *Given a blocker vertex c, Algorithm 5 correctly updates the scores of all nodes v such that v is a descendant of c in some tree T_x.*

Proof. Fix a vertex v and a tree T_x such that v is a descendant of c in T_x. By Lemma 5.3.3 $c \in Anc_x(v)$, and thus v will correctly update its score values in Steps 4-5. □

We now move to the last remaining part of the blocker set algorithm: our method to correctly update scores at ancestors of the newly chosen blocker node c in each T_x. Recall that if v is an ancestor of c in T_x we need to subtract $score_x(c)$ from $score_x(v)$. Here, in contrast to Algorithms 4 and 5 for nodes that are descendants of c in a tree, we do not precompute anything. Instead we give an $O(n)$-round method in Algorithm 6 to correctly update scores for each vertex for all trees in which that vertex is an ancestor of c.

Before we describe Algorithm 6 we establish the following lemma, which is key to our $O(n)$-round method.

Lemma 5.3.5. *Fix a vertex c. For each root vertex $x \in V - \{c\}$, let $\pi_{x,c}$ be the path from x to c in the h-hop SSSP tree T_x. Let $T = \cup_{x \in V-\{c\}}\{e \mid e \text{ lies on } \pi_{x,c}\}$, i.e., T is the set of edges that lie on some $\pi_{x,c}$. Then T is an in-tree rooted at c.*

Proof. If not, there exists some $x, y \in V - \{c\}$ such that $\pi_{x,c}$ and $\pi_{y,c}$ coincide first at some vertex z and the subpaths in $\pi_{x,c}$ and $\pi_{y,c}$ from z to c are different.

Let these paths coincide again at some vertex z' (such a vertex exists since their endpoint is same) after diverging from z. Let the subpath from z to z' in $\pi_{x,c}$ be $\pi^1_{z,z'}$ and the corresponding subpath in $\pi_{y,c}$ be $\pi^2_{z,z'}$. Similarly let $\pi_{x,z}$ be the subpath of $\pi_{x,c}$ from x to z and let $\pi_{y,z}$ be the subpath of $\pi_{y,c}$ from y to z.

Clearly both $\pi^1_{z,z'}$ and $\pi^2_{z,z'}$ have equal weight (otherwise one of $\pi_{x,c}$ or $\pi_{y,c}$ cannot be a shortest path). Thus the path $\pi_{x,z} \circ \pi^2_{z,z'}$ is also a shortest path.

Let (a, z') be the last edge on the path $\pi^1_{z,z'}$ and (b, z') be the last edge on the path $\pi^2_{z,z'}$.

Now since the path $\pi_{x,z'}$ has (a, z') as the last edge and we break ties using the IDs of the vertices, hence $ID(a) < ID(b)$. But then the shortest path $\pi_{y,z'}$ must also have chosen (a, z') as the last edge and hence $\pi_{y,z} \circ \pi^1_{z,z'}$ must be the subpath of path $\pi_{y,c}$, resulting in a contradiction □

Lemma 5.3.5 allows us to re-cast the task for ancestor nodes to the following

Algorithm 6 Pipelined Algorithm for updating scores at v for all trees T_x in which v is an ancestor of newly chosen blocker node c

Input: current blocker set Q, newly chosen blocker node c

1: **Send [lines 2-3]: (only for c)**
2: **Local Step at c:** create a list $list_c$ and **for each** $x \in V$ **do** add an entry $Z = \langle x, score_x(c)\rangle$ to $list_c$ if $score_x(c) \neq 0$; then set $score_x(c)$ to 0 for each $x \in V$ and set $score(c)$ to 0
3: **Round i:** let $Z = \langle x, score_x(c)\rangle$ be the i-th entry in $list_c$; send $\langle Z\rangle$ to c's parent in T_x
4: **In round $r > 0$: (for vertices $v \in V - Q - \{c\}$)**
5: **send [lines 6-8]:**
6: **if** v received a message in round $r-1$ **then**
7: let that message be $\langle Z\rangle = \langle x, score_x(c)\rangle$.
8: **if** $v \neq x$ **then** send $\langle Z\rangle$ to v's parent in T_x
9: **receive [lines 10-11]:**
10: **if** v receives a message M of the form $\langle x, score_x(c)\rangle$ **then**
11: $score_x(v) \leftarrow score_x(v) - score_x(c)$; $score(v) \leftarrow score(v) - score_x(c)$

(where we use the notation in the statement of Lemma 5.3.5): the new blocker node c needs to send $score_x(c)$ to all nodes on $\pi_{x,c}$ for each tree T_x. Recall that in the CONGEST model for directed graphs the graph edges are bi-directional. Hence this task can be accomplished by having c send out $score_x(c)$ for each tree T_x (other than T_c) in $n-1$ rounds, one score per round (in no particular order) along the parent edge for T_x. Each message $\langle x, score_x(c)\rangle$ will move along edges in $\pi_{x,c}$ (in reverse order) along parent edges in T_x from c to x. Consider any node v. In general it will be an ancestor of c in some subset of the $n-1$ trees T_x. But the characterization in Lemma 5.3.5 establishes that the incoming edge to v in all of these trees is the same edge (u, v) and this is the unique edge on the path from c to v in the h-hop SSSP. In fact, the messages for all of the trees in which v is an ancestor of c will traverse exactly the same path from c to v. Hence, for the messages sent out by c for the different trees in $n-1$ different rounds (one for each tree other than T_c), if each vertex simply forwards any message $\langle x, score_x(c)\rangle$ it receives to its parent in

tree T_x all messages will be pipelined to all ancestors in $n - 1 + h$ rounds. This is what is done in Algorithm 6, whose steps we describe below, for completeness.

Step 2 of Algorithm 6 is local computation at the new blocker vertex c where for each T_x to which c belongs, c adds an entry $\langle x, score_x(c)\rangle$ to a local list $list_c$. In round i, c sends the i-th entry in its list, say $\langle y, score_y(c)\rangle$, to its parent in T_y. For node v other than c, in a general round $r > 0$, if v receives a message for some $x \in V$ it updates its score value for x (Steps 10-11) and then forwards this message to its parent in T_x in round $r + 1$ (Step 6-8).

Lemma 5.3.6. *Given a new blocker vertex c, Algorithm 6 correctly updates the scores of all nodes v in every tree T_x in which v is an ancestor of c in $O(n + h)$ rounds.*

Proof. Correctness of Algorithm 6 was argued above. For the number of rounds, c sends out it last message in round $n - 1$, and if $\pi_{v,c}$ has length k then v receives all messages sent to it by round $n - 1 + k$. Since we only have h-hop trees $k \leq h$ for all nodes, and the lemma follows. □

5.4 Conclusion

We have presented a new distributed algorithm for the exact computation of weighted all pairs shortest paths in both directed and undirected graphs. This algorithm runs in $O(n^{3/2} \cdot \sqrt{\log n})$ rounds and is the first $o(n^2)$-round deterministic algorithm for this problem in the CONGEST model. At the heart of our algorithm is a deterministic algorithm for computing blocker set. Our blocker set construction may have applications in other distributed algorithms that need to identify a relatively small set of vertices that intersect all paths in a set of paths with the same (relatively long) length.

In Chapter 6 we present a deterministic pipelined approach to solve the

weighted all pairs shortest path problem. This approach gives an improvement in the round complexity for graphs with moderate integer edge weights. In Chapter 7, we present a $\tilde{O}(n^{4/3})$ rounds deterministic APSP algorithm that improves on the $\tilde{O}(n^{3/2})$ round bound presented in this chapter. The main component of this algorithm is a new faster method for computing blocker set deterministically and a new approach to propagate distance values from source nodes to blocker nodes.

Chapter 6

Improved Distributed Weighted APSP Through Pipelining

6.1 Introduction

In Chapter 5 we presented a $\tilde{O}(n^{3/2})$ rounds deterministic weighted all pairs shortest path algorithm in the CONGEST model for graphs with arbitrary edge weights. In this Chapter we focus on graphs with non-negative integer edge weights and we present a deterministic all pairs shortest path algorithm that provides improvement in the round complexity for graphs with moderate integer edge weights in the CONGEST model (Table 4.1 compare our results with other related results).

In sequential computation, shortest paths can be computed much faster in graphs with non-negative edge-weights (including zero weights) using the classic Dijkstra's algorithm [27] than in graphs with negative edge weights. Additionally, negative edge-weights raise the possibility of negative weight cycles in the graph, which usually do not occur in practice, and hence are not modeled by real-world weighted graphs. Thus, in the distributed setting, it is of importance to design fast shortest path algorithms that can handle non-negative edge-weights, including edges

of weight zero.

The presence of zero weight edges creates challenges in the design of distributed algorithms as observed in [50]. One approach used for positive integer edge weights is to replace an edge of weight d with d unweighted edges and then run an unweighted APSP algorithm such as [65, 47] on this modified graph. This approach is used in approximate APSP algorithms [70, 63]. However such an approach fails when zero weight edges may be present. There are a few known algorithms that can handle zero weights, such as our $\tilde{O}(n^{3/2})$-round deterministic APSP algorithm (described in Chapter 5) for graphs with arbitrary edge weights, and the randomized weighted APSP algorithms of Huang et al. [50] (for polynomially bounded non-negative integer edge weights), and of Elkin [28] and Bernstein and Nanongkai [18] for arbitrary edge weights. However no previous sub-$n^{3/2}$-round deterministic algorithm was known for weighted APSP that can handle zero weights.

All of our results hold for both directed and undirected graphs and we will assume w.l.o.g. that G is directed. Here is a summary of our results.

1. A Pipelined APSP Algorithm for Weighted Graphs. In this work we came up with a new pipelined approach for computing h-hop APSP, or more generally, (h, k)-SSP, the h-hop shortest path problem for k given sources (this problem is called the k-source short-range problem in [50]). We sometimes add an additional constraint that the shortest paths have distance at most $\triangle$ in G.

Our pipelined Algorithm 1 in Section 6.2 is compact and easy to implement, and has no large hidden constant factors in its bound on the number of rounds. It can be viewed as a (substantial) generalization of the pipelined method for unweighted APSP given in [47], which is a refinement of [65]. Our algorithm uses key values that depend on both the weighted distance and the hop length of a path, and it can store multiple distance values for a source at a given node, with the guarantee that the shortest path distance will be identified. This algorithm (Algorithm 1) achieves

the bounds in the following theorem.

Theorem 6.1.1. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph, where all edge weights are non-negative integers (with zero-weight edges allowed). The following deterministic bounds can be obtained in the* CONGEST *model for shortest path distances at most $\triangle$.*

(i) (h, k)-SSP *in* $2\sqrt{\triangle kh} + k + h$ *rounds.*

(ii) APSP *in* $2n\sqrt{\triangle} + 2n$ *rounds.*

(iii) k-SSP *in* $2\sqrt{\triangle kn} + n + k$ *rounds.*

2. Faster Deterministic APSP for Non-negative, Moderate Integer Weights. We improve on the bounds given in (*ii*) and (*iii*) of Theorem 6.1.1 by combining our pipelined Algorithm 1 with the deterministic APSP algorithm in Chapter 5. This gives our improved APSP Algorithm 3, with the bounds stated in the following Theorems 6.1.2 and 6.1.3. To obtain these improved bounds we also present an improved deterministic distributed algorithm to find a *blocker set* [10].

In our improved blocker set method we define the notion of a *consistent collection of h-hop trees, CSSSP* (Definition 6.4.1 in Section 6.4), and a simple method to compute such a collection. This result may be of independent interest.

Theorem 6.1.2. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph, where all edge weights are non-negative integers bounded by W (with zero-weight edges allowed). The following deterministic bounds can be obtained in the* CONGEST *model.*

(i) APSP in $O(W^{1/4} \cdot n^{5/4} \log^{1/2} n)$ *rounds.*

(ii) k-SSP in $O(W^{1/4} \cdot nk^{1/4} \log^{1/2} n)$ *rounds.*

Theorem 6.1.3. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph, where all edge weights are non-negative integers (with zero edge-weights allowed), and the shortest path distances are bounded by $\triangle$. The following deterministic bounds can be obtained in the* CONGEST *model.*

(i) APSP in $O(n(\triangle \log^2 n)^{1/3})$ rounds.

(ii) k-SSP in $O((\triangle k n^2 \log^2 n)^{1/3})$ rounds.

The range of values for W and Δ for which our results in Theorem 6.1.2 and 6.1.3 improve on the $\tilde{O}(n^{3/2})$ deterministic APSP bound presented in Chapter 7 are stated in the following Corollary.

Corollary 6.1.4. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph with non-negative edge weights (and zero-weight edges allowed). The following deterministic bounds hold for the* CONGEST *model for $1 \geq \epsilon \geq 0$.*

(i) If the edge weights are bounded by $W = n^{1-\epsilon}$, then APSP can be computed in $O(n^{3/2-\epsilon/4} \log^{1/2} n)$ rounds.

(ii) For shortest path distances bounded by $\Delta = n^{3/2-\epsilon}$, APSP can be computed in $O(n^{3/2-\epsilon/3} \log^{2/3} n)$ rounds.

The corresponding bounds for the weighted k-SSP problem are: $O(n^{5/4-\epsilon/4} k^{1/4} \log^{1/2} n)$ (when $W = n^{1-\epsilon}$) and $O(n^{7/6-\epsilon/3} k^{1/3} \log^{2/3} n)$ (when $\Delta = n^{3/2-\epsilon}$). Note that the result in (i) is independent of the value of Δ (it depends only on W) and the result in (ii) is independent of the value of W (it depends only on Δ).

6.1.1 Other Results

1. Simplifications to Earlier Algorithms. Our techniques simpler methods for some of procedures in two previous distributed weighted APSP algorithms that han-

Table 6.1: Table comparing our approximate APSP results for non-negative edge-weighted graphs (including zero edge weights) with previous known results.

Problem: $(1+\epsilon)$-Approximation Weighted APSP			
Author	handle zero weights	Randomized / Deterministic	Round Complexity
Nanongkai [70]	No	Randomized	$\tilde{O}(n/\epsilon^2)$
Lenzen & Patt-Shamir [63]	No	Deterministic	$\tilde{O}(n/\epsilon^2)$
Our Result	**Yes**	**Deterministic**	$\mathbf{\tilde{O}(n/\epsilon^2)}$

dle zero weight edges. In Section 6.2.6 we present simple deterministic algorithms that match the congest and dilation bounds in [50] for two of the three procedures used there: the *short-range* and *short-range-extension* algorithms. Our simplified algorithms are both obtained using a streamlined single-source version of our pipelined APSP algorithm (Algorithm 1).

2. Approximate APSP for Non-negative Edge Weights. In Section 6.5.1 we present an algorithm that matches the earlier bound for computing approximate APSP in graphs with *positive* integer edge weights [70, 63] by obtaining the same bound for non-negative edge weights. Table 6.1 compares our results with the previous results.

Theorem 6.1.5. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph, where all edge weights are non-negative integers polynomially bounded in n, and where zero-weight edges are allowed. Then, for any $\epsilon > 0$ we can compute $(1+\epsilon)$-approximate APSP in $O((n/\epsilon^2) \cdot \log n)$ rounds deterministically in the* Congest *model.*

3. Randomized APSP for Arbitrary Edge-Weights. We present a simple randomized APSP algorithm for directed graphs with arbitrary edge-weights that

runs in $\tilde{O}(n^{4/3})$ rounds, w.h.p. in n. No nontrivial sub-$n^{3/2}$ round algorithm was known prior to this result.

Theorem 6.1.6. *Let $G = (V, E)$ be a directed or undirected edge-weighted graph with arbitrary edge weights. Then, we can compute weighted APSP in G in the* CONGEST *model in $\tilde{O}(n^{4/3})$ rounds, w.h.p. in n.*

The corresponding bound for k-SSP is $\tilde{O}(n + n^{2/3}k^{2/3})$. This result improves on the prior $\tilde{O}(n^{3/2})$-round (deterministic) bound presented in Chapter 5 but it has been subsumed by a very recent result in [18] that gives an $\tilde{O}(n)$ rounds randomized algorithm for weighted APSP.

6.2 The Pipelined APSP Algorithm

We present a pipelined distributed algorithm to compute weighted APSP for shortest path distances at most Δ. The starting point for our algorithm is the distributed algorithm for *unweighted* APSP in [47], which is a streamlined variant of an earlier APSP algorithm [65]. This unweighted APSP algorithm is very simple: each source initiates its distributed BFS in round 1. Each node v retains the best (i.e., shortest) distance estimate it has received for each source, and stores these estimates in sorted order (breaking ties by source id). Let $d(s)$ (or $d_v(s)$) denote the shortest distance estimate for source s at v and let $pos(s)$ be its position in sorted order ($pos(s) \geq 1$). In a general round r, node v sends out a shortest distance estimate $d(s)$ if $r = d(s) + pos(s)$. Since $d(s)$ is non-decreasing and $pos(s)$ is increasing, there will be at most one $d(s)$ at v that can satisfy this condition. It is shown in [47] that shortest distances for all sources arrive at v in at most $2n$ rounds under this schedule and only one message is sent out by v for each source. The key to the $2n$-round bound is that if the current best distance estimate $d(s)$ for a source s reaches v in round r then $r < d(s) + pos(s)$. Since $d(s) < n$ for any source s and $pos(s)$ is at most n,

shortest path values for all sources arrive at any given node v in less than $2n$ rounds.

For our weighted case, since $d(s)$ is at most $\triangle$ for all s, v, it appears plausible that the above pipelining method would apply here as well. Unfortunately, this does not hold since we allow zero weight edges in the graph. The key to the guarantee that a $d(s)$ value arrives at v before round $d(s) + pos(s)$ in the unweighted case in [47] is that the predecessor y that sent its $d_y(s)$ value to v must have had $d_y(s) = d_v(s) - 1$. (Recall that in the unweighted case, $d_y(s)$ is simply the hop-length of the path taken from s to y.) If we have zero-weight edges this guarantee no longer holds for the weighted path length, and it appears that the key property of the unweighted pipelining methodology no longer applies. Since edge weights larger than 1 are also possible (as long as no shortest path distance exceeds $\triangle$), the hop length of a path can be either greater than or less than its weighted distance.

6.2.1 Our (h, k)-SSP algorithm

Algorithm 1 is our pipelined algorithm for a directed graph $G = (V, E)$ with non-negative edge-weights. The input is G, together with the subset S of k vertices for which we need to compute h-hop SSPs. An innovative feature of this algorithm is that the key κ it uses for a path is not its weighted distance, but a function of *both* its hop length l and its weighted distance d. More specifically, $\kappa = d \cdot \gamma + l$, where $\gamma = \sqrt{kh/\Delta}$. This allows the key to inherit some of the properties from the algorithm in [47] through the fact that the hop length is part of κ's value, while also retaining the weighted distance which is the actual value that needs to be computed.

The new key κ by itself is not sufficient to adapt the algorithm for unweighted APSP in [47] to the weighted case. In fact, the use of κ can complicate the computation since one can have two paths from s to v, with weighted distances $d_1 < d_2$, and yet for the associated keys one could have $\kappa_1 > \kappa_2$ (because the path with the smaller weight can have a larger hop-length). Our algorithm handles this with an-

other unusual feature: it may maintain several (though not all) of the key values it receives, and may also send out several key values, even some that it knows cannot correspond to a shortest distance. These features are incorporated into a carefully tailored algorithm that terminates in $O(\sqrt{\triangle kh})$ rounds with all h-hop shortest path distances from the k sources computed.

It is not difficult to show that eventually every shortest path distance key arrives at v for each source from which v is reachable when Algorithm 1 is executed. In order to establish the bound on the number of rounds, we show that our pipelined algorithm maintains two important invariants:

Invariant 1: If an entry Z is added to $list_v$ in round r, then $r < \lceil Z.\kappa + pos(Z) \rceil$, where $Z.\kappa$ is $Z's$ key value.

Invariant 2: The number of entries for a given source s at $list_v$ is at most $\sqrt{\triangle h/k} + 1$.

Invariant 1 is the natural generalization of the unweighted algorithms [65, 47] for the key κ that we use. On the other hand, to the best of our knowledge, Invariant 2 has not been used before, nor has the notion of storing multiple paths or entries for the same source at a given node. By Invariant 2, the number of entries in any list is at most $\sqrt{\triangle kh} + k$, so $pos(Z) \leq \sqrt{\triangle kh} + k$ for every list at every round. Since the value of any κ is at most $\triangle \cdot \gamma + h$, by Invariant 1 every entry is received by round $2\sqrt{\triangle kh} + k + h$.

We now give the details of Algorithm 1 starting with a step-by-step description followed by its analysis. Recall that the key value we use for a path π is $\kappa = d \cdot \gamma + l$, where $\gamma = \sqrt{kh/\Delta}$, d is the weighted path length, and l is the hop-length of π. At each node v our algorithm maintains a list, $list_v$, of the entries and associated data it has retained. Each element Z on $list_v$ is of the form $Z = (\kappa, d, l, x)$, where x is the source vertex for the path corresponding to κ, d, and l. The elements

Table 6.2: Notations

Global Parameters:	
S	set of sources
k	number of sources, or $\|S\|$
h	maximum number of hops in a shortest path
$\triangle$	maximum weighted distance of a shortest path
n	number of nodes
γ	parameter equal to $\sqrt{hk/\Delta}$
Local Variables at node v:	
d^*_x	current shortest path distance from x to v; same as $d^*_{x,v}$
$list_v$	list at v for storing the SP and non-SP entries
Variables/Parameters for entry $Z = (\kappa, d, l, x)$ in $list_v$:	
κ	key for Z; $\kappa = d \cdot \gamma + h$
d	weight (distance) of the path associated with this entry
l	hop-length of the path associated with this entry
x	start node (i.e. source) of the path associated with this entry
p	parent node of v on the path associated with this entry
ν	number of entries for x at or below Z in $list_v$ (not stored explicitly)
*flag-d**	flag to indicate if Z is the current SP entry for source x
pos	position of Z in $list_v$ in a round r; same as pos^r, pos^r_v
SP	shortest path

on $list_v$ are ordered by key value κ, with ties first resolved by the value of d, and then by the label of the source vertex. We use $Z.\nu$ to denote the number of keys for source x stored on $list_v$ at or below Z. The position of an element Z in $list_v$ is given by $pos(Z)$, which gives the number of elements at or below Z on $list_v$. If the vertex v and the round r are relevant to the discussion we will use the notation $pos^r_v(Z)$, but we will remove either the subscript or the superscript (or both) if they are clear from the context. We also have a flag $Z.$*flag-d**which is set if Z has the smallest (d, κ) value among all entries for source x (so d is the shortest weighted distance from s to v among all keys for x on $list_v$). A summary of our notation is in Table 6.2.

Initially, when round $r = 0$, $list_v$ is empty unless v is in the source set S.

Each source vertex $x \in S$ places an element $(0, 0, 0, x)$ on its $list_x$ to indicate a path of weight 0 and hop length 0 from x to x, and $Z.$*flag-d**is set to $true$. In Step 1 of the Initialization round 0, node v initializes the distance from every source to ∞. In Step 2 every source vertex initializes the distance from itself to 0 and adds the corresponding entry in its list. There are no Sends in round 0.

INITIALIZATION: Initialization procedure for Algorithm 1 at node v

Input: set of sources S

1: **for each** $x \in S$ **do** $d_x^* \leftarrow \infty$
2: **if** $v \in S$ **then** $d_v^* \leftarrow 0$; add an entry $Z = (0, 0, 0, v)$ to $list_v$; $Z.$*flag-d**$\leftarrow true$

Algorithm 1 Pipelined (h, k)-SSP algorithm at node v for round r

Input: A set of sources S

1: **send [Steps 1-2]: if** there is an entry Z with $\lceil Z.\kappa + pos_v^r(Z) \rceil = r$
2: **then** compute $Z.\nu$ and form the message $M = \langle Z,\ Z.$*flag-d**$, Z.\nu \rangle$ and send M to all neighbors
3: **receive [Steps 3-13]:** let I be the set of incoming messages
4: **for each** $M \in I$ **do**
5: let $M = (Z^- = (\kappa^-, d^-, l^-, x), Z^-.$*flag-d**$, Z^-.\nu)$ and let the sender be y.
6: $\kappa \leftarrow \kappa^- + w(y, v) \cdot \gamma + 1$; $d \leftarrow d^- + w(y, v)$; $l \leftarrow l^- + 1$
7: $Z \leftarrow (\kappa, d, l, x)$; $Z.$*flag-d**$\leftarrow false$; $Z.p \leftarrow y$ (Z may be added to $list_v$ in Step 11 or 13)
8: let Z^* be the entry for x in $list_v$ such that $Z^*.$*flag-d**$= true$, if such an entry exists (otherwise $d_x^* = \infty$)
9: **if** $Z^-.$*flag-d**$= true$ and $l \leq h$ and $((d < d_x^*)$ or $(d = d_x^*$ and $Z.\kappa < Z^*.\kappa)$ or $(d = d_x^*$ and $Z.\kappa = Z^*.\kappa$ and $Z.p < Z^*.p))$ **then**
10: $d_x^* \leftarrow d$; $Z.$*flag-d**$\leftarrow true$; $Z^*.$*flag-d**$\leftarrow false$ (if Z^* exists)
11: INSERT(Z)
12: **else**
13: **if** there are less than $Z^-.\nu$ entries for x with $key \leq Z.\kappa$ **then** INSERT(Z)

In a general round r, in Step 1 of Algorithm 1, v checks if $list_v$ contains an entry Z with $\lceil Z.\kappa + pos_v(Z) \rceil = r$. If there is such an entry Z then v sends Z to its neighbors, along with $Z.\nu$ and $Z.flag\text{-}d^*$ in Step 2. Steps 3-13 describe the steps taken at v after receiving a set of incoming messages I from its neighbors. In Step 7

INSERT(Z): Procedure for adding Z to $list_v$

1: insert Z in $list_v$ in sorted order of (κ, d, x)
2: **if** $\exists$ an entry $Z^{'}$ for x in $list_v$ such that $Z^{'}$.*flag-d**$= false$ and $pos(Z^{'}) > pos(Z)$ **then**
3: find $Z^{'}$ with smallest $pos(Z^{'})$ such that $pos(Z^{'}) > pos(Z)$ and $Z^{'}$.*flag-d**$= false$
4: remove $Z^{'}$ from $list_v$

an entry Z is created from an incoming message M, updated to reflect the d and l values at v. Step 9 checks if Z has a shorter distance than the current shortest path entry, Z^*, at v, or a shorter hop-length (if the distance is the same), or a parent with smaller ID (if both distance and hop-length are same). And if so, then Z is marked as SP in Step 10 and is then inserted in $list_v$ in Step 11. Otherwise, if Z is a non-SP it is inserted into $list_v$ in Step 13 only if the number of entries on $list_v$ for source x with key $< Z.\kappa$ in $list_v$ is less than $Z^{-}.\nu$. This is the rule that decides if a received entry that is not the SP entry is inserted into $list_v$.

Steps 1-4 of procedure INSERT perform the addition of a new entry Z to $list_v$. In Step 1 Z is inserted in $list_v$ in the sorted order of (κ, d, x). The algorithm then moves on to remove an existing entry for source x on $list_v$ if the condition in Step 2 holds. This condition checks if there is a non-SP entry above Z in $list_v$. If so then the closest non-SP entry above Z is removed in Steps 3-4.

Algorithm 1 performs these steps in successive rounds. We next analyze it for correctness and we also show that it terminates with all shortest distances computed before round $r = \lceil 2\sqrt{\triangle kh} + k + h \rceil$.

6.2.2 Correctness of Algorithm 1

We now provide proofs for establishing correctness of Alg. 1. The initial Observations and Lemmas given below establish useful properties of an entry Z in a $list_v$ and of $pos_v^r(Z)$ and its relation to $pos_y^r(Z^-)$. We then present the key lemmas. In

Lemma 6.2.11, we show that the collection of entries for a given source x in $list_v$ can be mapped into (d, l) pairs with non-negative l values such that $d = d^*$ for the shortest path entry, and the d values for all other entries are distinct and larger than d^*. (It turns out that we cannot simply use the d values already present in Z's entries for this mapping since we could have two different entries for source x on $list_v$, Z_1 and Z_2, that have the same d value.) Once we have Lemma 6.2.11 we are able to bound the number of entries for a given source at $list_v$ by $\frac{h}{\gamma} + 1$ in Lemma 6.2.13, and this establishes Invariant 2 (which is stated in Section 6.2.1). Lemma 6.2.14 establishes Invariant 1. In Lemma 6.2.15 we establish that all shortest path values reach node v. With these results in hand, the final Lemma 6.2.16 for the round bound for computing (h, k)-SSP with shortest path distances at most $\triangle$ is readily established, which then gives Theorem 6.1.1.

Observations and Lemmas 6.2.1-6.2.8: In the following Observations and Lemmas we point out the key facts about an entry Z in $list_v$ in our Algorithm 1. We use these in our proofs in this section.

Observation 6.2.1. *Let Z be an entry for a source $x \in S$ added to $list_v$ in round r. Then if Z is removed from $list_v$ in a round $r' \geq r$, it was replaced by another entry for x, Z', such that $pos_v^{r'}(Z) > pos_v^{r'}(Z')$ and $Z.\kappa \geq Z'.\kappa$.*

Proof. An entry is removed from $list_v$ only in Step 4 of INSERT, and this occurs at most once in round r' (through a call from either Step 11 or Step 13 of Algorithm 1). But immediately before that removal an entry Z' with a smaller value was inserted in $list_v$ in Step 1 of INSERT. □

Lemma 6.2.2. *Let Z be an entry in $list_v$. Then $pos_v^{r'}(Z) \geq pos_v^{r}(Z)$ for all rounds $r' > r$, for which Z exists in v's list.*

Proof. If not, then it implies that there exists Z' such that Z' was below Z in v's list in round r and was replaced by another entry Z'' that was above Z in a round r''

such that $r' \geq r'' > r$ and hence $pos_v^{r''}(Z'') > pos_v^{r''}(Z')$. But by Observation 6.2.1 this cannot happen and thus resulting in a contradiction. □

Observation 6.2.3. *Let Z be an entry for source x that was added to $list_v$. If there exists a non-SP entry for x above Z in $list_v$, then the closest non-SP entry above Z will be removed from $list_v$.*

Proof. This is immediate from Steps 1-4 of the procedure INSERT. □

Observation 6.2.4. *Let Z^- be an entry for a source x sent from y to v in round r, and let Z be the corresponding entry created for possible addition to $list_v$ in Step 7 of Algorithm 1. If Z is not added to $list_v$, then there is an entry $Z' \neq Z$ in $list_v$ with Z'.flag-d*= true, and there are at least $Z^-.\nu$ entries for x with key $\leq Z.\kappa$ at the end of round r.*

Proof. This is immediate from Steps 8 and 9 of Algorithm 1 where the current entry Z^* with Z^*.*flag-d*= true* is verified to have a shorter distance (or a smaller key if Z and Z^* have the same distance), and by the check in Step 13. □

Observation 6.2.5. *Let Z^* be a current SP entry for a source $x \in S$ present in $list_v$. Then $Z^*.l \leq h$.*

Proof. This is immediate from the check in Step 9. □

The above Observation should be contrasted with the fact that $list_v$ could contain entries Z with $Z.l > h$, but only if *flag-d*$^*(Z) = false$. In fact it is possible that $list_v$ contains an entry $Z' \neq Z^*$ with $Z'.d = d^*$ and $l > h$ since such an entry would fail the check in Step 9 but could then be inserted in Step 13 of Algorithm 1.

Lemma 6.2.6. *Let Z be an entry for source x that is present on $list_v$ in round r. Let $r' > r$, and let c and c' be the number of entries for source x on $list_v$ that have key value less than Z's key value in rounds r and r' respectively. Then $c' \geq c$.*

Proof. If $c' < c$ then an entry for x that was present below Z in round r must have been removed without having another entry for x being inserted below Z. But by Observation 6.2.1 this is not possible since any time an entry for source x is removed from $list_v$ another entry for source x with smaller key value is inserted in $list_v$. □

Lemma 6.2.6 holds for every round greater than r, even if Z is removed from $list_v$. The following stronger lemma holds for rounds greater than r when Z remains on $list_v$.

Lemma 6.2.7. *Let Z be a non-SP entry for source x that is present on $list_v$ in round r. Let $r' > r$, and let c and c' be the number of entries for source x on $list_v$ that have key value less than Z's key value in rounds r and r' respectively. Then $c' = c$.*

Proof. If a new entry Z' with key $< Z.\kappa$ for x is added, then by Observation 6.2.3 the closest non-SP entry for x with key $> Z'.\kappa$ must be removed from $list_v$ and thus $c' \leq c$. Then using Lemma 6.2.6 we have $c' = c$. □

Lemma 6.2.8. *Let Z^- be an entry for source x sent from y to v and suppose the corresponding entry Z (Step 7 of Algorithm 2) is added to $list_v$ in round r. Then there are at least $Z^-.\nu$ entries at or below Z in $list_v$ for source x.*

Proof. Let us assume inductively that this result holds for all entries on $list_v$ and $list_y$ with key value at most $Z.\kappa$ at all previous rounds and at y in round r as well. (It trivially holds initially.)

Let Z_1^- be the $(Z^-.\nu - 1)$-th entry for source x in $list_y$. Since Z_1^- has a key value smaller than Z^- it was sent to v in an earlier round r'. If the corresponding entry Z_1 created for possible addition to $list_v$ in Step 7 of Algorithm 1, was inserted in $list_v$ then by inductive assumption there were at least $Z_1^-.\nu = Z^-.\nu - 1$ entries for x at or below Z_1 in $list_v$. And by Lemma 6.2.6 this holds for round r as well and hence the result follows since Z is present above Z_1 in $list_v$.

And if Z_1 was not added to $list_v$ in round r', then by Observation 6.2.4 there were already $Z^-.\nu - 1$ entries for x with key $\leq Z_1.\kappa$ and by Lemma 6.2.6 there are at least $Z^-.\nu - 1$ entries for x with key $\leq Z_1.\kappa \leq Z.\kappa$ on $list_v$ at round r and hence the result follows. □

Establishing $\mathbf{pos^r_y(Z^-) \leq pos^r_v(Z)}$: For an entry Z^- sent from y to v such that Z is the corresponding entry created for possible addition to $list_v$ in Step 7 of Algorithm 1, in Lemma 6.2.9 and Corollary 6.2.10 we establish that if Z is added to $list_v$ then $pos^r_y(Z^-) \leq pos^r_v(Z)$, which is an important property of pos.

Lemma 6.2.9. *Let Z^- be an entry sent from y to v in round r and let Z be the corresponding entry created for possible addition to $list_v$ in Step 7 of Algorithm 1. For each source $x_i \in S$, let there be exactly c_i entries for x_i at or below Z^- in $list_y$. If Z is added to $list_v$, then for each $x_i \in S$, there are at least c_i entries for x_i at or below Z in $list_v$.*

Proof. If not there exists an $x_i \in S$ with strictly less than c_i entries for x_i at or below Z in $list_v$.

Let Z_1^- be the c_i-th entry for x_i in $list_y$ (if x_i is Z's source, then Z_1^- is Z^-). If Z_1^- is not Z^-, it is below Z^- in $list_y$ and so was sent in a round $r' < r$; if $Z_1^- = Z^-$ then $r' = r$. Let Z_1 be the corresponding entry created for possible addition to $list_v$ in Step 7 of Algorithm 1.

If Z_1 was added to $list_v$ and is also present in $list_v$ in round r, then by Lemma 6.2.8 and 6.2.6, there will be at least c_i entries for x_i at or below Z_1, resulting in a contradiction. And if Z_1 was removed from $list_v$ in a round $r'' < r$, then by Lemma 6.2.6, the number of entries for x_i with key $\leq Z_1.\kappa$ should be at least c_j.

Now if Z_1 was not added to $list_v$ in round r', then by Observation 6.2.4, we must already have at least c_i entries for x_i with key $\leq Z_1.\kappa$ in round r' and by Lemma 6.2.6, this must hold for all rounds $r'' > r$ as well. □

Corollary 6.2.10. *Let Z^- be an entry sent from y to v in round r and let Z be the corresponding entry created for possible addition to $list_v$ in Step 7 of Algorithm 1. If Z is added to $list_v$, then $pos_y^r(Z^-) \leq pos_v^r(Z)$.*

6.2.3 Establishing an Upper bound on $Z.\nu$

In this section (Lemmas 6.2.11-6.2.13) we establish an upper bound on the value of $Z.\nu$. This upper bound on $Z.\nu$ immediately gives a bound on the maximum number of entries that can be present in $list_v$ for a source $x \in S$.

Lemma 6.2.11. *Let $\mathcal{C}$ be the entries for a source $x \in S$ in $list_v$ in round r. Then the entries in $\mathcal{C}$ can be mapped to (d, l) pairs such that each $l \geq 0$ and each $Z \in \mathcal{C}$ is mapped to a distinct d value with $Z.\kappa = d \cdot \gamma + l$. Also $d = d_x^*$ if Z is a current shortest path entry, otherwise $d > d_x^*$.*

Proof. We will establish this result by induction on j, the number of entries in $\mathcal{C}$. For the base case, when $j = 1$, we can map d and l to the pair in the single entry Z since $Z.\kappa = d \cdot \gamma + l$. Assume inductively that the result holds at $list_u$ for all nodes u when the number of entries for x is at most $j - 1$. Consider the first time $|\mathcal{C}|$ becomes j at $list_v$, and let this occur when node y sends Z^- to v and this is updated and inserted as Z in $list_v$ in round r.

If Z is inserted as a new shortest path entry with distance value d^*, then the distinct d values currently assigned to the $j - 1$ entries for source x in $list_v$ must all be larger than d^* hence we can simply assign the d and l values in Z as its (d, l) mapping.

If Z is inserted as a non-SP entry then it is possible that the d value in Z has already been assigned to one of the $j - 1$ entries for source x on $list_v$. If this is the case, consider the entries for source x with key value at most $Z^-.\kappa$ in $list_y$ (at node y). By the check in Step 13 of Algorithm 1 we know that there are j such values. Inductively these j entries have j distinct d^- values assigned to them, and

we transform these into j distinct values for $list_v$ by adding $w'_x(y,v)\cdot\gamma+1$ to each of them. For at least one of these d^- values in y, call it d_1^-, it must be the case that $d' = d_1^- + w'_x(y,v)\cdot\gamma+1$ is not assigned to any of the $j-1$ entries for source x below Z in $list_v$. Let Z_1^- be the entry in y's list that is associated with distance d_1^-. We show that the associated l value for d' in Z on $list_v$ must be greater than 0.

$$\begin{aligned}(d' + w'_x(y,v))\cdot\gamma + l &= Z.\kappa \\ &= Z^-.\kappa + w'_x(y,v)\cdot\gamma + 1 \\ &\geq Z_1^-.\kappa + w'_x(y,v)\cdot\gamma + 1 \quad (\text{since } (pos_y(Z^-) > pos_y(Z_1^-))) \\ &= d'\cdot\gamma + l_1^- + w'_x(y,v)\cdot\gamma + 1 \\ &= (d' + w'_x(y,v))\cdot\gamma + l_1^- + 1\end{aligned}$$

Hence $l \geq l_1^- + 1 > 0$.

Since Z is a non-SP entry we also need to argue that $d' + w'_x(y,v) \neq d_x^*$. If not then by induction, it implies that the entry Z_1^- for x in y's list correspond to the current shortest path entry for x in $list_y$. Since Z_1^- gives the shortest path distance from x to y, the corresponding shortest path entry for x must be below Z in v's list and by induction, it must have d_x^* associated with it. This results in a contradiction since we chose the distance value, $d' + w'_x(y,v)$, such that it was different from the distances associated with the other $(j-1)$ entries for x in v's list.

We have shown that the lemma holds the first time a j-th entry is added to $list_v$ for source x. To complete the proof we now show that the lemma continues to hold if a new entry Z for source x is added to $list_v$ while keeping the number of entries at j. The argument is the same as the case of having j entries for source x for the first time except that we also need to consider duplication of a d value at an entry above the newly inserted Z. For this we proceed as in the previous case.

Let Z be inserted in position $p \leq j$. We assign a d value to Z as in the previous case, taking care that the d value assigned to Z is different from that for the $p-1$ entries below Z. Suppose Z's d value has been assigned to another entry Z'' in $list_v$ above Z. Then, we consider Z', the entry that was removed (in Step 5 of INSERT) in order to keep the total number of entries for source x at j. We assign to Z'' the value d' that was assigned to Z'. Since Z'' has a larger key value than Z' we will need to use an l'' at least as large as that used for Z' (call it l') in order satisfy the requirement that $Z''.\kappa = d' \cdot \kappa + l''$. Since l' must have been non-negative, l'' will also be non-negative as required, and all d values assigned to the entries for x will be distinct. □

Lemma 6.2.12. *Let Z be the current shortest path distance entry for a source $x \in S$ in v's list. Then the number of entries for x below Z in $list_v$ is at most $\gamma \cdot \frac{n}{k}$.*

Proof. By Lemma 6.2.11, we know that the keys of all the entries for x can be mapped to (d, l) pairs such that each entry is mapped to a distinct d value and $l > 0$.

We have $Z.\kappa = d_x^* \cdot \gamma + l_x^*$, where l_x^* is the hop-length of the shortest path from x to v. Let Z'' be an entry for x below Z in v's list. Then, $Z''.\kappa \leq Z.\kappa$. It implies

$$
\begin{aligned}
d'' \cdot \gamma + l'' &\leq d_x^* \cdot \gamma + l_x^* \\
d'' \cdot \gamma &\leq d_x^* \cdot \gamma + (l_x^* - l'') \\
d'' &\leq d_x^* + \frac{(l_x^* - l'')}{\gamma} \\
&\leq d_x^* + \frac{(h-1)}{\gamma} \\
&< d_x^* + \frac{h}{\gamma}
\end{aligned}
$$

$$= d_x^* + \frac{h}{\gamma^2} \cdot \gamma$$
$$= d_x^* + \frac{n}{k} \cdot \gamma$$

Thus $d'' < d_x^* + \frac{n}{k} \cdot \gamma$. Since $d'' \geq d_x^*$, there can be at most $\frac{n}{k} \cdot \gamma$ entries for x below Z in $list_v$. □

Lemma 6.2.13. *For each source $x \in S$, v's list has at most $\frac{n}{k} \cdot \gamma + 1$ entries for x.*

Proof. On the contrary, let Z be an entry for source $x \in S$ with the smallest key such that Z is the $(\gamma \cdot \frac{n}{k} + 2)$-th entry for x in $list_v$. Let y be the sender of Z to v and let the corresponding entry in y's list be Z^-.

If Z was added as a non-SP entry, then by Lemma 6.2.8 there are at least $\gamma \cdot \frac{n}{k} + 2$ entries for x at or below Z^- in $list_y$, resulting in a contradiction as Z is the entry with the smallest key that have this ν value.

Otherwise if Z was added as a current shortest path entry, then by Lemma 6.2.12, Z can have at most $\gamma \cdot \frac{n}{k}$ entries below it in any round and hence there are at most $\gamma \cdot \frac{n}{k} + 1$ at or below Z in $list_v$ in all rounds (and if Z is later marked as non-SP then by Lemma 6.2.7 $Z.\nu$ will stay fixed at that value), again resulting in a contradiction. □

6.2.4 Establishing an Upper Bound on the round r by which an entry Z is sent

Lemma 6.2.14. *If an entry Z is added to $list_v$ in round r then $r < Z.\kappa + pos_v^r(Z)$.*

Proof. The lemma holds in the first round since all entries have non-negative κ, any received entry has hop length at least 1, and the lowest position is 1 so for any entry Z received by v in round 1, $Z.\kappa + pos_v^1(Z) \geq 1 + 1 > 1$.

Let r be the first round (if any) in which the lemma is violated, and let it occur when entry Z is added to $list_v$. So $r \geq Z.\kappa + pos_v^r(Z)$. Let $r_1 = Z.\kappa + pos_v^r(Z)$

(so $r_1 < r$ by assumption).

Since Z was added to $list_v$ in round r, Z^- was sent to v by a node y in round r. So $r = Z^-.\kappa + pos_y^r(Z^-)$. But $Z.\kappa > Z^-.\kappa$ and $pos_v^r(Z) \geq pos_y^r(Z^-)$, hence r must be less than $Z.\kappa + pos_v^r(Z)$. □

Lemma 6.2.15. *Let $\pi_{x,v}^*$ be a shortest path from source x to v with the minimum number of hops among h-hop shortest paths from x to v. Let $\pi_{x,v}^*$ have l^* hops and shortest path distance $d_{x,v}^*$. Then v receives an entry $Z^* = (\kappa, d_{x,v}^*, l^*, x)$ by round $r < Z^*.\kappa + pos_v^r(Z^*)$.*

Proof. If an entry $Z^* = (\kappa, d_{x,v}^*, l^*, x)$ is placed on $list_v$ by v then by Lemma 6.2.14 it is received before round $Z^*.\kappa + pos_v^r(Z^*)$ and hence it will be sent in round $r = Z^*.\kappa + pos_v^r(Z^*)$ in Step 1. It remains to show that an entry for path $\pi_{x,v}^*$ is received by v. We establish this for all pairs x, v by induction on key value κ.

If $\kappa = 0$, then it implies that the shortest path is the vertex x itself and thus the statement holds for $\kappa = 0$. Let us assume that the statement holds for all keys $< \kappa$ and consider the path $\pi_{x,v}^*$ with key $\kappa = d_{x,v}^* \cdot \gamma + l^*$.

Let (y, v) be the last edge on the path $\pi_{x,v}^*$ and let $\pi_{x,y}^*$ be the subpath of $\pi_{x,v}^*$ from x to y. By construction the path $\pi_{x,y}^*$ is a shortest path from x to y and its hop length $l^* - 1$ is the smallest among all shortest paths from x to y. Hence by the inductive assumption an entry Z^- with $Z^-.\kappa = d_{x,y}^* \cdot \gamma + l^* - 1$ (which is strictly less than $Z^*.\kappa$) is received by y before round $Z^-.\kappa + pos_y'(Z^-)$ (by Lemma 6.2.14) and is then sent to v in round $r' = Z^-.\kappa + pos_y^{r'}(Z^-)$ in Step 1. Thus v adds the shortest path entry for x, Z^*, to $list_v$ by the end of round r'. □

6.2.5 Establishing an Upper Bound on the round r by which Algorithm 1 terminates

Lemma 6.2.16. *Algorithm 1 correctly computes the h-hop shortest path distances from each source $x \in S$ to each node $v \in V$ by round $(n-1)\gamma + h + n \cdot \gamma + k$.*

Proof. An h-hop shortest path has hop-length at most h and weight at most $n-1$, hence a key corresponding to a shortest path entry will have value at most $(n-1)\gamma+h$. Thus by Lemma 6.2.15, for every source $x \in S$ every node $v \in V$ should have received the shortest path distance entry, Z^*, for source x by round $r = (n-1)\gamma+h+pos_v^r(Z^*)$.

Now we need to bound the value of $pos_v^r(Z^*)$. By Lemma 6.2.13, we know that there are at most $\gamma \cdot \frac{n}{k} + 1$ entries for each source $x \in S$ in a node v's list. Now as there are k sources, v's list has at most $(\gamma \cdot \frac{n}{k} + 1) \cdot k \leq \gamma \cdot n + k$ entries, thus $pos_v^r(Z^*) \leq \gamma \cdot n + k$ and hence $r \leq (n-1)\gamma + h + \gamma \cdot n + k$. □

Since $\gamma = \sqrt{\frac{hk}{n}}$, Lemma 6.2.16 establishes the bounds given in Theorem 6.1.1.

6.2.6 Simplified Versions of Short-Range Algorithms in [50]

We describe here simplified versions of the *short-range* and *short-range-extension* algorithms used in the randomized $\tilde{O}(n^{5/4})$ round APSP algorithm in Huang et al. [50]. Our short-range Algorithm 2 is implicit in our pipelined APSP algorithm (Algorithm 1) and is much simpler than it since it is for a single source.

Given a hop-length h and a source vertex x, the short-range algorithm in [50] computes the h-hop shortest path distances from source x in a graph G' (obtained through 'scaling') where $\Delta \leq n-1$. The scaled graph has different edge weights for different sources, and hence h-hop APSP is computed through n h-hop SSSP (or *short-range*) computations, each of which runs with *dilation* (i.e., number of rounds) $\tilde{O}(n\sqrt{h})$ and *congestion* (i.e., maximum number of messages along an edge) $O(\sqrt{h})$. By running this algorithm using each vertex as source, h-hop APSP is computed in G' in $O(n\sqrt{h})$ rounds w.h.p. in n using a scheduling result in Ghaffari's framework [39], which gives a randomized method to execute this collection of different short-range executions simultaneously in $\tilde{O}(\text{dilation} + n \cdot \text{congestion}) = \tilde{O}(n\sqrt{h})$ rounds.

The short-range algorithm in [50] for a given source runs in two stages:.

Initially every zero edge-weight is increased to a positive value $\alpha = 1/\sqrt{h}$ and then h-hop SSSP is computed using a BFS variant in $\tilde{O}(n/\alpha) = \tilde{O}(n\sqrt{h})$ rounds. This gives an approximation to the h-hop SSSP where the additive error is at most $h\alpha = \sqrt{h}$. This error is then fixed by running the Bellman-Ford algorithm [15] for h rounds. The total round complexity of this SSSP algorithm is $\tilde{O}(n\sqrt{h})$ and the congestion is $O(\sqrt{h})$.

Algorithm 2 Round r of short-range algorithm for source x
(initially $d^* \leftarrow 0$; $l^* \leftarrow 0$ at source x)

(at each node $v \in V$)
1: **send: if** $\lceil d^* \cdot \sqrt{h} + l^* \rceil = r$ **then** send (d^*, l^*) to all the neighbors
2: **receive [Steps 2-6]:** let I be the set of incoming messages
3: **for each** $M \in I$ **do**
4: let $M = (d^-, l^-)$ and let the sender be y.
5: $d \leftarrow d^- + w(y, v)$; $l \leftarrow l^- + 1$
6: **if** $d < d^*$ or ($d = d^*$ and $l < l^*$) **then** set $d^* \leftarrow d$; $l^* \leftarrow l$

We now present a simplified short-range algorithm (Algorithm 2) with the same dilation $O(n\sqrt{h})$ and congestion $O(\sqrt{h})$. Here d^* is the current best estimate for the shortest path distance from x at node v and l^* is the hop-length of the corresponding path. Source node x initializes d^* and l^* values to zero and sends these values to its neighbors in round 0 (Step 1). At the start of a round r, each node v checks if its current d^* and l^* values satisfy $\lceil d^* \cdot \sqrt{h} + l^* \rceil = r$, and if so, it sends this estimate to each of its neighbors. To bound the number of such messages v sends throughout the entire execution, we note that v will send another message in a future round only if it receives a smaller d^* value with higher $\lceil d^* \cdot \sqrt{h} + l^* \rceil$ value. But since $l^* \leq h$ and d^* values are non-negative integers, v can send at most $\sqrt{h}$ messages to its neighbors throughout the entire execution. A proof similar to [47] (a simplified version of Lemma 6.2.14) shows that as long as edge-weights are non-negative, v will always receive the message that creates the pair d^*, l^* at v before round $\lceil d^* \cdot \sqrt{h} + l^* \rceil$.

If shortest path distances are bounded by Δ, Algorithm 2 runs in $\lceil \Delta \cdot \sqrt{h} + h \rceil$ rounds with congestion at most $\sqrt{h}$. And if $\Delta \leq n-1$ (as in [50]), then we can compute shortest path distances from x to every node v in $O(n\sqrt{h})$ rounds.

We can similarly simplify the short-range-extension algorithm in [50], where some nodes already know their distance from source x and the goal is to compute shortest paths from x by extending these already computed shortest paths to u by another h hops. To implement this, we only need to modify the initialization in Algorithm 2 so that each such node u initializes d^* with this already computed distance. The round complexity is again $O(\Delta\sqrt{h})$ and the congestion per source is $O(\sqrt{h})$. This gives us the following result.

Lemma 6.2.17. *Let $G = (V, E)$ be a directed or undirected graph, where all edge weights are non-negative distances (and zero-weight edges are allowed), and where shortest path distances are bounded by Δ. Then by using Algorithm 2, we can compute h-hop SSSP and h-hop extension in $O(\Delta\sqrt{h})$ rounds with congestion bounded by $\sqrt{h}$.*

As in [50] we can now combine our Algorithm 2 with Ghaffari's randomized framework [39] to compute h-hop APSP and h-hop extensions (for all source nodes) in $\tilde{O}(\Delta\sqrt{h} + n\sqrt{h})$ rounds w.h.p. in n. The result can be readily modified to include the number of sources, k, by sending the current estimates (d^*, l^*) in round $\lceil d^* \cdot \gamma + l^* \rceil$, where $\gamma = \sqrt{hk/\Delta}$ as in Algorithm 1 (instead of $\lceil d^* \cdot \sqrt{h} + l^* \rceil$), and the resulting algorithm runs in $O(\sqrt{\Delta hk})$ rounds with congestion bounded by $\sqrt{\Delta h/k}$. Then we can compute h-hop k-SSP and h-hop extensions for all k sources in $\tilde{O}(\sqrt{\Delta hk})$ rounds.

6.3 Faster k-SSP Algorithm using blocker set

In this section we give faster APSP and k-SSP algorithms. The overall Algorithm 3 has the same structure as the deterministic $O(n^{3/2} \cdot \sqrt{\log n})$ round weighted APSP

algorithm in Chapter 5 but we use a variant of Algorithm 1 in place of Bellman-Ford, and we also present new methods within two of the steps.

In our improved Algorithm 3, Steps 3-5 are unchanged from the algorithm in Chapter 5 (Algorithm 1). However we give an alternate method for Step 1, which computes h-hop CSSSP, since the method in Chapter 5 (Algorithm 1) takes $\Theta(n \cdot h)$ rounds, which is too large for our purposes. Our new method is very simple and uses Algorithm 1 and runs in $O(\sqrt{\triangle hk})$ rounds. The following lemma is straightforward and can be established by replacing Bellman-Ford algorithm with Algorithm 1 in Lemma 6.4.3.

Lemma 6.3.1. *h-hop CSSSPs can be computed in $O(\sqrt{\triangle hk})$ rounds using Algorithm 1.*

For Step 2 we use the overall blocker set algorithm from Chapter 5 (Algorithm 2), which runs in $O(n \cdot h + (n^2 \log n)/h)$ rounds and computes a blocker set of size $q = O((n \log n)/h)$ for the h-hop trees constructed in Step 1 of Algorithm 3. But this gives only an $\tilde{O}(n^{3/2})$ bound for Step 2 (by setting $h = \tilde{O}(\sqrt{n})$), so it will not help us to improve the bound on the number of rounds for APSP beyond Algorithm 1. Instead we modify and improve a key step where that earlier blocker set algorithm has a $\Theta(n \cdot h)$ round preprocessing step. We give the details of our method for Step 2 in Section 6.4.1.

Algorithm 3 Overall k-SSP algorithm (adapted from Algorithm 1 (Chapter 5))

Input: set of sources S, number of hops h

1: Compute h-hop CSSSP rooted at each source $x \in S$ (described in Section 6.4).
2: Compute a blocker set Q of size $\Theta(\frac{n \log n}{h})$ for the h-hop CSSSP computed in Step 1 (described in Section 6.4.1).
3: **for each** $c \in Q$ **in sequence:** compute SSSP tree rooted at c.
4: **for each** $c \in Q$ **in sequence:** broadcast $ID(c)$ and the shortest path distance values $\delta_h(x, c)$ for each $x \in S$.
5: **Local Step at node** $v \in V$**:** for each $x \in S$ compute the shortest path distance $\delta(x, v)$ using the received values.

Lemma 6.3.2. *Algorithm 3 computes k-SSP in* $O(\frac{n^2 \log n}{h} + \sqrt{\triangle hk})$ *rounds.*

Proof. The correctness of Algorithm 3 is established in Lemma 5.2.2. Step 1 runs in $O(\sqrt{\triangle hk})$ rounds by Lemma 6.3.1. In Section 6.4.1 we will give an $O(n \cdot q + \sqrt{\triangle hk})$ rounds algorithm to find a blocker set of size $q = O(\frac{n \log n}{h})$. Steps 3 and 4 take $O(n \cdot q)$ rounds (Lemma 5.2.6). Step 5 has no communication. Hence the overall bound for Algorithm 3 is $O(n \cdot q + \sqrt{\triangle hk})$ rounds. Since $q = O(\frac{n \log n}{h})$ this gives the desired bound. □

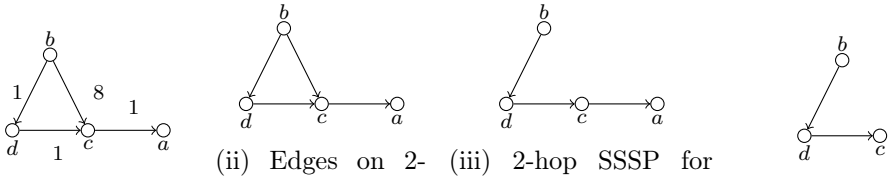
Proofs of Theorem 6.1.3 and 6.1.2: Using $h = \frac{n^{4/3} \cdot \log^{2/3} n}{(2k \cdot \triangle)^{1/3}}$ in Lemma 6.3.2 we obtain the bounds in Theorem 6.1.3.

If edge weights are bounded by W, the weight of any h-hop path is at most hW. Hence by Lemma 6.3.2, the k-SSP algorithm (Algorithm 3) runs in $O(\frac{n^2 \log n}{h} + h\sqrt{Wk})$ rounds. Setting $h = n \log^{1/2} n/(W^{1/4}k^{1/4})$ we obtain the bounds stated in Theorem 6.1.2. □

6.4 Computing Consistent h-hop trees (CSSSP)

We first present our new notion of computing *Consistent h-hop trees*, which forms an important component of our *blocker set* algorithm. We then describe our algorithm for computing blocker set.

Recall that an h-hop shortest path from a source s to a vertex v in G is a path of minimum weight from s to v among all paths with at most h hops. If we consider the graph consisting of an h-hop shortest path from a source s to every vertex in G reachable from s within h hops, it need not form a tree since the prefix of an h-hop shortest path may not itself be an h-hop shortest path. The parent pointers for the h-hop shortest paths computed by Bellman-Ford algorithm [15] (or our pipelined (h, k)-SSP algorithm in Chapter 5) suffer from a similar problem: the tree constructed by the parent pointers could have height greater than h (see



(i) Example graph G.

(ii) Edges on 2-hop shortest paths from source node b.

(iii) 2-hop SSSP for source node b constructed by Bellman-Ford.

(iv) 2-hop CSSSP for source node b.

Figure 6.1: This figure gives an example graph G where the union of the edges on the 2-hop shortest paths from source node b differs from the 2-hop SSSP constructed by Bellman-Ford (or using our (h, k)-SSP pipelined algorithm in Chapter 5), and both are different from the 2-hop CSSSP generated for source node b.

Fig 6.1).

Within the algorithm for computing blocker set in our APSP algorithm (described in Section 5.3), there are algorithms for updating the 'scores' of the ancestor and descendant nodes of a newly chosen blocker node in the collection of trees that contain h-hop shortest paths. The efficient methods used in these algorithms are based on having a consistent set of paths across all trees in the collection. In order to create a consistent collection of paths across all sources, we introduce the following definition of an *h-hop Consistent SSSP (CSSSP)* collection.

Definition 6.4.1 (CSSSP). *Let H be a collection of rooted trees of height h in a graph $G = (V, E)$. Then H is an* h-hop CSSSP collection *(or simply an* h-hop CSSSP*) if for every $u, v \in V$ the path from u to v is the same in each of the trees in H (in which such a path exists), and is the h-hop shortest path from u to v in the h-hop tree T_u rooted at u. Further, each T_u contains every vertex v that has a path with at most h hops from u in G that has distance $\delta(u, v)$.*

Running h iterations of Bellman-Ford (or our pipelined (h, k)-SSP algorithm in Chapter 5) is not guaranteed to construct a CSSP collection. At the same time, we observe that the trees in an h-hop CSSSP collection may not contain all h-hop

shortest paths: In particular, if every shortest path from source s to a vertex x has more than h hops, then the h-hop tree for source s in the CSSSP collection is not required to have x in it (see Fig. 6.1).

Our method to construct an h-hop CSSSP collection is very simple: We execute Bellman-Ford algorithm to construct $2h$-hop SSSPs instead of h-hop SSSPs (Note that if there are two different shortest paths between a pair of vertices u and v, then the one with shorter hop-length is preferred and in case of a tie, the one with smaller last edge ID is preferred). Our CSSSP collection will retain the initial h hops of each of these $2h$-hop SSSPs. In other words, each vertex v willl set the parent pointer $p(v)$ to NIL for a source s if the hop-length of the corresponding path is greater than h. In [8] we show that this simple construction results in an h-hop CSSSP collection. Thus we are able to construct h-hop CSSSPs by incurring just a constant factor overhead in the number of rounds over the bound for constructing h-hop SSSPs.

Lemma 6.4.2. *Consider running Bellman-Ford algorithm (or our pipelined (h,k)-SSP algorithm in Chapter 5) using the hop-length bound $2h$. Let $\mathcal{C}$ be the collection of h-hop trees formed by retaining the initial h hops in each of these $2h$-hop SSSPs. Then the collection $\mathcal{C}$ forms an h-hop CSSSP collection.*

Proof. If not, then there exist vertices u, v and trees T_x, T_y such that the paths from u to v in T_x and T_y are different. Let $\pi^x_{u,v}$ and $\pi^y_{u,v}$ be the corresponding paths in these trees.

There are three possible cases: (1) when $wt(\pi^x_{u,v}) \neq wt(\pi^y_{u,v})$ (2) when paths $\pi^x_{u,v}$ and $\pi^y_{u,v}$ have same weight but different hop-lengths (3) when both $\pi^x_{u,v}$ and $\pi^y_{u,v}$ have same weight and hop-length.

(1) $wt(\pi^x_{u,v}) \neq wt(\pi^y_{u,v})$: w.l.o.g. assume that $wt(\pi^x_{u,v}) < wt(\pi^y_{u,v})$. Now if we replace $\pi^y_{u,v}$ in T_y with $\pi^x_{u,v}$, we get a path of smaller weight from y to v of hop-length at most $2h$. But then node v should have picked this lighter path during the

execution of Bellman-Ford with y as the source node, resulting in a contradiction.

(2) paths $\pi^x_{u,v}$ and $\pi^y_{u,v}$ have same weight but different hop-lengths. W.l.o.g. assume that path $\pi^x_{u,v}$ has smaller hop-length than $\pi^y_{u,v}$. Then $\kappa(\pi^x_{u,v}) < \kappa(\pi^y_{u,v})$ and hence $\kappa(\pi^y_{y,u} \circ \pi^x_{u,v}) < \kappa(\pi^y_{y,u} \circ \pi^y_{u,v})$. And again v would have picked the path $\pi^y_{y,u} \circ \pi^x_{u,v}$ as the shortest path from y during the execution of Bellman-Ford with y as the source, since paths with smaller hop-length are preferred even if they have same weighted distance.

(3) both $\pi^x_{u,v}$ and $\pi^y_{u,v}$ have same weight and hop-length. W.l.o.g. assume that these two paths have the smallest hop-length for which the paths differ. Let (a, v) be the last edge on the path $\pi^x_{u,v}$ and let (b, v) be the last edge on the path $\pi^y_{u,v}$. W.l.o.g. assume that $ID(a) < ID(b)$ (a cannot equal b since the resulting smaller hops subpaths $\pi^x_{u,a}$ and $\pi^y_{u,a}$ would be different, which is not possible). Then again during the execution of Bellman-Ford with y as the source, v would have chosen the path $\pi^y_{y,u} \circ \pi^x_{u,v}$ as the shortest path from y instead of $\pi^y_{y,v}$, since paths with smaller parent ID are preferred even if they have same weight and hop-length. □

Lemma 6.4.3. *h-hop CSSSPs can be computed in $O(nh)$ rounds using the Bellman-Ford algorithm.*

We now show two properties of an h-hop CSSSP collection that we will use in our blocker set algorithm in the next section. In the following, we call a tree T rooted at a vertex c an out-tree if all the edges incident to c are outgoing edges from c and we call T an in-tree if all the edges incident to c are incoming edges.

Lemma 6.4.4. *Let $\mathcal{C}$ be an h-hop CSSSP collection. Let c be a vertex in G and let T be the union of the edges in the collection of subtrees rooted at c in the trees in $\mathcal{C}$. Then T forms an out-tree rooted at c.*

Proof. If not, there exist nodes u and v and trees T_x and T_y such that the path from c to u in T_x and path from c to v in T_y first diverge from each other after starting

from c and then coincide again at some vertex z. But since $\mathcal{C}$ is an h-hop CSSSP collection, by Lemma 6.4.2 the path from c to z in the collection $\mathcal{C}$ is unique. □

Lemma 6.4.5. *Let $\mathcal{C}$ be an h-hop CSSSP collection. Let c be a vertex in G and let T be the union of the edges on the tree-path from the root of each tree in $\mathcal{C}$ to c (for the trees that contain c). Then T forms an in-tree rooted at c.*

Proof. If not, then there exist nodes x and y such that the path from x to c in T_x and path from y to c in T_y first coincide at some vertex z and then diverge from each other. But since $\mathcal{C}$ is an h-hop CSSSP collection, by Lemma 6.4.2 the path from z to c in the collection $\mathcal{C}$ is unique. □

6.4.1 Computing a Blocker Set

Our overall blocker set algorithm runs in $O(\frac{n^2 \log n}{h} + \sqrt{\triangle hk})$ rounds. It differs from the blocker set algorithm in Chapter 5 (Algorithm 2) by developing faster algorithms for two steps that take $O(nh)$ rounds in the earlier blocker set algorithm.

The first step in Algorithm 2 (Chapter 5) that takes $O(nh)$ rounds is the step that computes the initial 'scores' at all nodes for all h-hop trees in the CSSSP collection. The score of node v in an h-hop tree is the number of v's descendants in that tree. Here we compute scores for all trees at all nodes in $O(\sqrt{\triangle hk})$ rounds with a timestamp pipelining technique introduced in [47] for propagating values from descendants to ancestors in the shortest path trees within the same bound as the APSP algorithm.

To explain the second $O(nh)$-round step in Algorithm 2 (Chapter 5), we first give a recap of the blocker set algorithm in Chapter 5 (Algorithm 2). This algorithm picks nodes to be added to the blocker set greedily. The next node that is added to the blocker set is one that lies in the maximum number of paths in the h-hop trees that have not yet been covered by the already selected blocker nodes. To identify such a node, the algorithm maintains at each node v a count (or *score*) of the

number of descendant leaves in each tree, since the sum of these counts is precisely the number of root-to-leaf paths in which v lies. Once all vertices have their overall score, the new blocker node c can be identified as one with the maximum score. It now remains for each node v to update its scores to reflect the fact that paths through c no longer exist in any of the trees. This update computation is divided into two steps in Algorithm 2 (Chapter 5). In both steps, the main challenge is for a given node to determine, in each tree T_x, whether it is an ancestor of c, a descendant of c, or unrelated to c.

1. *Updates at Ancestors.* For each v, in each tree T_x where v is an ancestor of c, v needs to reduce its score for T_x by c's score for T_x since all of those descendant leaves have been eliminated. In Chapter 5 an $O(n)$-round pipelined algorithm (using the in-tree property for CSSSP in Lemma 6.4.5) (Algorithm 6) is given for this update at all nodes in all trees, and this suffices for our purposes.

2. *Updates at Descendants.* For each v, in each tree T_x where v is a descendant of c, v needs to reduce its score for T_x to zero, since all descendant leaves are eliminated once c is removed. In Chapter 5, this computation is performed by an $O(nh)$-round precomputation in which each vertex identifies all of its ancestors in all of the h-hop trees and thereafter can readily identify the trees in which it is a descendant of a newly chosen blocker node c once c broadcasts its identity to all nodes. But this is too expensive for our purposes.

Here, we perform no precomputation but instead in Algorithm 4 we use the property in Lemma 6.4.4 for CSSSP to develop a method similar to the one for updates at ancestors. Initially c creates a list, $list_c$, where it adds the IDs of all the source nodes x such that c lies in tree T_x. In round i, c sends the i-th entry $\langle x \rangle$ in $list_c$ to all its children in T_x. Since T (in Lemma 6.4.4) is a tree, every node v receives at most one message in a given round r. If v receives the message for source x in round r, it forwards this message to all its children in T_x in the next

Algorithm 4 Pipelined Algorithm for updating scores at v in trees T_x in which v is a descendant of newly chosen blocker node c

Input: Q: blocker set, c: newly chosen blocker node, S: set of sources

(only for c)

1: **Local Step at c:** create $list_c$ to store the ID of each source $x \in S$ such that $score_x(c) \neq 0$; **for each** $x \in S$ **do** set $score_x(c) \leftarrow 0$; set $score(c) \leftarrow 0$

2: **Send: Round i:** let $\langle x \rangle$ be the i-th entry in $list_c$; send $\langle x \rangle$ to c's children in T_x.

(round $r > 0$: for vertices $v \in V - Q - \{c\}$)

3: **send[lines 3-4]: if** v received a message $\langle x \rangle$ in round $r-1$ **then**

4: **if** $v \neq x$ **then** send $\langle x \rangle$ to v's children in T_x

5: **receive[lines 5-6]: if** v receives a message $\langle x \rangle$ **then**

6: $score(v) \leftarrow score(v) - score_x(v)$; $score_x(v) \leftarrow 0$

round, $r+1$, and also sets its score for source x to 0. Similar to the algorithm for updating ancestors of c [10], it is readily seen that every descendant of c in every tree T_x receives a message for x by round $k+h-1$.

Lemma 6.4.6. *Algorithm 4 correctly updates the scores of all nodes v in every tree T_x in which v is a descendant of c in $k+h-1$ rounds.*

6.5 Additional Results

6.5.1 An $\tilde{O}(n)$-Rounds $(1+\epsilon)$ Approximation Algorithm for Weighted APSP with Non-negative Integer Edge-Weights

Here we deal with the problem of finding $(1+\epsilon)$-approximate solution to the weighted APSP problem. If edge-weights are strictly positive, the following result is known.

Theorem 6.5.1 ([70, 63]). *There is a deterministic algorithm that computes $(1+\epsilon)$-approximate APSP on graphs with positive polynomially bounded integer edge weights in $O((n/\epsilon^2) \cdot \log n)$ rounds.*

The above result does not hold when *zero weight edges* are present. Here

we match the deterministic $O((n/\epsilon^2) \cdot \log n)$-round bound for this problem with an algorithm that also handles zero edge-weights.

We first compute reachability between all pairs of vertices connected by zero-weight paths. This is readily computed in $O(n)$ rounds, e.g., using [65, 47] while only considering only the zero weight edges (and ignoring the other edges).

We then consider shortest path distances between pairs of vertices that have no zero-weight path connecting them. The weight of any such path is at least 1. To approximate these paths we increase the zero edge-weights to 1 and transform every non-zero edge weight $w(e)$ to $n^2 \cdot w(e)$. Let this modified graph be $G' = (V, E, w')$. Thus the weight of an l-hop path p in G', $w'(p)$, satisfies $w'(p) \leq w(p) \cdot n^2 + l$. Since the modified graph G' has polynomially bounded positive edge weights, we can use the result in Theorem 6.5.1 to compute $(1 + \epsilon/3)$-approximate APSP on this graph in $\tilde{O}(9n/\epsilon^2)$ rounds.

Fix a pair of vertices u, v. Let p be a shortest path from u to v in G, and let its hop-length be l. Then $w'(p) \leq n^2 \cdot w(p) + l$. Let p' be a $(1 + \epsilon/3)$-approximate shortest path from u to v, and let its hop-length be l. Then $w'(p') \leq (1+\epsilon/3) \cdot w'(p) \leq (1 + \epsilon/3) \cdot (n^2 \cdot w(p) + l)$. Dividing $w'(p')$ by n^2 gives us $w'(p')/n^2 < w(p)(1 + \epsilon/3) + (l/n^2)(1 + \epsilon/3) < w(p) + w(p)\epsilon/3 + 2/n \leq w(p)(1 + \epsilon/3) + 2\epsilon/3 \leq w(p)(1 + \epsilon)$ (as long as $\epsilon > 3/n$ and since $w(p) \geq 1$), and this establishes Theorem 6.1.5.

6.5.2 A Simple $\tilde{O}(n^{4/3})$ Rounds Randomized Algorithm for Weighted APSP with Arbitrary Edge-Weights

We adapt the randomized framework of Huang et al. [50] to obtain a simple randomized algorithm for weighted APSP with arbitrary edge weights. Our randomized algorithm runs in $\tilde{O}(n^{4/3})$ rounds w.h.p. in n. We describe our randomized algorithm below.

As described in Section 6.2.6, Huang et al.[50] use two algorithms *short-range*

and *short-range-extension* for integer-weighted APSP for which they have randomized algorithms that run in $\tilde{O}(n\sqrt{h})$ rounds w.h.p. in n. (We presented simplified versions of these two algorithms in Section 6.2.6.) Since we consider arbitrary edge weights here, we will instead use h rounds of the Bellman-Ford algorithm [15] for both steps, which will take $O(kh)$ rounds for k source nodes.

We keep the remaining steps in [50] unchanged: These steps involve having every 'center' c broadcast its estimated shortest distances, $\delta(c', c)$, from every other center c', and each source node $x \in S$ sending its correct shortest distance, $\delta(x, c)$, to each center c. (The set of *centers* is a random subset of vertices in G of size $\tilde{O}(\sqrt{n})$.) These steps are shown in [50] to take $\tilde{O}(n + \sqrt{nkq})$ rounds in total w.h.p. in n, where $q = \Theta(\frac{n \log n}{h})$. This gives an overall round complexity $\tilde{O}(kh + n + \sqrt{nkq})$ for our algorithm. Setting $h = n^{2/3}/k^{1/3}$ and $q = n^{1/3}k^{1/3} \log n$, we obtain the desired bound of $\tilde{O}(n + n^{2/3}k^{2/3})$ in Theorem 6.1.6.

6.6 Conclusion

We have presented new deterministic distributed algorithms for weighted shortest paths (both APSP, and for k sources) in graphs with moderate non-negative integer weights. Our contributions include a novel pipelined strategy for computing weighted shortest paths, improvements to the distributed deterministic construction of a blocker set, and simplifications to earlier shortest path algorithms in [50, 10]. A key feature of our shortest path algorithms is that they can handle zero-weighted edges, which are known to present a challenge in the design of distributed algorithms for non-negative integer weights (see [50]). We have also present an approximate APSP algorithm that can handle zero-weighted edges.

Our work leaves a couple of major open problems. We could obtain a faster deterministic APSP algorithm with non-negative polynomially bounded integer weights if our pipelined strategy can be made to works with Gabow's scaling

technique [34]. Our current algorithm assumes that all sources see the same weight on each edge, while in the scaling algorithm each source sees a different edge weight on a given edge. While this can be handled with n different SSSP computations in conjunction with the randomized scheduling result of Ghaffari [39], it will be very interesting to see if a deterministic pipelined strategy could achieve the same result.

In Chapter 7 we present a $\tilde{O}(n^{4/3})$ round deterministic APSP algorithm that improves on the results presented in this Chapter and Chapter 5. The main component of this algorithm is a faster algorithm for computing blocker set deterministically and a new approach to propagate distance values from source nodes to blocker nodes.

Chapter 7

Faster Deterministic All Pairs Shortest Paths

7.1 Introduction

In this Chapter we present an $\tilde{O}(n^{4/3})$ rounds deterministic algorithm for the weighted APSP problem. Table 4.1 (Chapter 4) compares our result with the earlier results for this problem. All of these results as well as our new result can handle zero weight edges, and these algorithms are qualitatively different from algorithms for unweighted APSP.

Our new deterministic algorithm directly improves on the APSP algorithm in Chapter 5 (Algorithm 1, Chapter 5). The algorithm in Chapter 5 (Algorithm 1) computes Step 1 in $O(n \cdot h)$ rounds by running the distributed Bellman-Ford algorithm for h hops from each source. Our algorithm leaves Step 1 unchanged, but it improves on both Step 2 and Step 3.

For Step 2, in Chapter 5 we described a deterministic algorithm that greedily chooses vertices to add to Q at the cost of $O(n)$ rounds per vertex added, for the cleanup cost for removing paths that are covered by this newly chosen vertex; this

is after an initial start-up cost of $O(n \cdot h)$. This gives an overall cost of $O(nh + nq)$ for Step 2, where $q = |Q| = O((n/h) \cdot \log n)$. Our new contribution is to constuct Q in a sequence of $polylog(n)$ steps, where each step adds several vertices to Q. Our method incurs a cleanup cost of $O(|S| \cdot h)$ rounds per step after an initial start-up cost of $O(|S| \cdot h)$ rounds for an arbitrary source set S, thereby removing the dependence on q from this bound. ($S = V$ gives the standard setting used in previous APSP algorithms.) We achieve this by framing the computation of a small blocker set as an approximate set cover problem on a related hypergraph. We then adapt the efficient NC algorithm in Berger et al. [16] for computing an approximate minimum set cover in a hypergraph to an $\tilde{O}(|S| \cdot h)$-round CONGEST algorithm. As in [16] this involves two main parts. We first give a randomized $\tilde{O}(|S| \cdot h)$-round algorithm that computes a blocker set of expected size $\tilde{O}(n/h)$ using only pairwise independent random variables. We then derandomize this algorithm, again with an $\tilde{O}(|S| \cdot h)$-round algorithm.

For Step 3, in Chapter 5 we gave a deterministic $O(n \cdot q)$-round algorithm, and [50] gave a randomized $\tilde{O}(n \cdot \sqrt{q} + n \cdot \sqrt{h})$-round algorithm. We replace the $n \cdot \sqrt{h}$ randomized algorithm used in [50] with a simple $n \cdot h$ round algorithm (similar to Step 1). The randomized $O(n \cdot \sqrt{q})$ method in [50] computes the *reversed q-sink shortest paths* problem that appears to use randomization in a crucial manner, by invoking the randomized scheduling result of Ghaffari [39], which allows multiple algorithms to run concurrently in $O(\Delta + \kappa \cdot \log n)$ rounds, where Δ bounds the *dilation* of any of the concurrent algorithms and κ bounds the *congestion* on any edge when considering all algorithms. It is known that this result in [39] cannot be derandomized in a completely general setting. For Step 3, our contribution is to give a deterministic $\tilde{O}(n \cdot \sqrt{q})$-round algorithm for the reversed q-sink shortest paths problem. Our algorithm uses a simple round-robin pipelined approach. To obtain the desired round bound we rephrase the algorithm to work in *frames* which

allows us to establish suitable progress in the pipelining to show that it terminates in $\tilde{O}(n\cdot\sqrt{q})$ rounds. We note that the standard known results on efficiently broadcasting multiple values, and on sending or receiving messages using the routing schedule in an undirected APSP algorithm [48, 64] do not apply to this setting.

Finally we obtain the $\tilde{O}(n^{4/3})$ bound on the number of rounds by balancing the $\tilde{O}(nh)$ bound for Steps 1 and 2 with the $\tilde{O}(n\cdot\sqrt{q})$ bound for the reversed q-sink shortest path problem, as stated in the following theorem.

Theorem 7.1.1. *There is a deterministic distributed algorithm that computes APSP on an n-node graph with arbitrary non-negative edge-weights, directed or undirected, in $\tilde{O}(n^{4/3})$ rounds.*

Theorem 7.1.1 improves on prior results for deterministic APSP on weighted graphs in the CONGEST model. If randomization is allowed, the very recent result in [19] gives an $\tilde{O}(n)$-round randomized algorithm, which is close to the known lower bound of $\Omega(n)$ rounds [22], that holds even for unweighted APSP.

Roadmap. In Section 7.2 we present our overall APSP algorithm. Section 7.3 sketches our blocker set algorithm and Section 7.4 gives our pipelined algorithm for the reversed q-sink shortest path problem. Section 7.5 gives additional details about our results in Sections 7.3 and 7.4.

7.2 Overall APSP Algorithm

Algorithm 5 gives our overall APSP algorithm. In Step 1 we use the (simple) $O(n\cdot h)$-round algorithm in [8] to compute h-hop *Consistent* SSSP (CSSSP) for the vertex set V (Definition 6.4.1, Chapter 5). The advantage of using h-hop CSSSPs instead of other types of h-hop shortest paths is that the CSSSPs create a consistent collection of paths across all trees in the collection, i.e. a path from u to v is same in all trees T in the CSSSP collection $\mathcal{C}$ (in which such a path exists). We exploit this useful

property of CSSSPs throughout this chapter.

Algorithm 5 Overall APSP Algorithm

Input: number of hops $h = n^{1/3}$

1: Compute h-hop CSSSP for set V using the algorithm in Chapter 5.
2: Compute a blocker set Q of size $\tilde{O}(\frac{n}{h})$ for the h-hop CSSSP computed in Step 1 (described in Section 7.3).
3: **For each $c \in Q$ in sequence:** Compute h-hop in-SSSP rooted at c.
4: **For each $c \in Q$ in sequence:** Broadcast $ID(c)$ and the shortest path distance value $\delta_h(c, c')$ for each $c' \in Q$.
5: **Local Step at node $x \in V$:** For each $c \in Q$ compute the shortest path distance values $\delta(x, c)$ using the distance values received in Step 4.
6: Run Algorithms 10 and 11 described in Section 7.4 to propagate each distance value $\delta(x, c)$ from source $x \in V$ to blocker node $c \in Q$.
7: **For each $x \in V$ in sequence:** Compute *extended* h-hop shortest paths starting from every $c \in Q$ using Bellman-Ford algorithm (described in Section 7.5.2).

Step 2 computes a blocker set Q (Definition 5.2.1, Chapter 5). Our deterministic blocker set algorithm for Step 2 is completely different from the blocker set algorithms in Chapters 5 and 6 with significant improvement in the round complexity. We describe this algorithm in Section 7.3. Our blocker set algorithm is based on the NC approximate Set Cover algorithm of Berger et al. [16] and runs in $\tilde{O}(|S| \cdot h)$ rounds, where S is the set of sources. Previous deterministic blocker set algorithms in Chapters 5 and 6 have an additional $\tilde{O}(n \cdot |Q|)$ term in the round complexity.

In Step 3 we compute, for each $c \in Q$, the h-hop in-SSSP rooted at c, which is the set of in-coming h-hop shortest paths ending at node c. We can compute these h-hop in-SSSPs in $O(h)$ rounds per source using Bellman-Ford algorithm [15]. In Step 4 every blocker node $c \in Q$ broadcasts its ID and the corresponding h-hop shortest path distance values $\delta_h(c, c')$ for every $c' \in Q$. Step 5 is a local computation step where every node x computes its shortest path distances $\delta(x, c)$ to every $c \in Q$ using the shortest path distance values it computed and received in Steps 3 and 4 respectively.

In Step 6 every node x wants to send each shortest path distance value $\delta(x, c)$

it computed in Step 5 to blocker node $c \in Q$. This is the reversed q-sink shortest path problem, where $q = |Q|$, and is the other crucial step in our APSP algorithm. This step requires sending $\tilde{O}(n^{5/3})$ different distance values across $\tilde{O}(n^{2/3})$ different sources (using $|Q| = \tilde{O}(n^{2/3})$). A trivial solution is to broadcast all these messages in the network, resulting in a round complexity of $\tilde{O}(n^{5/3})$ rounds. However this is the only method known so far to implement this step deterministically. In Section 7.4 we give a pipelined algorithm for implementing this step more efficiently in $\tilde{O}(n^{4/3})$ rounds. After the execution of Step 6 every blocker node $c \in Q$ knows its shortest path distance from every node $x \in V$.

Finally, in Step 7 for every source $x \in V$, we run Bellman-Ford algorithm for h hops with distance values $\delta(x, c)$ used as the initialization values at every blocker node $c \in Q$. These constructed paths are also known as *extended* h-hop shortest paths [50]. After this step, each $t \in V$ knows the shortest path distance value $\delta(x, t)$ from every source $x \in V$, which gives the desired APSP output. We describe Step 7 in Section 7.5.2.

Proof of Theorem 7.1.1. Fix a pair of nodes x and t. If the shortest path from x to t has less than h hops, then $\delta(x, t) = \delta_h(x, t)$ and the correctness is straightforward from Lemma 6.4.2.

Otherwise, we can divide the shortest path from x to t into subpaths x to c_1, c_1 to c_2, ..., c_l to t where $c_i \in Q$ for $1 \leq i \leq l$ and each of these subpaths have hop-length at most h. Since x knows $\delta_h(x, c_1)$ from Step 3 and $\delta_h(c_i, c_{i+1})$ distance values from Step 4, it can correctly compute $\delta(x, c_l)$ distance value in Step 5. And from Lemmas 7.4.1 and 7.4.4, c_l knows the distance value $\delta(x, c_l)$ after Step 6. Since the shortest path from c_l to t has hop-length at most h, from Lemma 7.5.7 t will compute $\delta(x, t)$ in Step 7.

Step 1 runs in $O(nh) = O(n^{4/3})$ rounds (Lemma 6.4.3, Chapter 5). In Section 7.3, we will give an $\tilde{O}(nh) = \tilde{O}(n^{4/3})$ rounds algorithm to compute a blocker set

of size $q = \tilde{O}(\frac{n}{h}) = \tilde{O}(n^{2/3})$ (Step 2). Step 3 takes $O(|Q| \cdot h) = \tilde{O}(n)$ rounds using Bellman-Ford algorithm (Lemma 6.4.3, Chapter 5). Since $|Q|^2 = \tilde{O}(\frac{n^2}{h^2}) = \tilde{O}(n^{4/3})$, Step 4 takes $\tilde{O}(n^{4/3})$ rounds using Lemma 5.2.5 (Chapter 5). Step 5 is local computation and has no communication. From Lemmas 7.4.1 and 7.4.5, Step 6 takes $\tilde{O}(n^{4/3})$ rounds and Step 7 can be computed in $O(nh) = O(n^{4/3})$ rounds using Lemma 7.5.7. Hence the overall algorithm runs in $\tilde{O}(n^{4/3})$ rounds. □

7.3 Computing Blocker Set

In this section we describe our algorithm to compute a small blocker set. We frame this problem as that of finding a small set cover in an associated hypergraph. We then adapt the efficient NC algorithm for finding a provably good approximation to this NP-hard problem given in Berger et al. [16] to obtain our deterministic distributed algorithm.

As in [16] our algorithm has two parts. We first present a randomized algorithm to find a blocker set of size $\tilde{O}(n/h)$ in $\tilde{O}(|S| \cdot h)$ rounds using only pairwise independence. This is described in Section 7.3.1. Then in Section 7.3.2 we describe how to use the exhaustive search technique of Luby [68] along with the ideas from [16] to derandomize this algorithm, again in $\tilde{O}(|S| \cdot h)$ rounds. In our overall APSP algorithm $S = V$ but we will also use this algorithm in Section 7.4 with a different set for S.

7.3.1 Randomized Algorithm for Computing Blocker Set

Given a hypergraph $H = (V, F)$, a subset of vertices R is a set cover for H if R contains at least one vertex in every edge in F. Computing a set cover of minimum size is NP-hard. Berger et al. [16] gave an efficient NC algorithm to compute an $O(\log n)$ approximation to the minimum set cover.

We can map the problem of computing a minimum blocker set for an h-hop

Table 7.1: Notations

Global Parameters:	
$\mathcal{C}$	h-hop CSSSP collection
S	set of sources in $\mathcal{C}$
h	number of hops in a path
n	number of nodes
ϵ, δ	positive constants $\leq 1/12$
Q	blocker set (being constructed)
$score(v)$	number of root-to-leaf paths in $\mathcal{C}$ that contain v (local var. at v)
V_i	set of nodes v with $score(v) \geq (1+\epsilon)^{i-1}$
P_i	set of paths in $\mathcal{C}$ with at least one node in V_i
P_{ij}	set of paths in P_i with at least $(1+\epsilon)^{j-1}$ nodes in V_i
$score^{ij}(v)$	number of paths in P_{ij} that contain v (local var. at v)

CSSSP collection $\mathcal{C}$ in a graph $G = (V, E)$ to the minimum set cover problem in the hypergraph $H = (V, F)$ where V remains the vertex set of G and each edge in F consists of the vertices in a root-to-leaf path in a tree in $\mathcal{C}$. This hypergraph has n vertices and at most $n \cdot |S|$ edges, where S is the number of sources (i.e., trees) in $\mathcal{C}$. Each edge in F has exactly h vertices. We now use this mapping to rephrase the algorithm in [16] in our setting, and we derive an $\tilde{O}(|S| \cdot h)$-round randomized algorithm to compute a blocker set of expected size within $O(\log n)$ of the optimal size, using only pairwise independent random variables. Since we know there exists a blocker set of size $O((n/h) \cdot \log n)$ the size of the blocker set constructed by this randomized algorithm is $\tilde{O}(n/h)$.

Our randomized blocker set method is in Algorithm 6. Table 7.1 presents the notation we use for this section. In Step 1 for each node v we compute $score(v)$, the number of h-hop shortest paths in CSSSP collection $\mathcal{C}$ that contain node v. This can be done in $O(|S| \cdot h)$ rounds for all nodes $v \in V$ using Algorithm 3 in Chapter 5. Our algorithm proceeds in stages from $i = \log_{1+\epsilon} n^2$ down to 2 (Steps 2-17), where ϵ is a small positive constant $\leq 1/12$, such that at the start of stage i, all nodes in V have score value at most $(1+\epsilon)^i$ and in stage i we focus on V_i, the set of nodes v with

Algorithm 6 Randomized Blocker Set Algorithm

Input: S: set of source nodes; h: number of hops; $\mathcal{C}$: collection of h-hop CSSSP for set S; ϵ, δ: positive constants $\leq 1/12$

1: Compute $score(v)$ for all nodes $v \in V$ using an algorithm from [10].
2: **for stage** $i = \log_{1+\epsilon} n^2$ **down to** 1 **do** ▷ All nodes have score less than $(1+\epsilon)^i$
3: Compute V_i and broadcast it using the algorithm described in Section 7.5.1.1.
4: Compute P_i^v (at each $v \in V$) using Algorithm 13 (Section 7.5.1.1).
5: **for phase** $j = \log_{1+\epsilon} h$ **down to** 1 **do**
6: **while** there is a path in P_i with atleast $(1+\epsilon)^{j-1}$ nodes in V_i **do**
7: Compute (a) P_{ij}^v (at each $v \in V$) using Algorithm 14 and (b) $|P_{ij}|$ using Algorithm 15 (Section 7.5).
8: Compute $score^{ij}(v)$ for all nodes $v \in V_i$ (using Algorithm 3 (Chapter 5)) and broadcast $score^{ij}(v)$ values.
9: **if** there exists $c \in V_i$ such that $score^{ij}(c) > (\delta^3/(1+\epsilon)) \cdot |P_{ij}|$ **then**
10: **Local Step at $v \in V$:** add c to Q. Break ties with $score^{ij}$ value and node ID.
11: **else** ▷ Run a selection procedure to select a set of nodes
12: **Local Step at $v \in V_i$:** add v to set A with probability $p = \delta/(1+\epsilon)^j$ (pairwise independently).
13: **For each $v \in V_i$:** node v broadcast $ID(v)$ if it added itself to A in previous step.
14: **For each $v \in V$:** node v broadcast the number of paths in P_i^v and P_{ij}^v covered by this set A.
15: **Local Step at $v \in V$:** Check if A is a good set and if so, add A to Q. Otherwise, go back to Step 12.
16: **For each $x \in S$ in sequence:** Remove subtrees rooted at $c' \in Q$ using Algorithm 16 (Section 7.5.1.4).
17: Re-compute $score(v)$ for all nodes v and re-construct sets V_i and P_i as described in Steps 3 and 4.

score value greater than $(1+\epsilon)^{i-1}$. (This ensures that the nodes that are added to the blocker set have their score values near the maximum score value). Let P_i be the set of paths in $\mathcal{C}$ that contain a vertex in V_i and let P_i^v be the set of paths in P_i with v as the leaf node. These sets are readily computed in $O(n)$ and $O(|S| \cdot h)$-rounds respectively (see Section 7.5.1.1).

Similar to [16], in order to ensure that the average number of paths covered by the newly chosen blocker nodes is near the maximum score value, we further divide our algorithm for stage i into a sequence of $\log_{1+\epsilon} h = \log_{1+\epsilon} n^{1/3}$ phases, where in each phase j we focus on the paths in P_i with at least $(1+\epsilon)^{j-1}$ nodes in V_i. We call this set of paths P_{ij}. We maintain that at the start of phase j, every path in P_i has at most $(1+\epsilon)^j$ nodes in V_i. We now describe our algorithm for phase j (Steps 5-17). The algorithm for phase j consists of a series of selection steps (Steps6-17) (similar to [16]) which are performed until there are no more paths in P_{ij}.

Now we describe how we select nodes to add to blocker set Q. Let δ be some fixed positive constant less than or equal to 1/12. In Step 9 we check if there exists a node v which covers at least $\delta^3/(1+\epsilon)$ fraction of paths in P_{ij} and if so, we add this node to the blocker set in Step 10. In case of multiple such nodes, we pick the one with the maximum $score^{ij}$ value and break ties using node IDs. Otherwise in Step 12, we randomly pick every node with probability $\delta/(1+\epsilon)^j$, pairwise independently, and form a set A. In Step 15 we check if A is a good set, otherwise we try again and form a new set A in Step 12. As in [16] we define the notion of a good set as given below and we will later show that A is a good set with probability at least 1/8.

Definition 7.3.1. *A set of nodes $A \subseteq V_i$ is a good set if A covers at least $(1+\epsilon)^i \cdot (1-3\delta-\epsilon) \cdot |A|$ paths in P_i and at least a $\delta/2$ fraction of paths from P_{ij}.*

Before the next selection step, we remove the paths covered by these newly chosen node from the collection $\mathcal{C}$ along with recomputing the score values and sets V_i and P_i (Steps 16-17).

7.3.1.1 Analysis of the Randomized Algorithm

Similar to [16] we get the following Lemmas which give us a bound on the number of selection steps and a bound on the size of Q. Table 7.2 presents the notation we use in our analysis in this section.

Lemma 7.3.2. *The set Q constructed in Algorithm 6 is a blocker set for the CSSSP collection $\mathcal{C}$.*

Proof. To show that Q is a blocker set, we need to show that the computed blocker set Q indeed covers all paths in the CSSSP collection $\mathcal{C}$. The while loop in Steps 6-17 runs as long as there is a path in P_i with at least $(1+\epsilon)^{j-1}$ nodes in V_i and since this loop terminated for $i = 1$ and $j = 1$, it implies that there is no path in $\mathcal{C}$ which is not covered by some node in Q. □

Lemma 7.3.3. *If the check in Step 9 fails, then $|V_i| > \frac{(1+\epsilon)^j}{\delta^3}$.*

Proof. Since no node in V_i covers a $\frac{\delta^3}{(1+\epsilon)}$ fraction of paths from P_{ij}, hence the total $score_{ij}$ values (defined in Step 8) for all nodes in V_i has value at most $|V_i| \cdot \frac{\delta^3}{(1+\epsilon)} \cdot |P_{ij}|$. And since every path in P_{ij} has atleast $(1+\epsilon)^{j-1}$ nodes in V_i,

$$|V_i| \cdot \frac{\delta^3}{(1+\epsilon)} \cdot |P_{ij}| > |P_{ij}| \cdot (1+\epsilon)^{j-1}$$

This establishes that $|V_i| > \frac{(1+\epsilon)^j}{\delta^3}$. □

Lemma 7.3.4. *The set A constructed in Step 12 of Algorithm 6 has size at most $(\delta + 2\delta^2) \cdot \frac{|V_i|}{(1+\epsilon)^j}$ and atleast $(\delta - 2\delta^2) \cdot \frac{|V_i|}{(1+\epsilon)^j}$ with probability at least $3/4$.*

Proof. Consider random variable X_v where $X_v = 1$ if v is present in A, otherwise $X_v = 0$. Thus $\sum_{v \in V_i} X_v$ denotes the size of A. We now calculate its expectation and variance.

Table 7.2: List of Notations Used in the Analysis of the Randomized Algorithm

Q	blocker set (being constructed)
$\mathcal{C}$	h-hop CSSSP collection
S	set of sources in $\mathcal{C}$
h	number of hops in a path
n	number of nodes
V_i	set of nodes v with $score(v) \geq (1+\epsilon)^{i-1}$
P_i	set of paths in $\mathcal{C}$ with at least one node in V_i
P_{ij}	set of paths in P_i with at least $(1+\epsilon)^{j-1}$ nodes in V_i
ϵ, δ	positive constants $\leq 1/12$
A	set constructed in Step 12
X_v	1 if v is present in A, otherwise 0
Y_1	$\sum_{p \in P_i} \sum_{v \in V_i \cap p} X_v$
Y_2	$\sum_{p \in P_i} \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}$
Y_3	$\sum_{p \in P_{ij}} \sum_{v \in V_i \cap p} X_v$
Y_4	$\sum_{p \in P_{ij}} \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}$
$n_{V_i,p}$	number of nodes from V_i in p
$n_{v,P_{ij}}$	number of paths in P_{ij} that contain node v
$score(v)$	number of root-to-leaf paths in $\mathcal{C}$ that contain v (local var. at v)
$score^{ij}(v)$	number of paths in P_{ij} that contain v (local var. at v)

$$E[\sum_{v \in V_i} X_v] = |V_i| \cdot \frac{\delta}{(1+\epsilon)^j} \tag{7.1}$$

$$Var[\sum_{v \in V_i} X_v] = |V_i| \cdot Var[X_v] \leq |V_i| \cdot E[X_v^2] = |V_i| \cdot \frac{\delta}{(1+\epsilon)^j} \tag{7.2}$$

We now use Chebyshev's inequality to get an upper bound on the size of A. Using Chebyshev's inequality the following holds with probability at least 3/4:

$$\begin{aligned}
||A| - E[|A|]| &\leq 2\sqrt{Var[|A|]} \\
&\leq 2\sqrt{|V_i| \cdot \frac{\delta}{(1+\epsilon)^j}} \\
&\leq 2 \cdot |V_i| \cdot \frac{\delta^2}{(1+\epsilon)^j} \quad (\text{by Lemma 7.3.3 } \frac{1}{|V_i|} < \frac{\delta^3}{(1+\epsilon)^j}) \\
|A| &\leq |V_i| \cdot \frac{\delta}{(1+\epsilon)^j} + 2 \cdot |V_i| \cdot \frac{\delta^2}{(1+\epsilon)^j}
\end{aligned}$$

Using the above analysis we can also show that $|A| \geq (\delta - 2\delta^2) \cdot \frac{|V_i|}{(1+\epsilon)^j}$ with probability at least 3/4. □

Lemma 7.3.5. *The set A constructed in Step 12 of Algorithm 6 covers at least $|A| \cdot (1+\epsilon)^i \cdot (1 - 3\delta - \epsilon)$ paths in P_i with probability at least 1/2.*

Proof. Consider the 0-1 random variable X_v which is equal to 1 if $v \in A$. A path p is covered by A if $v \in A$, i.e. $X_v = 1$ for some $v \in V_i \cap p$. To get a lower bound on the number of paths covered by A, we use the term $\sum_{v \in V_i \cap p} X_v - \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}$ to denote if a path p is covered by A or not. Note that this term has value at most 1 which is attained when either 1 or 2 nodes from p are picked in A and otherwise the value is non-positive. Thus the term $\sum_{p \in P_i} [\sum_{v \in V_i \cap p} X_v - \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}]$ gives a lower bound on the number of paths covered by A in P_i. Let this term be Y. Note

that even though the lower bound achieved using this term is very weak, improving it further will not improve the overall bound on the size of A by more than a polylog factor (Lemma 7.3.4).

Now we show that value of Y is $\geq |A| \cdot (1+\epsilon)^i \cdot (1-3\delta-\epsilon)$ with probability atleast $1/2$.

We first split Y into Y_1 and Y_2 where $Y_1 = \sum_{p \in P_i} \sum_{v \in V_i \cap p} X_v$ and $Y_2 = \sum_{p \in P_i} \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}$.

We first get a lower bound on the term Y_1.

$$\begin{aligned}
Y_1 &= \sum_{p \in P_i} \sum_{v \in V_i \cap p} X_v \\
&= \sum_{v \in V_i} \sum_{\{p \in P_i : v \in p\}} X_v \\
&\geq (1+\epsilon)^{i-1} \cdot \sum_{v \in V_i} X_v \quad \text{(since every node in } V_i \text{ lies in } \geq (1+\epsilon)^{i-1} \text{ paths in } P_i) \\
&= (1+\epsilon)^{i-1} \cdot |A|
\end{aligned}$$

We now need to get an upper bound on the term Y_2. We first compute an upper bound on $E[Y_2]$ and then use Markov inequality to get an upper bound on Y_2. (Let $n_{V_i,p}$ denotes the number of nodes from V_i in p. Clearly $n_{V_i,p} \leq (1+\epsilon)^j$)

$$\begin{aligned}
E[Y_2] &= \sum_{p \in P_i} \sum_{v,v' \in V_i \cap p} E[X_v \cdot X_{v'}] \\
&= \sum_{p \in P_i} \sum_{v,v' \in V_i \cap p} E[X_v] \cdot E[X_{v'}] \quad \text{(follows since } X_v \text{ and } X'_v \text{ are pairwise independent)} \\
&= \sum_{p \in P_i} \binom{n_{V_i,p}}{2} \left(\frac{\delta}{(1+\epsilon)^j}\right)^2
\end{aligned}$$

$$
\begin{aligned}
&\leq (1+\epsilon)^j \cdot \sum_{p \in P_i} \frac{n_{V_i,p}}{2} \left(\frac{\delta}{(1+\epsilon)^j} \right)^2 \quad (\text{since } n_{V_i,p} \leq (1+\epsilon)^j) \\
&\leq (1+\epsilon)^j \cdot \frac{\sum_{v \in V_i} score(v)}{2} \cdot \left(\frac{\delta}{(1+\epsilon)^j} \right)^2 \\
&\leq (1+\epsilon)^j \cdot \frac{|V_i|}{2} \cdot \max_{v \in V_i} score(v) \cdot \left(\frac{\delta}{(1+\epsilon)^j} \right)^2 \\
&\leq \frac{|V_i|}{2} \cdot (1+\epsilon)^{i-j} \cdot \delta^2 \qquad (7.3)
\end{aligned}
$$

Now using Markov inequality we get the following upper bound on Y_2 with probability atleast 3/4:

$$Y_2 \leq 4E[Y_2] \leq 2\delta^2 \cdot (1+\epsilon)^{i-j} \cdot |V_i|$$

Since $|A| \geq (\delta - 2\delta^2) \cdot \frac{|V_i|}{(1+\epsilon)^j}$ with probability at least 3/4 by Lemma 7.3.4, $Y_2 \leq 2\delta^2 \cdot (1+\epsilon)^i \cdot \frac{|A|}{(\delta - 2\delta^2)}$ with probability at least 1/2.

Combining the bounds for Y_1 and Y_2 we get the following lower bound on Y with probability at least 1/2:

$$
\begin{aligned}
Y &= Y_1 - Y_2 \\
&\geq (1+\epsilon)^{i-1} \cdot |A| - 2\delta^2 \cdot (1+\epsilon)^i \cdot \frac{|A|}{(\delta - 2\delta^2)} \\
&= (1+\epsilon)^i \cdot |A| \cdot (\frac{1}{1+\epsilon} - \frac{2\delta}{1-2\delta}) \\
&= (1+\epsilon)^i \cdot |A| \cdot (1 - \frac{\epsilon}{1+\epsilon} - \frac{3\delta}{3/2 - 3\delta}) \\
&\geq (1+\epsilon)^i \cdot |A| \cdot (1 - \epsilon - 3\delta)
\end{aligned}
$$

This establishes the lemma. □

Lemma 7.3.6. *The set A constructed in Step 12 of Algorithm 6 covers at least a*

$\delta/2$ fraction of paths in P_{ij} with probability at least $5/8$.

Proof. Similar to the proof of Lemma 7.3.5 we can lower bound the number of paths covered by set A in P_{ij} by the term $\sum_{p \in P_{ij}} [\sum_{v \in V_i \cap p} X_v - \sum_{v,v' \in V_i \cap p} X_v \cdot X_{v'}]$. Let this term be Y', with first term Y_3 and the second term Y_4. As noted in the proof of Lemma 7.3.5, this term gives a very weak lower bound (however the sum here is over the paths in the set P_{ij} instead of P_i) and it is sufficient to get our desired bound on the size of A.

Note that even though the lower bound achieved using this term is very weak, improving it further will not improve the overall bound on the size of A by more than a polylog factor (Lemma 7.3.4).

Now we need to show that $Y' \geq \frac{\delta}{2} \cdot |P_{ij}|$ with probability at least $5/8$.

We first give a lower bound on Y_3. To get the lower bound, we first compute a lower bound on $E[Y_3]$ and an upper bound on $Var[Y_3]$ and then use Chebyshev's inequality. (Let $n_{v,P_{ij}}$ represent the number of paths in P_{ij} that contain node v. Since no node covers at least $\frac{\delta^3}{(1+\epsilon)}$ fraction of paths in P_{ij}, $n_{v,P_{ij}} < \frac{\delta^3}{(1+\epsilon)}$)

$$
\begin{aligned}
E[Y_3] &= E[\sum_{p \in P_{ij}} \sum_{v \in V_i \cap p} X_v] \\
&\geq E[\sum_{p \in P_{ij}} (1+\epsilon)^{j-1} \cdot X_v] \quad \text{(since every path in } P_{ij} \text{ has atleast } (1+\epsilon)^{j-1} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{nodes from } V_i) \\
&= (1+\epsilon)^{j-1} \cdot |P_{ij}| \cdot \frac{\delta}{(1+\epsilon)^j} \\
&= |P_{ij}| \cdot \frac{\delta}{(1+\epsilon)}
\end{aligned}
$$

$$
\begin{aligned}
Var[Y_3] &= Var[\sum_{p \in P_{ij}} \sum_{v \in V_i \cap p} X_v] \\
&= Var[\sum_{v \in V_i} \sum_{\{p \in P_{ij} : v \in p\}} X_v] \\
&= Var[\sum_{v \in V_i} n_{v,P_{ij}} X_v] \\
&= \sum_{v \in V_i} n_{v,P_{ij}}^2 \cdot Var[X_v] \quad \text{(linearity of variance follows since } X_v\text{'s are} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{pairwise independent)} \\
&\leq \frac{\delta}{(1+\epsilon)^j} \cdot \frac{\delta^3}{(1+\epsilon)} \cdot |P_{ij}| \cdot \sum_{v \in V_i} n_{v,P_{ij}} \quad \left(\text{since } n_{v,P_{ij}} < \frac{\delta^3}{(1+\epsilon)} \cdot |P_{ij}|\right) \\
&\leq \frac{\delta^4}{(1+\epsilon)^{j+1}} \cdot |P_{ij}| \cdot |P_{ij}| \cdot (1+\epsilon)^j \quad \text{(since every path in } P_{ij} \text{ has at most} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (1+\epsilon)^j \text{ nodes from } V_i) \\
&\leq \delta^4 \cdot |P_{ij}|^2 \qquad (7.4)
\end{aligned}
$$

We now use Chebyshev's inequality to get a lower bound on the value of Y_3. Using Chebyshev's inequality the following holds with probability at least 7/8:

$$
\begin{aligned}
|Y_3 - E[Y_3]| &\leq 2\sqrt{2}\sqrt{Var[Y_3]} \\
Y_3 &\geq E[Y_3] - 2\sqrt{2}\delta^2 \cdot |P_{ij}| \\
&\geq |P_{ij}| \cdot \frac{\delta}{(1+\epsilon)} - 2\sqrt{2}\delta^2 \cdot |P_{ij}|
\end{aligned}
$$

We now need to get an upper bound on the term Y_4. We first compute an upper bound on $E[Y_4]$ and then use Markov inequality to get an upper bound on Y_4.

$$
\begin{aligned}
E[Y_4] &= \sum_{p \in P_{ij}} \sum_{v,v' \in V_i \cap p} E[X_v \cdot X_{v'}] \\
&= \sum_{p \in P_{ij}} \sum_{v,v' \in V_i \cap p} E[X_v] \cdot E[X_{v'}] \quad \text{(follows since } X_v \text{ and } X'_v \text{ are pairwise independent)} \\
&\leq |P_{ij}| \cdot \frac{(1+\epsilon)^{2j}}{2} \cdot \left(\frac{\delta}{(1+\epsilon)^j} \right)^2 \quad \text{(since there are at most } (1+\epsilon)^j \text{ nodes from } V_i \text{ in any path in } P_{ij}) \\
&= |P_{ij}| \cdot \frac{\delta^2}{2} \qquad (7.5)
\end{aligned}
$$

Now using Markov inequality we get the following upper bound on Y_4 with probability atleast 3/4:

$$Y_4 \leq 4E[Y_4] \leq 2\delta^2 \cdot |P_{ij}|$$

Combining the bounds for Y_3 and Y_4 we get the following lower bound on Y' with probability at least 5/8:

$$
\begin{aligned}
Y' &= Y_3 - Y_4 \\
&\geq |P_{ij}| \cdot \frac{\delta}{(1+\epsilon)} - 2\sqrt{2}\delta^2 \cdot |P_{ij}| - 2\delta^2 \cdot |P_{ij}| \\
&\geq |P_{ij}| \cdot \delta \cdot (1 - \epsilon - 5\delta) \\
&\geq |P_{ij}| \cdot \frac{\delta}{2} \quad (\text{since } \epsilon, \delta \leq 1/12)
\end{aligned}
$$

This establishes the lemma. □

Lemma 7.3.7. *The set A constructed in Step 12 is a good set with probability at least* 1/8.

Proof. This is immediate from Lemma 7.3.5 and 7.3.6. □

Lemma 7.3.8. *The while loop in Steps 6-17 runs for at most* $O(\log^3 n/(\delta^3 \cdot \epsilon^2))$ *iterations in total.*

Proof. The while loop runs until P_{ij} is non-empty, i.e. there exists a path in P_i with atleast $(1+\epsilon)^{j-1}$ nodes in V_i. In each iteration, the algorithm either covers at least $\frac{\delta^3}{(1+\epsilon)}$ fraction of paths in P_{ij} (if node c is added to blocker set Q in Step 10) or at least $\frac{\delta}{2}$ fraction of paths from P_{ij} (if set A is added to Q in Step 15). Since there are at most n^2 paths and each iteration of the while loop covers at least $\frac{\delta^3}{(1+\epsilon)}$ fraction of P_{ij}, there are at most $O\left(\frac{\log n^2}{\log\left(\frac{1}{1-\frac{\delta^3}{(1+\epsilon)}}\right)}\right) = O\left(\frac{(1+\epsilon)\log n}{\delta^3}\right) = O\left(\frac{\log n}{\delta^3}\right)$ iterations. Since both the inner and outer for loop runs for $O(\log_{1+\epsilon} n) = O(\frac{\log n}{\epsilon})$ iterations, this establishes the lemma. □

Lemma 7.3.9. *Each iteration of the inner for loop (Steps 5-17) in Algorithm 6 takes* $\tilde{O}\left(\frac{|S|\cdot h}{\delta^3}\right)$ *rounds in expectation.*

Proof. We first show that each iteration of the while loop in Steps 6-17 takes $O(|S|\cdot h)$ rounds in expectation. Step 7 takes $O(|S|\cdot h)$ rounds by Lemmas 7.5.3 and 7.5.3 and so does Step 8 [10] and by Lemma 5.2.5. The check in Step 9 involves no communication and so does Step 10, since every node knows the $score_{ij}$ values for every other node and also the value of $|P_{ij}|$, i.e. the number of paths that belong to P_{ij}. Steps 12 and 15 are also local steps and does not involve any communication. Steps 13 and 14 involves broadcasting at most $O(n)$ messages and hence takes $O(n)$ rounds using Lemma 5.2.5. Since by Lemma 7.3.7 the set A constructed in Step 12 is good with probability at least 1/8, Steps 12-15 are executed $O(1)$ times in expectation. Step 17 takes $O(|S|\cdot h)$ rounds [10] and using Lemma 7.5.2. Since the while loop runs for at most $O\left(\frac{\log n}{\delta^3}\right)$ iterations (by Lemma 7.3.8), this establishes the lemma. □

Lemma 7.3.10. *The blocker set Q constructed by Algorithm 6 has size $O(\frac{n \log n}{h})$.*

Proof. As shown in [58, 10] the size of the blocker set computed by an optimal greedy algorithm is $\Theta(\frac{n \ln p}{h})$, where p is the number of paths that need to be covered. We will now argue that the blocker set constructed by Algorithm 6 is at most a factor of $\frac{1}{(1-3\delta-\epsilon)}$ larger than the greedy solution, thus showing that the constructed blocker set Q has size at most $O(\frac{n \ln p}{h} \cdot \frac{1}{(1-3\delta-\epsilon)}) = \tilde{O}(\frac{n \log n}{h})$ since $p \leq n^2$ and $0 < \delta, \epsilon \leq \frac{1}{12}$.

The blocker set Q constructed by Algorithm 6 has 2 types of nodes: (1) node c added in Step 10, (2) set of nodes A added in Step 12. Since the while loop in Steps 6-17 runs for at most $O\left(\frac{\log^3 n}{\delta^3 \cdot \epsilon^2}\right)$ iterations (by Lemma 7.3.8), hence there are at most $O\left(\frac{\log^3 n}{\delta^3 \cdot \epsilon^2}\right)$ nodes of type 1. Since $\frac{\log^3 n}{\delta^3 \cdot \epsilon^2} = o(\frac{n}{h})$, hence we only need to bound the number of nodes added in Steps 12-15.

Since A is a good set, by Lemma 7.3.7 the number of paths covered by A is at least $|A| \cdot (1+\epsilon)^i \cdot (1-3\delta-\epsilon)$, where $(1+\epsilon)^i$ is the maximum possible score value across all nodes in V (in the current iteration). Since maximum possible score value is $(1+\epsilon)^i$, any greedy solution must add atleast $|A| \cdot (1-3\delta-\epsilon)$ nodes in the blocker set to cover these paths. Hence the choice of A is at most a factor of $\frac{1}{(1-3\delta-\epsilon)}$ larger than the greedy solution. This establishes the lemma. □

Lemma 7.3.11. *Algorithm 6 computes the blocker set Q in $\tilde{O}(|S| \cdot h/(\epsilon^2 \delta^3))$ rounds, in expectation.*

Proof. Step 1 runs in $O(|S| \cdot h)$ rounds [10]. The for loop in Steps 2-17 runs for $\log_{1+\epsilon} n^2 = O\left(\frac{\log n}{\epsilon}\right)$ iterations. Each iteration takes $\tilde{O}\left(\frac{|S| \cdot h}{\epsilon \delta^3}\right)$ rounds in expectation: Step 4 is readily seen to run in $O(|S| \cdot h)$ rounds (Lemma 7.5.2). The inner for loop in Steps 5-17 runs for $\log_{1+\epsilon} h = O\left(\frac{\log n}{\epsilon}\right)$ iterations, with each iteration taking $\tilde{O}\left(\frac{|S| \cdot h}{\delta^3}\right)$ rounds in expectation using Lemma 7.3.9. □

7.3.2 Deterministic Blocker Set Algorithm

The only place where randomization is used in Algorithm 6 is in Steps 12-15, where a good set A (see Definition 7.3.1) is chosen. Fortunately, the X_v's are pairwise-independent random variables, where $X_v = 1$ if $v \in A$ and 0 otherwise. We use a linear-sized sample space [11, 60, 20] for generating pairwise independent random variables and then find a good sample point(i.e., a good set A) in this $O(n)$-sized sample space in $O(|S| \cdot h + n)$ rounds.

Note that a trivial solution is to run the inner loop (Steps 12-15) in the randomized blocker set algorithm for each sample point in the $O(n)$ sample space until a good set is identified. However in the worst case, we may need to run this loop $O(n)$ times instead of just a constant number of times in expectation, and that would worsen the round complexity by a factor of n.

Algorithm 7, our derandomized algorithm, works as follows. Recall that P_i^v and P_{ij}^v denote the set of paths in P_i and P_{ij}, respectively, that have v as the leaf node. We start with create an incoming BFS tree rooted at l (Step 1, Alg. 7). We assume that the X values are enumerated in order and every node knows this enumeration. Let $X^{(\mu)}$ refers to the μ-th vector in this enumeration and let $\sigma_{P_i,v}^{(\mu)}$ and $\sigma_{P_{ij},v}^{(\mu)}$ refers to the number of paths covered by $X^{(\mu)}$ in sets P_i^v and P_{ij}^v respectively. Similarly let $\nu_{P_i}^{(\mu)}$ and $\nu_{P_{ij}}^{(\mu)}$ refers to the total number of paths covered by $X^{(\mu)}$ in sets P_i and P_{ij} respectively. In Step 2 (Alg. 7), the leader l receive sums of the $\nu_{P_i,u}$ and $\nu_{P_{ij},u}$ values for all sample points from the nodes u using the algorithm in Section 7.3.3. The leader then is able to compute the number of paths covered in both P_i and P_{ij} for each μ and then picks one that satisfies the good set criterion (Step 3, Alg. 7). It then broadcasts the corresponding X vector to every node in the network (Step 4, Alg. 7). Algorithm 7 describes the pseudocode of this algorithm.

Lemma 7.3.12. *The leader node l can identify a good sample point $X \in \{0,1\}^{|V_i|}$, and thus a good set A in $O(|S| \cdot h + n)$ rounds.*

Algorithm 7 Deterministic Algorithm for picking good set A

Input: h: number of hops; S: set of sources; $\mathcal{C}$: h-hop CSSSP collection; $X^{(\mu)}$: μ-th vector in sample space; P_i^v: set of paths in P_i with v as the leaf node; P_{ij}^v: set of paths in P_{ij} with v as the leaf node

1: Compute BFS in-tree T rooted at leader l.
2: Compute $\sigma_{P_i,u}^{(\mu)}$ and $\sigma_{P_{ij},u}^{(\mu)}$ terms locally at each $v \in V$, for each sample point μ, and then using the pipelined algorithm in Section 7.3.3, send these values to the leader l.
3: **Local Step at l:** For each $1 \leq \mu \leq n$, compute $\nu_{P_i}^{(\mu)}$ and $\nu_{P_{ij}}^{(\mu)}$. Let μ' be such that $X^{(\mu')}$ corresponds to good set A (in case of ties, pick the highest one).
4: Node l broadcast $X^{(\mu')}$ values. (This corresponds to good set A)

Proof. Step 1 computes the incoming BFS tree rooted at leader node l in $O(n)$ rounds. Step 2 takes $O(n)$ rounds by Lemmas 7.3.14 and 7.3.15 . Step 3 is a local step and involves no communication. Step 4 involves an all-to-all broadcast of at most n messages and thus takes $O(n)$ rounds using Lemma 5.2.5. □

Let Algorithm 2′ be the blocker set algorithm obtained after replacing Steps 12-15 in Algorithm 6 with the deterministic algorithm for generating a good set A (Algorithm 7). Lemma 7.3.12 together with Lemma 7.3.11, gives us the following Corollary.

Corollary 7.3.13. *Algorithm 2′ computes the blocker set Q deterministically in $\tilde{O}(\frac{|S|\cdot h}{\epsilon^2\delta^3})$ rounds.*

7.3.3 Distributed Computation of Terms ν_{P_i} and $\nu_{P_{ij}}$

In this Section we describe a simple pipelined algorithm to compute ν_{P_i} and $\nu_{P_{ij}}$ terms at leader node l. Both algorithms are similar to an algorithm in [10] (for computing 'initial scores'). Recall that $\sigma_{P_i,v}^{(\mu)}$ refers to the number of paths in P_i^v covered by the sample point $X^{(\mu)}$ and $\sigma_{P_{ij},v}^{(\mu)}$ refers to the total number of paths in P_{ij}^v covered by the sample point $X^{(\mu)}$. Let $\nu_{P_i,v}^{(\mu)}$ refers to the sum total of the $\sigma_{P_i,w}^{(\mu)}$ values of all descendant nodes w of v and similarly let $\nu_{P_{ij},v}^{(\mu)}$ refers to the sum total

of the $\sigma^{(\mu)}_{P_{ij},w}$ values of all descendant nodes w of v. Also recall from Section 7.3.2 that $\nu^{(\mu)}_{P_i}$ and $\nu^{(\mu)}_{P_{ij}}$ refers to the total number of paths covered by $X^{(\mu)}$ in sets P_i and P_{ij} respectively. Table 7.3 presents the notations that we use in this Section.

7.3.3.1 Computing ν_{P_i}

Consider computing the $\nu^{(\mu)}_{P_i}$ terms for each sample point μ, at leader node l (Algorithm 8) ($\nu_{P_{ij}}$ can be computed similarly). First every node v initializes its $\nu^{(\mu)}_{P_i,v}$ value, for each sample point μ, in Step 1. Recall that we assume that all X values are enumerated in order and every node knows this enumeration. In round $n-1-h+\mu$, the node u at height h sends its corresponding $\nu_{P_i,u}$ value for $X^{(\mu)}$ (Step 3) along with the total value of ν_{P_i} it received from its children for $X^{(\mu)}$ (Steps 5-9). Leader node l then computes the total sum $\nu^{(\mu)}_{P_i}$ for each sample point μ, by summing up the received $\nu^{(\mu)}_{P_i,w}$ values from all its children w in Step 10. In Lemma 7.3.14 we show that leader l correctly computes ν_{P_i} values for all $X^{(\mu)}$'s in $O(n)$ rounds.

Algorithm 8 COMPUTE-ν_{P_i}: Compute sum of ν_{P_i} values at leader node l

Input: h: number of hops; S: set of sources; $\mathcal{C}$: h-hop CSSSP collection; $X^{(\mu)}$: μ-th vector in sample space; T: BFS in-tree rooted at leader l

1: **Local Step at** $v \in V$**:** Let $\mathcal{P}$ be the set of paths in P_i with v as the leaf node. **For each** $1 \leq \mu \leq n$, set $\nu^{(\mu)}_{P_i,v} = \sum_{p\in\mathcal{P}} \vee_{z\in p} X^{(\mu)}_z$
2: **In round** $r > 0$ **(for all nodes** $v \in V - \{t\}$**):**
3: **send: if** $r = n - h(v) + \mu - 1$ **then** send $\langle \nu^{(\mu)}_{P_i,v} \rangle$ to $parent(v)$ in T
4: **receive [lines 5-9]:**
5: **if** $r = n - h(v) + \mu - 2$ **then**
6: let $\mathcal{I}$ be the set of incoming messages to v
7: **for each** $M \in \mathcal{I}$ **do**
8: let the sender be w and let $M = \langle \nu^{(\mu)}_{P_i,w} \rangle$ and
9: **if** w is a child of v in T **then** $\nu^{(\mu)}_{P_i,v} \leftarrow \nu^{(\mu)}_{P_i,v} + \nu^{(\mu)}_{P_i,w}$
10: **Local Step at leader** l**:** Compute the total sum $\nu^{(\mu)}_{P_i}$ for each sample point μ, by summing up the received $\nu^{(\mu)}_{P_i,w}$ values from all its children w.

Table 7.3: List of Notations Used in the Analysis of the Deterministic Algorithm

A	set constructed in Step 12 of Randomized Blocker Set Algorithm (Alg. 6)
X_v	1 if v is present in A, otherwise 0
X	vector composed of X_v's
$X^{(\mu)}$	μ-th vector in the enumeration of X in the sample space
S	set of sources in $\mathcal{C}$
h	number of hops in a path
n	number of nodes
V_i	set of nodes v with $score(v) \geq (1+\epsilon)^{i-1}$
P_i	set of paths in $\mathcal{C}$ with at least one node in V_i
P_{ij}	set of paths in P_i with at least $(1+\epsilon)^{j-1}$ nodes in V_i
P_i^u	set of paths in P_i with leaf node u
P_{ij}^u	set of paths in P_{ij} with leaf node u
$\sigma_{P_i,u}$	$\sum_{p \in P_i^u} \vee_{v \in V_i \cap p} X_v$
$\sigma_{P_{ij},u}$	$\sum_{p \in P_{ij}^u} \vee_{v \in V_i \cap p} X_v$
$\nu_{P_i,u}$	sum total of $\sigma_{P_i,w}$ values for all descendant nodes w of v
$\nu_{P_{ij},u}$	sum total of $\sigma_{P_{ij},w}$ values for all descendant nodes w of v
$\nu_{P_i}^{(\mu)}$	value of ν_{P_i} with $X^{(\mu)}$ as the input
$\nu_{P_{ij}}^{(\mu)}$	value of $\nu_{P_{ij}}$ with $X^{(\mu)}$ as the input
$\nu_{P_i,u}^{(\mu)}$	value of $\nu_{P_i,u}$ with $X^{(\mu)}$ as the input
$\nu_{P_{ij},u}^{(\mu)}$	value of $\nu_{P_{ij},u}$ with $X^{(\mu)}$ as the input
$\mathcal{C}$	h-hop CSSSP collection
T_x	h-hop shortest path tree rooted at x in collection $\mathcal{C}$
Q	blocker set (being constructed)
l	leader node

Lemma 7.3.14. COMPUTE-ν_{P_i} *(Algorithm 8) correctly computes the* $\nu_{P_i}^{(\mu)}$ *values at leader node* l *for all* μ *in* $O(n)$ *rounds.*

Proof. In Step 1, every node v correctly initialize their contribution to the overall $\nu_{P_i,v}$ term for each μ locally. Since the height of tree T is at most $n-1$, it is readily seen that a node v that is at depth $h(v)$ in T will receive the $count_{P_i}^{(\mu)}$ values from its children in round $n-h(v)+\mu-2$ (Steps 5-9) and thus will have the correct $\nu_{P_i}^{(\mu)}$ value to send in round $n-h(v)+\mu-1$ in Step 3. Since $\mu = O(n)$, Steps 3-9 runs in $O(n)$ rounds. Step 10 is a local step and thus does not involve any communication. This establishes the lemma. □

7.3.3.2 Computing $\nu_{P_{ij}}$

Here we describe our algorithm for computing $\nu_{P_{ij}}^{(\mu)}$ terms for each sample point μ, at leader node l (Algorithm 9). Every node v first initializes its $\nu_{P_{ij}}^{(\mu)}$ value in Step 1. Recall that we assume that all X values are enumerated in order and every node knows this enumeration. In round $n-1-h+\mu$, the node u at height h sends its corresponding $\nu_{P_{ij},u}$ value for $X^{(\mu)}$ (Step 3) along with the total value of $\nu_{P_{ij}}$ it received from its children for $X^{(\mu)}$ (Steps 5-9). Leader node l then computes the total sum $\nu_{P_i}^{(\mu)}$ for each sample point μ, by summing up the received $\nu_{P_i,w}^{(\mu)}$ values from all its children w in Step 10. In Lemma 7.3.15 we show that leader l correctly computes $\nu_{P_{ij}}$ values for all $X^{(\mu)}$'s in $O(n)$ rounds.

Lemma 7.3.15. COMPUTE-$\nu_{P_{ij}}$ *(Algorithm 9) correctly computes the* $\nu_{P_{ij}}^{(\mu)}$ *values at leader node* l *for all* μ *in* $O(n)$ *rounds.*

Proof. In Step 1, every node v correctly initialize their contribution to the overall $\nu_{P_{ij}}$ term for each μ locally. Since the height of tree T is at most $n-1$, it is readily seen that a node v that is at depth $h(v)$ in T will receive the $\nu_{P_{ij}}^{(\mu)}$ values from its children in round $n-h(v)+\mu-2$ (Steps 5-9) and thus will have the correct $\nu_{P_{ij}}^{(\mu)}$

Algorithm 9 COMPUTE-$\nu_{P_{ij}}$: Compute sum of $\nu_{P_{ij}}$ values at leader node l

Input: h: number of hops; S: set of sources; $\mathcal{C}$: h-hop CSSSP collection; $X^{(\mu)}$: μ-th vector in sample space; T: BFS in-tree rooted at leader l

1: **Local Step at** $v \in V$**:** Let $\mathcal{P}$ be the set of paths in P_{ij} with v as the leaf node. **For each** $1 \leq \mu \leq n$, set $\nu^{(\mu)}_{P_{ij},v} = \sum_{p \in \mathcal{P}} \vee_{z \in p} X^{(\mu)}_z$
2: **In round** $r > 0$**:**
3: **send: if** $r = n - h(v) + \mu - 1$ **then** send $\langle \nu^{(\mu)}_{P_{ij},v} \rangle$ to $parent(v)$ in T
4: **receive [lines 5-9]:**
5: **if** $r = n - h(v) + \mu - 2$ **then**
6: let $\mathcal{I}$ be the set of incoming messages to v
7: **for each** $M \in \mathcal{I}$ **do**
8: let the sender be w and let $M = \langle \nu^{(\mu)}_{P_{ij},w} \rangle$ and
9: **if** w is a child of v in T **then** $\nu^{(\mu)}_{P_{ij},v} \leftarrow \nu^{(\mu)}_{P_{ij},v} + \nu^{(\mu)}_{P_{ij},w}$
10: **Local Step at leader** l**:** Compute the total sum $\nu^{(\mu)}_{P_{ij}}$ for each sample point μ, by summing up the received $\nu^{(\mu)}_{P_{ij},w}$ values from all its children w.

value to send in round $n - h(v) + \mu - 1$ in Step 3. Since $\mu = O(n)$, Steps 3-9 runs for at most $2n$ rounds. Step 10 is a local step and thus does not involve any communication. This establishes the lemma. □

7.4 A $\tilde{O}(n^{4/3})$ Rounds Algorithm for Step 6 of Algorithm 5

In Step 6 of Algorithm 5, the goal is to send the distance values $\delta(x, c)$ (which are already computed at node x) from source node x to the corresponding blocker node c. Since there are n sources and $|Q| = \tilde{O}(n^{2/3})$ blocker nodes, this step can be implemented in $\tilde{O}(n^{5/3})$ rounds using all-to-all broadcast (Lemma 5.2.5). One could conjecture that the techniques in [48, 64] could be used to send these $\tilde{O}(n^{5/3})$ messages from the source nodes to the blocker nodes by constructing trees rooted at each c. However, it is not clear how these methods can distribute the $\tilde{O}(n^{5/3})$ different source-destination messages in $o(n^{5/3})$ rounds.

We now describe a method to implement this step more efficiently in $\tilde{O}(n^{4/3})$

rounds deterministically. A randomized $\tilde{O}(n^{4/3})$-round algorithm for this problem is given in Huang et al. [50]. Our algorithm uses the concept of *bottleneck nodes* from that result but is otherwise quite different.

Our algorithm is divided into two cases: (i) when $hops(x, c) > n^{2/3}$ and, (ii) when $hops(x, c) \leq n^{2/3}$ ($hops(x, c)$ denotes the number of edges on the shortest path from x to c).

Case (i) $hops(x, c) > n^{2/3}$: Algorithm 10 describes our algorithm for this case. We first construct an $n^{2/3}$-hop in-CSSSP collection (i.e., CSSSP in-trees) using the blocker set Q as the source set (Step 1, Alg. 10). In Step 2 (Alg. 10) we construct a blocker set Q' of size $\tilde{O}(n^{1/3})$ for this CSSSP collection using deterministic Algorithm 2′ in Sections 6.4.1 and 7.3.2. Then for each $c' \in Q'$ we construct the incoming and outgoing shortest path tree rooted at c' (Step 3, Alg. 10). In Step 4 (Alg. 10), every source $x \in V$ broadcasts the distance value $\delta(x, c')$ for each $c' \in Q'$. The lemma below shows that each $c \in Q$ can determine the $\delta(x, c)$ values for all x for which $hops(x, c) > n^{2/3}$, and the algorithm runs in $\tilde{O}(n^{4/3})$ rounds.

Algorithm 10 Compute $\delta(x, c)$ at c: **when $hops(x, c) > n^{2/3}$**

Input: Q: blocker set

1: Compute $n^{2/3}$-hop in-CSSSP for source set Q using the algorithm in [8].

2: Compute a blocker set Q' of size $\tilde{O}(n/n^{2/3}) = \tilde{O}(n^{1/3})$ for the $n^{2/3}$-hop CSSSP computed in Step 1 using the blocker set algorithm described in Section 6.4.1.

3: **For each $c' \in Q'$ in sequence:** Compute in-SSSP and out-SSSP rooted at c' using Bellman-Ford algorithm.

4: **For each $x \in V$ in sequence:** Broadcast $ID(x)$ and the shortest path distance values $\delta(x, c')$ for each $c' \in Q'$.

5: **Local Step at node $c \in Q$:** For each $x \in V$ compute the shortest path distance value $\delta(x, c)$ using the $\delta(x, c')$ distance values received in Step 4 and the $\delta(c', c)$ distance values computed in Step 3.

Lemma 7.4.1. *Let V' be the set of nodes x such that there is a shortest path from x to a blocker node $c \in Q$ with hop-length greater than $n^{2/3}$. Using Algorithm 10 each blocker node c can correctly compute $\delta(x, c)$ for all such $x \in V'$ in $\tilde{O}(n^{4/3})$ rounds.*

Proof. Since $hops(x, c) > n^{2/3}$, there exists a blocker node $c' \in Q'$ (constructed in Step 2) such that the shortest path from x to c passes through c'. Thus c can compute the distance value $\delta(x, c)$ by adding $\delta(x, c')$ (received in Step 4) and $\delta(c', c)$ (computed in Step 3) values in Step 5.

Step 1 takes $O(n^{2/3} \cdot |Q|) = \tilde{O}(n^{4/3})$ rounds using Bellman-Ford algorithm. Step 2 requires $\tilde{O}(n^{2/3} \cdot n^{2/3}) = \tilde{O}(n^{4/3})$ rounds by Corollary 7.3.13. Since $|Q'| = \tilde{O}(n/n^{2/3}) = \tilde{O}(n^{1/3})$, Step 3 takes $\tilde{O}(n \cdot n^{1/3}) = \tilde{O}(n^{4/3})$ rounds using Bellman-Ford algorithm and so does Step 4 using Lemma 5.2.5. Step 5 is a local step and has no communication. □

Case (ii) $hops(x, c) \leq n^{2/3}$: This case deals with sending the distance values from source nodes x to the blocker nodes c when the shortest path between x and c has hop-length at most $n^{2/3}$. Recall that using an all-to-all broadcast or the techniques in [48, 64] for sending these $\tilde{O}(n^{5/3})$ messages appears to require at least $\tilde{O}(n^{5/3})$ rounds.

Let $\mathcal{C}^Q$ be the $n^{2/3}$-hop in-CSSSP collection for source set Q. A set $B \subset V$ is a set of bottleneck nodes if removing the nodes in B, along with their descendants in the trees in the collection $\mathcal{C}^Q$, reduces the congestion to at most $\tilde{O}(n^{4/3})$, i.e. every node would need to send at most $\tilde{O}(n^{4/3})$ messages if all nodes x transmitted their $\delta(x, c)$ values along the pruned CSSSP trees in the collection $\mathcal{C}^Q$. This notion is defined in Huang et al. [50], where they present a randomized algorithm using the randomized scheduling algorithm in Ghaffari [39] to identify such a set of bottleneck nodes. Here we deterministically identify a set of bottleneck nodes B where $|B| = \tilde{O}(n^{1/3})$ (Step 1, Alg. 11) using a pipelined strategy (Alg. 17 in Appendix 7.5.3.1). Clearly, after we remove these bottleneck nodes, any remaining node needs to send

at most $\tilde{O}(n^{4/3})$ messages.

After we identify the set of bottleneck nodes B we run Bellman-Ford algorithm [15] for each $b \in B$ to compute both the incoming and outgoing shortest path tree rooted at b (Step 2, Alg. 11). We then broadcast the $\delta(x, b)$ distance values from every source $x \in V$ to the corresponding $b \in B$ (Step 3, Alg. 11). Thus if x lies in the subtree rooted at b for a blocker node c, then c can compute $\delta(x, c)$ value by adding $\delta(x, b)$ and $\delta(b, c)$ distance values (Step 4, Alg. 11).

It remains to send the distance value $\delta(x, c)$ to blocker node c if x is not part of a subtree of any bottleneck node b in c's shortest path tree. Since the maximum congestion at any node is at most $\tilde{O}(n^{4/3})$ after removing bottleneck nodes in B, we are able to perform this computation deterministically. In Steps 8-9 (Alg. 11), we use a simple round-robin strategy to propagate these distance values from each source x to all blocker nodes c in the network. We show in Section 7.4.1, using the notion of *frames*, that this simple strategy achieves the desired $\tilde{O}(n^{4/3})$-round bound.

Algorithm 11 Compute $\delta(x, c)$ at c: **when** $\boldsymbol{hops(x, c) \leq n^{2/3}}$

Input: Q: blocker set; $|Q| \leq n^{2/3} \log n$; $\mathcal{C}^Q$: $n^{2/3}$-hop in-CSSSP collection for set Q

1: Compute a set of bottleneck nodes B of size $\tilde{O}(n^{1/3})$ using Algorithm 17 (Section 7.5.3.1).

2: **For each $b \in B$ in sequence:** Compute both in-SSSP and out-SSSP tree rooted at b using Bellman-Ford algorithm.

3: **For each $x \in V$ in sequence:** Broadcast $ID(x)$ and the shortest path distance values $\delta(x, b)$ for each $b \in B$.

4: **Local Step at node $c \in Q$:** For each $x \in V$ compute $\delta^{(B)}(x, c) = \min_{b \in B}\{\delta(x, b) + \delta(b, c)\}$ using the $\delta(x, b)$ distance values received in Step 3 and $\delta(b, c)$ distance values computed in Step 2.

5: Remove subtrees rooted at $b \in B$ from the collection $\mathcal{C}^Q$ using Algorithm 16 (Section 7.5.1.4).

6: Reset round counter to 0.

7: Assume the nodes in Q are ordered in a (cyclic) sequence O.

8: **Round** $0 < r \leq (n^{4/3} \log n + n^{4/3}) \cdot ((1/3) \cdot \log n / \log \log n - 1)$**:**

9: **Round-robin sends:** At each node v, forward an unsent message for the next blocker node c in O to its parent in c's tree.

Lemma 7.4.2. *If the shortest path from $x \in V$ to a blocker node $c \in Q$ has hop-length at most $n^{2/3}$ and there exists a bottleneck node $b \in B$ on this path, then after executing Steps 1-4 of Algorithm 11 blocker node c knows the distance value $\delta(x, c)$ for all such $x \in V$.*

Proof. This is immediate from Step 4 (Alg. 11) where c will compute $\delta(x, c)$ by adding the distance values $\delta(x, b)$ (received in Step 3, Alg. 11) and $\delta(b, c)$ value (computed at c in Step 2, Alg. 11). □

Lemma 7.4.3. *If a source node x lies in a blocker node c's tree in the CSSSP*

collection $\mathcal{C}^Q$ after the execution of Step 5 of Algorithm 11, then c would have received $\delta(x,c)$ value by $(n^{4/3}\log n + n^{4/3}) \cdot ((1/3) \cdot \log n/\log\log n - 1)$ rounds of Step 9 of Algorithm 11.

Lemma 7.4.3 is established below in Section 7.4.1. Lemmas 7.4.2 and 7.4.3 establish the following lemma.

Lemma 7.4.4. *If the shortest path from $x \in V$ to a blocker node $c \in Q$ has hop-length at most $n^{2/3}$, then after running Algorithm 11 blocker node c knows the distance value $\delta(x,c)$ for all such $x \in V$.*

Lemma 7.4.5. *Algorithm 11 runs for $\tilde{O}(n^{4/3})$ rounds in total.*

Proof. Step 1 takes $\tilde{O}(n^{4/3})$ rounds by Lemma 7.5.10. Since $|B| = \tilde{O}(n^{1/3})$, Step 2 takes $\tilde{O}(n \cdot n^{1/3}) = \tilde{O}(n^{4/3})$ rounds using Bellman-Ford algorithm and so does Step 3 using Lemma 5.2.5. Step 4 is a local step and involves no communication. Step 5 takes $\tilde{O}(n^{2/3} \cdot |Q|) = \tilde{O}(n^{4/3})$ rounds using Lemma 7.5.6. Step 9 runs for $\tilde{O}(n^{4/3})$ rounds, thus establishing the lemma. □

7.4.1 Correctness of Step 9 of Algorithm 11

In this section we will establish that the simple round-robin approach used in Steps 8-9 of Algorithm 11 is sufficient to propagate distance values $\delta(x,c)$ from source nodes $x \in V$ to blocker nodes $c \in Q$ in $\tilde{O}(n^{4/3})$ rounds, when the congestion at any node is at most $\tilde{O}(n^{4/3})$. While this looks plausible, the issue to resolve is whether a node could be left idling when there are more messages it needs to pass on from its descendants to its parents in some of the trees. This could happen because each node forwards at most one message per round and these descendants might have forwarded messages for other blocker nodes. The round robin scheme appears to only guarantee that a message for a chosen blocker node will be sent from a node to its parent at least once every $|Q|$ rounds.

We now present and analyze a more structured version of Steps 9-10 to establish the bound. In this Algorithm 6 we divide Step 9 (Alg. 11) into $(1/3) \cdot (\log n / \log\log n) - 1$ different *stages*, with each stage running for at most $n^{4/3}\log^{3/2} n + n^{4/3}$ rounds (we assume $|Q| \leq n^{2/3}\log n$). Our key observation (in Lemma 7.4.8) is that at the start of Stage i, every node v only needs to send the distance values for at most $n^{2/3}/\log^{i-1/2} n$ different blocker nodes (note that i is not a constant), thus more messages can be sent by v to each blocker node in later stages.

Let $Q_{v,i}$ be the set of blocker nodes for which node v has messages to send at start of stage i. We introduce the notion of a *frame*, where each frame has a single round available for each blocker node in $Q_{v,i}$. Stage i is divided into $n^{2/3}\log^{i+1} n + n^{2/3}$ frames (we will show that each frame consists of $\lceil n^{2/3}/\log^{i-1/2} n \rceil$ rounds). In each frame, node v sends out an unsent message for each $c \in Q_{v,i}$ to its parent in c's tree (Step 4, Alg. 12).

Algorithm 12 Algorithm for Stage i at node $v \in V$

Input: O: (cyclic) sequence of nodes in blocker set Q; $\mathcal{C}^Q$: $n^{2/3}$-hop CSSSP collection for set Q

1: **for** $\boldsymbol{i \geq 0}$: **do**
2: Let $Q_{v,i}$ be the set of nodes in Q for which v contains at least one unsent message during Stage-i.
3: **for** frame $j = 1$ to $\lceil n^{2/3}\log^{i+1} n + n^{2/3} \rceil$ **do**
4: **For each** $c \in Q_{v,i}$ **in sequence:** v forwards an unsent message for c to its parent in c's tree.

Lemma 7.4.6. *For all blocker nodes $c \in Q_{v,i}$, node v would have sent α messages to its parent in c's tree by $\alpha + n^{2/3} - h_c(v)$ frames of Stage i, where $h_c(v) = hops(v, c)$, provided at least α messages are routed through v in Step 4 of Algorithm 12.*

Proof. Fix a blocker node c. Let i' be the smallest i for which the above statement

does not hold and let v be a node with maximum $h_c(v)$ value for which this statement is violated in Stage i'. Node v is not a leaf node since α is 0 or 1 for a leaf and a leaf would have sent its distance value to its parent in the first frame of Stage-0.

So v must be an internal node. Since the statement does not hold for v for the first time for α, it implies that v has already sent $\alpha - 1$ messages (including its own distance value $\delta(v, c)$) by $(\alpha - 1) + n^{2/3} - h_c(v)$ frames and now does not have any message to send to its parent in c's tree in the next frame. However since the statement holds for all of v's children, v should have received at least $\alpha - 1$ messages from its children by $(\alpha - 1) + n^{2/3} - (h_c(v) + 1)$-th frame, resulting in a contradiction. □

Since $h_c(v) \leq n^{2/3}$, Lemma 7.4.6 leads to the following Corollary.

Corollary 7.4.7. *After the completion of Stage i, every node v would have sent all or at least $n^{2/3} \log^{i+1} n$ different distance values for all blocker nodes $c \in Q_{v,i}$.*

Lemma 7.4.8. *The set $Q_{v,i}$ has size at most $\lceil n^{2/3} / \log^{i-1/2} n \rceil$.*

Proof. By Corollary 7.4.7 after the completion of Stage $i - 1$, every node v would have sent all or at least $n^{2/3} \log^{i} n$ different distance values for all blocker nodes in $Q_{v,i-1}$. Thus the set $Q_{v,i}$ will consist of only those nodes from Q for which v needs to send at least $n^{2/3} \log^{i} n$ different distance values. Since congestion at any node v is at most $n\sqrt{|Q|} = n^{4/3} \log^{1/2} n$ (using Lemma 7.5.8), the size of $Q_{v,i}$ is at most $n^{4/3} \log^{1/2} n / n^{2/3} \log^{i} n = n^{2/3} / \log^{i-1/2} n$. This establishes the lemma. □

Proof of Lemma 7.4.3. Since $|Q_{v,i}| \leq n^{2/3} / \log^{i-1/2} n$ (by Lemma 7.4.8), Stage i runs for $n^{2/3} / \log^{i-1/2} n \cdot (n^{2/3} \log^{i+1} n + n^{2/3}) \leq n^{4/3} \log^{3/2} n + n^{4/3}$ rounds. Lemma 7.4.3 is immediately established from Corollary 7.4.7 and the fact that there are $(1/3) \cdot \log n / \log \log n - 1$ stages. □

7.5 Helper Algorithms

7.5.1 Helper Algorithms for Randomized Blocker Set Algorithm

7.5.1.1 Algorithm for Computing V_i and P_i

Here we describe our algorithm for computing Steps 3 and 4 of Algorithm 6, which computes the set V_i and identifies which paths belong to P_i respectively. Since every node with score value greater than or equal to $(1+\epsilon)^{i-1}$ belongs to V_i, computing V_i is quite trivial. And to determine if a path p belong to P_i, we only need to check if one of the nodes in p is in V_i.

Our algorithm for computing V_i works as follows: Every node v checks if its score value is greater than or equal to $(1+\epsilon)^{i-1}$ and if so, it broadcast its ID to every other node. The set V_i is then constructed by including the IDs of all such nodes. Since there are at most n messages involved in the broadcast step, this algorithm takes $O(n)$ rounds. This leads to the following lemma.

Lemma 7.5.1. *Given the score(v) values for every $v \in V$, the set V_i can be constructed in $O(n)$ rounds.*

We now describe our algorithm for computing P_i. Fix a source node $x \in V$. In Round 0 x initializes $flag$ to *true* if it belongs to V_i, otherwise set it to $false$ (Step 1). It then sends this $flag$ to its children in next round (Step 3). In round $r \geq 1$, a node v that is r hops away from x receives the $flag$ from its parent (Steps 5-8) and v updates the $flag$ value in Step 8 (set it to true if $v \in V_i$) and send it to its children in x's tree in round $r+1$ (Step 3).

Lemma 7.5.2. *Using* Compute-P_i *(Algorithm 13), P_i can be computed in $O(h)$ rounds per source node.*

Proof. Fix a path p from source x to leaf node v. After h rounds, v will know if any node in p belongs to V_i (using the $flag$ value it received in Steps 5-8).

Algorithm 13 COMPUTE-P_i: Algorithm for computing paths in P_i for source x at node v

Input: V_i; h: number of hops; T_x: tree for source x

1: **(Round 0): if** $v \in V_i$ **then** set $flag \leftarrow true$ **else** $flag \leftarrow false$
2: **Round** $h \geq r > 0$**:**
3: **Send: if** $r = h_x(v) + 1$ **then** send $\langle flag \rangle$ to all children
4: **receive [lines 5-8]:**
5: **if** $r = h_x(v)$ **then**
6: let M be the incoming message to v
7: let the sender be w and let $M = \langle flag_w \rangle$ and
8: **if** w is a parent of v in T_x **then** $flag \leftarrow flag \vee flag_w$
9: **Local Step at** v**: if** v is a leaf node **and** $flag = true$ **then** the path from x to v is in P_i.

The algorithm takes h rounds per source x and thus P_i can be computed in $O(|S| \cdot h)$ rounds in total (since we need to run the algorithm for every source x). □

7.5.1.2 Algorithm for Computing P_{ij}

Here we describe our algorithm for computing Step 7(a) of Algorithm 6, which identifies the paths in P_i that also belong to P_{ij}. Since every path in P_{ij} has at least $(1+\epsilon)^{j-1}$ nodes from V_i, for each path p we need to determine the number of nodes in p that belong to V_i. We do this by counting the number of nodes that are in V_i, starting from root to leaf node.

Our algorithm works as follows: Fix a source node $x \in V$. In Round 0 x initializes β value to 1 if it belongs to V_i, otherwise set it to 0 (Step 1). It then sends this β value to its children in next round (Step 3). In round $r \geq 1$, a node v that is r hops away from x receives the β value from its parent (Steps 5-8) and v updates the β value in Step 8 (increment it by 1 if $v \in V_i$) and send it to its children in x's tree in round $r + 1$ (Step 3).

Lemma 7.5.3. *Using* COMPUTE-P_{ij} *(Algorithm 14),* P_{ij} *can be computed in* $O(h)$ *rounds per source node.*

Algorithm 14 COMPUTE-P_{ij}: Algorithm for computing paths in P_{ij} for source x at node v

Input: V_i; h: number of hops; T_x: tree for source x

1: **(Round 0): if** $v \in V_i$ set $\beta \leftarrow 1$ **else** $\beta \leftarrow 0$
2: **Round** $h \geq r > 0$**:**
3: **Send: if** $r = h_x(v) + 1$ **then** send $\langle \beta \rangle$ to all children
4: **receive [lines 5-8]:**
5: **if** $r = h_x(v)$ **then**
6: let $\mathcal{M}$ be the incoming message to v
7: let the sender be w and let $M = \langle \beta_w \rangle$ and
8: **if** w is a parent of v in T_x **then** $\beta \leftarrow \beta + \beta_w$
9: **Local Step at** v**: if** v is a leaf node **and** $\beta \geq (1+\epsilon)^{j-1}$ **then** the path from x to v is in P_{ij}.

Proof. Fix a path p from source x to leaf node v. After h rounds, v will know the number of nodes that belong to V_i (using the β values it received in Steps 5-8).

The algorithm takes h rounds per source x and thus P_{ij} can be computed in $O(|S| \cdot h)$ rounds in total (since we need to run the algorithm for every source x). □

7.5.1.3 Algorithm for Computing $|P_{ij}|$

Algorithm 15 describes our algorithm for computing Step 7(b) of Algorithm 6, which computes the value of $|P_{ij}|$. Let P_{ij}^v represents the set of paths p in P_{ij} with v as the leaf node. Every node v knows the set P_{ij}^v after running the algorithm described in the previous section. Our algorithm works as follows: Every node v first compute $|P_{ij}^v|$ (Step 1) and then broadcast this value in Step 2. Every node v then compute $|P_{ij}|$ by summing up the values received in Step 2 (Step 3).

Algorithm 15 COMPUTE-$|P_{ij}|$

Input: P_{ij}^v: paths in P_{ij} with v as the leaf node

1: **Local Step at** $v \in V$**:** set $\alpha_{P_{ij}^v} \leftarrow |P_{ij}^v|$
2: **For each** $v \in V$**:** Broadcast $ID(v)$ and the value $\alpha_{P_{ij}^v}$.
3: **Local Step at** $v \in V$**:** $|P_{ij}| \leftarrow \sum_{v' \in V} \alpha_{P_{ij}^{v'}}$

Lemma 7.5.4. COMPUTE-$|P_{ij}|$ *(Algorithm 15) computes* $|P_{ij}|$ *in* $O(n)$ *rounds.*

Proof. Steps 1 and 3 are local steps and involves no communication. Step 2 involves a broadcast of n messages and takes $O(n)$ rounds using Lemma 5.2.5. □

7.5.1.4 Remove Subtrees rooted at $z \in Z$

In this Section we describe a deterministic algorithm for implementing Step 15 of Algorithm 6, which removes subtrees rooted at nodes $z \in Z$ from the trees in the given h-hop CSSSP collection $\mathcal{C}$. This algorithm (Algorithm 16) is quite simple and works as follows: Fix a source x and let its corresponding tree in $\mathcal{C}$ be T_x. Every node $z \in Z$ in T_x send its ID to all its children in T_x (Step 1). Every node v on receiving a message from its parent in T_x, forwards it to all its children and set the parent pointer in T_x to NIL (Step 2).

Algorithm 16 REMOVE-SUBTREES: Algorithm for Removing Subtrees rooted at $z \in Z$ for source x at node v

Input: S: set of sources; $\mathcal{C}$: h-hop CSSSP collection for set S

1: **(Round 0:) If** $v \in Z$ **then** send $\langle ID(v) \rangle$ to all children in T_x and set $parent_x(v)$ to NIL.
2: **(Round** $r > 0$**:) If** v received a message M in round $r-1$ **then** set $parent_x(v)$ to NIL and send M to all children in T_x.

Lemma 7.5.5. *Given a source* $x \in S$ *and tree* $T_x \in \mathcal{C}$*, then* REMOVE-SUBTREES *(Algorithm 16) removes all subtrees rooted at* $z \in Z$ *in* T_x*.*

Proof. Every $z \in Z$ in T_x removes its parent pointer in T_x in Step 1. Any node $v \in V$ that lies in the subtree rooted at a $z \in Z$ in T_x would have received a message with $ID(z)$ from its parent by h rounds (since height of T_x is at most h) and hence would have set its parent pointer to NIL in Step 2. □

Lemma 7.5.6. REMOVE-SUBTREES *(Algorithm 16) requires at most* h *rounds per source node* $x \in S$*.*

Proof. Since the height of T_x is at most h, any node $v \in V$ which lies in the subtree rooted at a $z \in Z$ will receive the message from z by h rounds. This establishes the lemma. □

7.5.2 h-hop Shortest Path Extension Algorithm [50]

We now describe an algorithm for computing Step 7 of Algorithm 5, which computes h-hop *extensions* based on the Bellman-Ford algorithm [15]. This algorithm is also used as a step in the randomized APSP algorithm of Huang et al. [50]. Here every blocker node $c \in Q$ knows its shortest path distance value from every source node $x \in V$ and the goal is to extend the shortest path from x to c by additional h hops.

This algorithm works as follows: Fix a source $x \in V$. Every blocker node $c \in Q$ initializes the shortest path distance from x to $\delta(x, c)$ (this value is already known to every c). We then run Bellman-Ford algorithm at every node $v \in V$ for source x for h rounds using these initialized values. We repeat this for every $x \in V$.

After this algorithm terminates, every sink node $t \in V$ knows the shortest path distance from every $x \in V$. Since we run Bellman-Ford for h rounds per source node, this whole algorithm takes $O(nh)$ rounds in total. This leads to the following lemma.

Lemma 7.5.7. *The h-hop shortest path extensions can be computed in $O(nh)$ rounds for every source $x \in V$ using Bellman-Ford algorithm.*

7.5.3 Helper Algorithms for Algorithm 11

7.5.3.1 Computing Bottleneck Nodes

Here we describe our deterministic algorithm for computing Step 1 of Algorithm 11, which identifies a set B of bottleneck nodes such that removing this set of nodes reduces the congestion in the network from $O(n \cdot |Q|)$ to $O(n \cdot \sqrt{|Q|})$. However when randomization is allowed, there is a $O(n \cdot \sqrt{|Q|})$ randomized algorithm of Huang et

al [50] that computes this set w.h.p. in n. Our deterministic algorithm is however very different from the randomized algorithm given in [50] and it uses ideas from our blocker set algorithm in [10].

We now give an overview of the randomized algorithm of [50] that computes this set of bottleneck nodes. For a source x and its incoming shortest path tree T_x, every node in T_x calculates the number of outgoing messages for source x. This is done by waiting for messages from all children nodes, followed by sending a message to its parent in T_x. This takes $O(n)$ rounds and can be run across multiple nodes in Q as congestion is at most $O(|Q|)$. Thus using the randomized algorithm of Ghaffari [39], this algorithm can be run across all nodes in Q concurrently in $\tilde{O}(n + |Q|) = \tilde{O}(n)$ rounds. After computing these values, a node b with maximum count is selected to the set B and is then removed from the network. The algorithm repeats this for $O(\sqrt{|Q|})$ times, thus eliminating all nodes that needed to send at least $n\sqrt{|Q|}$ messages (since removal of every such node eliminates $O(n\sqrt{|Q|})$ nodes across all trees and there are at most $n \cdot |Q|$ nodes).

Our deterministic algorithm for computing bottleneck nodes (Algorithm 17) works as follows: In Step 1, the algorithm computes the $count_{v,c}$ values (number of messages v needs to send to its parent in c's tree) using Algorithm 18 described in Section 7.5.3.2. Every node v calculates the total number of messages it needs to send by summing up the values computed in Step 1 (Step 2) and then broadcast this value in Step 4. The node with maximum value is added to the bottleneck node set B (Step 5) and the values of its ancestors and descendants are updated using the algorithms in [8]. In Lemma 7.5.10 we establish that the whole algorithm runs in $O(n\sqrt{|Q|} + h \cdot |Q|)$ rounds deterministically.

Lemma 7.5.8. *After* Compute-Bottleneck *(Algorithm 17) terminates,* $total\_count_v \leq n\sqrt{|Q|}$ *for all nodes* v.

Proof. This is immediate since the while loop in Steps 3-6 terminates only when

Algorithm 17 COMPUTE-BOTTLENECK: Compute Bottleneck Nodes Set B

Input: Q: blocker set; $\mathcal{C}^Q$: CSSSP collection for blocker set Q
Output: B: set of bottleneck nodes

1: **For each** $c \in Q$ **in sequence:** Compute $count_{v,c}$ values at every node $v \in V$ using Algorithm 18 (Section 7.5.3.2).
2: **Local Step at** $v \in V$**:** Compute $total\_count_v \leftarrow \sum_{c \in Q} count_{v,c}$
3: **while there is a node** v **with** $\boldsymbol{total\_count_v > n\sqrt{|Q|}}$ **do**
4: **For each** $v \in V$**:** Broadcast $ID(v)$ and $total\_count_v$ value .
5: Add node b to B such that b has maximum $total\_count_v$ value (break ties using IDs).
6: Update $total\_count_v$ values for the descendants and ancestors of b across all trees in the collection $\mathcal{C}^Q$ using Algorithm 6 in Chapter 5 and Algorithm 4 in Chapter 6.

there is no node v with $total\_count_v > n\sqrt{|Q|}$. □

Lemma 7.5.9. *The set of bottleneck nodes, B, constructed by* COMPUTE-BOTTLENECK *(Algorithm 17) has size at most $\sqrt{|Q|}$.*

Proof. Since every node b added to set B has $total\_count_b > n\sqrt{|Q|}$, removing such b is going to remove at least $n\sqrt{|Q|}$ nodes across all trees in Q in Step 6. And since there are at most $n \cdot |Q|$ nodes across all trees, set B has size at most $\sqrt{|Q|}$. □

Lemma 7.5.10. COMPUTE-BOTTLENECK *(Algorithm 17) runs for $O(n\sqrt{|Q|} + h \cdot |Q|)$ rounds.*

Proof. Step 1 takes $O(h \cdot |Q|)$ rounds using Lemma 7.5.11. Step 2 is a local computation step and involves no communication. Step 4 involves a broadcast of at most n messages and hence takes $O(n)$ rounds using Lemma 5.2.5. Step 5 again do not involve any communication. Both Algorithm 6 and Algorithm 4 takes $O(n)$ rounds (Lemmas 5.3.6 and 6.4.6). Since B has size at most $\sqrt{|Q|}$ (by Lemma 7.5.9), the while loop runs for at most $\sqrt{|Q|}$ iterations, thus establishing the lemma. □

7.5.3.2 Computing $count_{v,c}$ Values

Here we describe our algorithm for computing Step 1 of Algorithm 17, which computes $count_{v,c}$ values in a given h-hop CSSSP collection $\mathcal{C}$ for source set S. Our algorithm (Algorithm 18) is quite simple and works as follows: Fix a source $c \in S$ and let T_c be the tree corresponding to source c in $\mathcal{C}$. The goal is to compute the number of messages each node $v \in T_c$ needs to send to its parent. In Step 1 every node $v \in T_c$ initializes its $count_{v,c}$ value to 1. Every node v that is $h_c(v)$ hops away from c receives the *count* values from all its children by round $h - h_c(v)$ (Steps 5-9) and it then send it to its parent in round $h - h_c(v) + 1$ (Step 3) after updating it (Step 9).

Algorithm 18 COMPUTE-COUNT: Algorithm for computing $count_{v,c}$ values for source c at node v

Input: h: number of hops, T_c: tree for source c

1: **(Round 0): if** $v \in T_c$ **then** set $count_{v,c} \leftarrow 1$ **else** $count_{v,c} \leftarrow 0$
2: **Round** $h + 1 \geq r > 0$**:**
3: **Send: if** $r = h - h_c(v) + 1$ **then** send $\langle count_{v,c} \rangle$ to v's parent
4: **receive [lines 5-9]:**
5: **if** $r = h - h_c(v)$ **then**
6: let $\mathcal{I}$ be the set of incoming message to v
7: **for** $M \in \mathcal{I}$ **do**
8: let the sender be w and let $M = \langle count_{w,c} \rangle$ and
9: **if** w is a child of v in T_c **then** $count_{v,c} \leftarrow count_{v,c} + count_{w,c}$

Lemma 7.5.11. COMPUTE-COUNT *(Algorithm 18) correctly computes* $count_{v,c}$ *for every* $v \in T_c$ *in* $h + 1$ *rounds per source node* c.

Proof. Every leaf node v can initialize their $count_{v,c}$ values to 1 in Step 1. For every other internal node v, v correctly computes $count_{v,c}$ value after receiving the *count* values from all its children by round $h - h_c(v)$ (Steps 5-9) and then send the correct $count_{v,c}$ value to its parent in round $h - h_c(v) + 1$ in Step 3.

Since $h_c(v) \geq 0$, this algorithm requires at most $h + 1$ rounds. □

7.6 Conclusion

We have presented a new deterministic distributed algorithm for computing exact weighted APSP in $\tilde{O}(n^{4/3})$ rounds in both directed and undirected graphs with arbitrary edge weights. This algorithm improves on the $\tilde{O}(n^{3/2})$ round APSP algorithm presented in Chapter 5. At the heart of our algorithm is an efficient distributed algorithm for sending the distance values from source nodes to the blocker nodes and an improved deterministic algorithm for computing the blocker set using pairwise independence and derandomization. We believe that both these techniques may be of independent interest for obtaining results for other distributed graph problems.

The main open question left by our work is whether we can get a deterministic algorithm that can match the current $\tilde{O}(n)$ randomized bound for computing weighted APSP [19].

Bibliography

[1] A. Abboud, K. Censor-Hillel, and S. Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Proc. ISDC*, pages 29–42. Springer, 2016.

[2] A. Abboud, F. Grandoni, and V. V. Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In *Proc. SODA*, pages 1681–1697, 2015.

[3] A. Abboud, V. Vassilevska Williams, and H. Yu. Matching triangles and basing hardness on an extremely popular conjecture. In *Proc. STOC*, pages 41–50. ACM, 2015.

[4] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. FOCS*, pages 434–443. IEEE, 2014.

[5] A. Abboud, V. V. Williams, and J. Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proc. SODA*, pages 377–391. SIAM, 2016.

[6] U. Agarwal and V. Ramachandran. Finding k simple shortest paths and cycles. In *Proc. ISAAC*, pages 8:1–8:12, 2016.

[7] U. Agarwal and V. Ramachandran. Fine-grained complexity for sparse graphs. In *Proc. STOC*, pages 239–252. ACM, 2018.

[8] U. Agarwal and V. Ramachandran. Distributed weighted all pairs shortest paths through pipelining. In *Proc. IPDPS*. IEEE, 2019.

[9] U. Agarwal and V. Ramachandran. Faster deterministic all pairs shortest paths in CONGEST model. *Manuscript*, 2019.

[10] U. Agarwal, V. Ramachandran, V. King, and M. Pontecorvi. A deterministic distributed algorithm for exact weighted all-pairs shortest paths in $\tilde{O}(n^{3/2})$ rounds. In *Proc. PODC*, pages 199–205. ACM, 2018.

[11] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

[12] N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proc. FOCS*, pages 417–426. IEEE, 1992.

[13] N. Alon, R. Yuster, and U. Zwick. Color-coding. *JACM*, 42(4):844–856, 1995.

[14] A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proc. STOC*, pages 51–58. ACM, 2015.

[15] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[16] B. Berger, J. Rompel, and P. W. Shor. Efficient NC algorithms for set cover with applications to learning and geometry. *J. Comp. Sys. Sci.*, 49(3):454–477, 1994.

[17] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proc. STOC*, pages 101–110, 2009.

[18] A. Bernstein and D. Nanongkai. Distributed exact weighted all-pairs shortest paths in near-linear time. In *Proc. STOC*. ACM, 2019.

[19] A. Bernstein and D. Nanongkai. Distributed exact weighted all-pairs shortest paths in near-linear time. In *Proc. STOC*. ACM, 2019.

[20] S. Bernstein. Theory of probability, 1927.

[21] U. Brandes. A faster algorithm for betweenness centrality. *Jour. Math. Soc.*, 25(2):163–177, 2001.

[22] K. Censor-Hillel, S. Khoury, and A. Paz. Quadratic and near-quadratic lower bounds for the congest model. In *Proc. DISC*, 2017.

[23] K. Censor-Hillel, M. Parter, and G. Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *DISC*, 2017.

[24] S. Chechik, T. D. Hansen, G. F. Italiano, V. Loitzenbauer, and N. Parotsidis. Faster algorithms for computing maximal 2-connected subgraphs in sparse directed graphs. In *Proc. SODA*, pages 1900–1918. SIAM, 2017.

[25] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51:968–992, 2004.

[26] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM Jour. Comput.*, 37:1299–1318, 2008.

[27] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[28] M. Elkin. Distributed exact shortest paths in sublinear time. In *Proc. STOC*, pages 757–770. ACM, 2017.

[29] M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Proc. FOCS*, pages 128–137. IEEE, 2016.

[30] D. Eppstein. Finding the k shortest paths. *SIAM Jour. Comput.*, 28:652–673, 1998.

[31] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[32] L. R. Ford Jr. Network flow theory. Technical report, RAND CORP SANTA MONICA CA, 1956.

[33] S. Frischknecht, S. Holzer, and R. Wattenhofer. Networks cannot compute their diameter in sublinear time. In *SODA '12*, pages 1150–1162, 2012.

[34] H. N. Gabow. Scaling algorithms for network problems. *J. Comp. Sys. Sci.*, 31(2):148–168, 1985.

[35] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5(3):165–185, 1995.

[36] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. Syst.*, 5(1):66–77, 1983.

[37] J. Gao, R. Impagliazzo, A. Kolokolova, and R. Williams. Completeness for first-order properties on sparse structures with algorithmic applications. In *Proc. SODA*, pages 2162–2181. SIAM, 2017.

[38] J. A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comp.*, 27(1):302–316, 1998.

[39] M. Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proc. PODC*, pages 3–12. ACM, 2015.

[40] M. Ghaffari and J. Li. Improved distributed algorithms for exact shortest paths. In *Proc. STOC*, pages 431–444. ACM, 2018.

[41] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Inf. Proc. Lett.*, 109(7):352–355, 2009.

[42] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. ICALP*, pages 61–72. Springer, 2000.

[43] M. Henzinger, S. Krinninger, and V. Loitzenbauer. Finding 2-edge and 2-vertex strongly connected components in quadratic time. In *Proc. ICALP*, pages 713–724. Springer, 2015.

[44] M. Henzinger, S. Krinninger, and D. Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proc. STOC*, pages 489–498. ACM, 2016.

[45] M. Henzinger, S. Krinninger, D. Nanongkai, and T. Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. STOC*, pages 21–30. ACM, 2015.

[46] J. Hershberger, S. Suri, and A. Bhosle. On the difficulty of some shortest path problems. *ACM Trans. Alg. (TALG)*, 3(1):5, 2007.

[47] L. Hoang, M. Pontecorvi, R. Dathathri, G. Gill, B. You, K. Pingali, and V. Ramachandran. A round-efficient distributed betweenness centrality algorithm. In *Proc. PPoPP*. ACM, 2019.

[48] L. Hoang, M. Pontecorvi, R. Dathathri, G. Gill, B. You, K. Pingali, and V. Ramachandran. A round-efficient distributed betweenness centrality algorithm. 2019.

[49] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *PODC '12*, pages 355–364, 2012.

[50] C.-C. Huang, D. Na Nongkai, and T. Saranurak. Distributed exact weighted all-pairs shortest paths in $\tilde{O}(n^{5/4})$ rounds. In *Proc. FOCS*, pages 168–179. IEEE, 2017.

[51] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Jour. Comput. Sys. Sci.*, 62(2):367–375, 2001.

[52] A. Itai and M. Rodeh. Finding a minimum circuit in an graph. *SIAM Jour. Comput.*, 7(4):413–423, 1978.

[53] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Jour. Comput.*, 4(1):77–84, 1975.

[54] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *JACM*, 24(1):1–13, 1977.

[55] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.

[56] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[57] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.

[58] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. IEEE FOCS*, pages 81–89. IEEE, 1999.

[59] S. Kutten and D. Peleg. Fast distributed construction of smallk-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998.

[60] H. O. Lancaster. Pairwise statistical independence. *Annals of Mathematical Statistics*, 36(4):1313–1317, 1965.

[61] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[62] E. L. Lawler. Comment on a computing the k shortest paths in a graph. *CACM*, 20(8):603–605, 1977.

[63] C. Lenzen and B. Patt-Shamir. Fast partial distance estimation and applications. In *Proc. PODC*, pages 153–162. ACM, 2015.

[64] C. Lenzen, B. Patt-Shamir, and D. Peleg. Distributed distance computation and routing with small messages. *Dist. Comp.*, 32(2):133–157, 2019.

[65] C. Lenzen and D. Peleg. Efficient distributed source detection with limited bandwidth. In *Proc. PODC*, pages 375–382. ACM, 2013.

[66] A. Lincoln, V. V. Williams, and R. Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proc. SODA*, pages 1236–1252. SIAM, 2018.

[67] A. Lingas and E.-M. Lundell. Efficient approximation algorithms for shortest cycles in undirected graphs. *Inf. Proc. Lett.*, 109(10):493–498, 2009.

[68] M. Luby. Removing randomness in parallel computation without a processor penalty. *J. Comp. Sys. Sci.*, 47(2):250–286, 1993.

[69] E. Minieka. On computing sets of shortest paths in a graph. *CACM*, 17(6):351–353, 1974.

[70] D. Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. STOC*, pages 565–573. ACM, 2014.

[71] J. B. Orlin and A. Sedeno-Noda. An $O(nm)$ time algorithm for finding the min length directed cycle in a graph. In *Proc. SODA*. SIAM, 2017.

[72] M. Pătraşcu and R. Williams. On the possibility of faster sat algorithms. In *Proc. SODA*, pages 1065–1075. SIAM, 2010.

[73] D. Peleg. Distributed computing: A locality-sensitive approach. *SIAM Monographs on discrete mathematics and applications*, 5, 2000.

[74] D. Peleg, L. Roditty, and E. Tal. Distributed algorithms for network diameter and girth. In *Proc. ICALP*, pages 660–672. Springer, 2012.

[75] D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *Proc. FOCS*, pages 253–261. IEEE, 1999.

[76] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[77] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Jour. Comput.*, 34(6):1398–1431, 2005.

[78] L. Roditty and V. Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proc. STOC*, pages 515–524. ACM, 2013.

[79] L. Roditty and V. V. Williams. Minimum weight cycles and triangles: Equivalences and algorithms. In *Proc. FOCS*, pages 180–189. IEEE, 2011.

[80] L. Roditty and U. Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Trans. Alg. (TALG)*, 8(4):33, 2012.

[81] P. Sankowski and K. Węgrzycki. Improved distance queries and cycle counting by Frobenius Normal Form. In *Proc. STACS*, pages 56:1–56:14, 2017.

[82] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Jour. Comput. Sys. Sci.*, 51(3):400–403, 1995.

[83] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. FOCS*, pages 605–614. IEEE, 1999.

[84] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Jour. Comput.*, 2(3):211–216, 2005.

[85] M. Thorup. Undirected single source shortest paths in linear time. In *Proc. FOCS*, pages 12–21. IEEE, 1997.

[86] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *CACM*, 13:722–726, 1970.

[87] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comp.*, 20(1):100–125, 1991.

[88] V. Vassilevska Williams. Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In *LIPIcs-Leibniz Intl. Proc. Informatics*, volume 43. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[89] H. Weinblatt. A new search algorithm to find the elementary circuits of a graph. *JACM*, 19:43–56, 1972.

[90] R. Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365, 2005.

[91] V. V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proc. IEEE FOCS*, pages 645–654. IEEE, 2010.

[92] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[93] R. Yuster. A shortest cycle for each vertex of a graph. *Inf. Proc. Lett.*, 111(21):1057–1061, 2011.

[94] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *JACM*, 49(3):289–317, 2002.

Vita

Udit Agarwal was born in Firozabad, India. He received a Bachelor's Degree in Mathematics and Computing from Indian Institute of Technology Guwahati in 2013. He joined the Computer Science doctoral program at the University of Texas at Austin in August 2014.

Permanent Address: udit@cs.utexas.edu

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.