DISTRIBUTED, PARALLEL AND DYNAMIC DISTANCE STRUCTURES

by

Yasamin Nazari

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland

June, 2021

Abstract

Many fundamental computational tasks can be modeled by distances on a graph. This has inspired studying various structures that preserve approximate distances, but trade off this approximation factor with size, running time, or the number of hops on the approximate shortest paths. Our focus is on three important objects involving preservation of graph distances: hopsets, in which our goal is to ensure that small-hop paths also provide approximate shortest paths; distance oracles, in which we build a small data structure that supports efficient distance queries; and spanners, in which we find a sparse subgraph that approximately preserves all distances.

We study efficient constructions and applications of these structures in various models of computation that capture different aspects of computational systems. Specifically, we propose new algorithms for constructing hopsets and distance oracles in two modern distributed models: the Massively Parallel Computation (MPC) and the Congested Clique model. These models have received significant attention recently due to their close connection to present-day big data platforms.

In a different direction, we consider a centralized dynamic model in which the input changes over time. We propose new dynamic algorithms for constructing hopsets and distance oracles that lead to state-of-the-art approximate single-source, multi-source and all-pairs shortest path algorithms with respect to update-time.

Finally, we study the problem of finding optimal spanners in a different distributed model, the LOCAL model. Unlike our other results, for this problem our goal is to find the best solution for a specific input graph rather than giving a general guarantee that holds for all inputs.

One contribution of this work is to emphasize the significance of the tools and the techniques used for these distance problems rather than heavily focusing on a specific model. In other words, we show that our techniques are broad enough that they can be utilized in different models.

Thesis Readers

Dr. Michael Dinitz (Primary Advisor)
Associate Professor
Department of Computer Science
Johns Hopkins University

Dr. Amitabh Basu
Associate Professor
Department of Applied Mathematics and Statistics
Johns Hopkins University

Dr. Xin Li
Associate Professor
Department of Computer Science
Johns Hopkins University

Dedicated to my parents and grandparents who have always encouraged me to pursue my passion and education wherever it took me.

Acknowledgements

First and foremost, I would like to thank my advisor, Mike Dinitz, for his support and guidance. Mike encouraged me to pursue my own path in research, even if it meant we published less than we could together. This helped me gain confidence and independence that will be tremendously helpful in the future. His true passion for CS theory has strengthened my own enthusiasm and perseverance. I would also like to thank my mentor, Amitabh Basu. Amitabh's patience and endurance has aspired me to be a better researcher and educator, and has made me fond of certain areas in math that once seemed intimidating to me.

I would also like to thank my internship hosts at Google, NYC: Jakub (Kuba) Łącki and Creighton Thomas. Kuba patiently guided me while I was learning a new area, and provided me with an enormous amount of helpful feedback. Both Kuba and Creighton made me realize that I would also very much enjoy an intriguing industry career, and they broadened my perspective in CS beyond academia. I would also like to thank my GBO and thesis committee members, Dr. Xin Li, Prof. William Cook, Prof. Rao Kosarajou, and Dr. Vladimir Braverman.

As I am concluding this phase of my studies, I would also like to acknowledge all the mentors and teachers who assisted me along the way to a PhD program: My master's supervisors and mentors Philipp Woelfel, George Giakkoupis, Petra Berenbrink and Peter Kling, who laid the foundation of CS theory research for me; My undergraduate instructors in Shiraz University of Technology and numerous high school teachers all of whom enforced my passion in Math and Computer Science.

Next, I would like to thank all my other collaborators, some of whom are also my coauthors. I have learned a great deal from each one of them, and I am confident that I will continue to do so in the upcoming years. Some of our problems might have remained unsolved, but each experience has helped me grow as a researcher. Namely, I thank (in alphabetic order): Melika Abolhassani, Sara Ahmadian, Mohammad Hossein Bateni, Amartya Shanka Biswas, Greg Bodwin, Michal Dory, Hossein Esfandiari, Venkat Gandikota, Mohsen Ghaffari, Ama Koranteng, Thomas Lavastida, Roie Levin, Slobodan Mitrović, Benjamin Moseley, Balasubramanian Sivan, Yifeng Teng, Leonidas Tsepenekas, Enayat Ullah, Zeyu Zhang and Goran Zuzic.

I am forever grateful to my family for their unwavering support and encouragement.I would specially like to thank my parents, grand parents, my brother Behzad, and his wife Leslie. It must have been particularly difficult for my parents to have both of their children be in the US, be unjustly banned from visiting them, and lose many milestones, including my brother's wedding. They endured all of this as they always put our success and future first. I would also like to thank my extended family in the US, my aunt and uncle (Niloofar and Reza) and my great aunt and uncle (Fati and John), who made sure I felt like home, despite being apart from the rest of our family.

During much of my PhD, the CS department at Hopkins became my home. My friends were always there for me, and helped me cope with a fair bit of non-academic stress. Just to name a few Hopkins and non-Hopkins friends[1], I am grateful to (in no particular order): Kestrel, Mahshid, Aarushi, Jalaj, Disa, Aditya, Alishah, Jasper, Eli, Jaron, Razieh, Rohit, Arka, Nikita, Teodore, Ravi, Huda, Chris Wilks, Charlotte, Gabby, Pouriya, Rachel Sherman, Max, Rob DiPietro, Leila, Johannes, Melanie, Mike Alonge, Maryam Neghbani, James, Chang and Zaoxing.

Last but not least, I would like to thank the 20 (and counting) foster cats who were a reliable source of joy and bliss through all the challenges I faced in the past 3

[1]The lists are in no way mutually exclusive even though I am not repeating any names.

years. Unfortunately, they cannot receive this note (except for the one who is still here and just stared at me when I thanked him out loud). But I would like to use any opportunity to encourage everyone to foster and volunteer with their local shelters to help pets in need.

As I am writing this, the world is still devastated by COVID-19. We are counting on science (and vaccines) to save us. Here is to hoping that we always use science wisely and for a good cause.

Funding. This work is generously funded by NSF awards CCF-1535887, CCF-1464239 and CCF-1909111, the Computer Science department at Johns Hopkins University, and Google (summer internship 2020).

Contents

Abstract ii

Dedication iv

Acknowledgements v

Contents viii

Chapter 1 Introduction 1

1.1 Models 3

1.1.1 Connections and Comparison 6

1.2 Distance Structures 8

1.2.1 Hopsets 9

1.2.2 Distance Oracles 11

1.2.3 Spanners 12

1.2.4 Connections 13

Chapter 2 Massively Parallel Distance Sketches and Distributed Distance Oracles 16

2.1 Introduction 16

2.1.1 Our Results 18

2.1.2 Our Techniques 22

2.1.3 Related Work 24

2.2 Preliminaries and Notation . . . 26

2.2.1 Notation . . . 26

2.2.2 Algorithmic Building Blocks . . . 26

2.3 Distance Oracles in Congested Clique . . . 30

2.4 Distance Sketches in Massively Parallel Computation Model . . . 36

2.4.1 Polylogarithmic Round Complexity . . . 42

2.4.2 Single-source shortest path . . . 47

2.5 Distance Oracles in the Streaming Model . . . 48

2.6 Comparison with Alternative Methods . . . 52

2.7 Discussion and Future Work . . . 53

Chapter 3 Sparse Hopsets in Congested Clique . . . 55

3.1 Introduction . . . 55

3.1.1 Our contribution . . . 57

3.1.2 Overview of techniques. . . . 64

3.1.3 Preliminaries . . . 67

3.2 Algorithmic Tools. . . . 69

3.3 Neighborhood covers using low-diameter decomposition . . . 70

3.4 Congested Clique Hopset Construction . . . 75

3.5 Massively Parallel Hopsets and MSSP . . . 80

3.6 Conclusion and Open Problem . . . 82

Chapter 4 Dynamic Hopsets . . . 83

4.1 Introduction . . . 83

4.1.1 Preliminaries and Notation . . . 85

4.1.2 Summary of Results . . . 86

4.2 Overview of Our Algorithms . . . 90

4.2.1 Static Hopset Construction . . . 91

4.2.2 Maintaining Restricted Hopsets Dynamically 93

4.2.3 Decremental Approximate Distances 99

4.3 Decremental Hopset 100

4.3.1 Maintaining a Restricted Hopset 101

4.3.2 New Hopsets with Improved Running Time. 105

4.3.3 Hopset stretch 114

4.4 Applications 125

4.4.1 $(1+\epsilon)$-approximate SSSP and $(1+\epsilon)$-MSSP 126

4.4.2 APSP distance oracles 127

4.5 Conclusion and Future Direction 130

Chapter 5 Distributed Distance Bounded Network Design 131

5.1 Introduction 131

5.1.1 Our Results 133

5.1.1.1 Solving convex programs 133

5.1.1.2 Distributed approximation algorithms for network design 136

5.1.2 Related Work 139

5.2 Preliminaries and Notation 141

5.3 Padded decompositions 142

5.4 Distributed distance bounded network design convex programming . . 146

5.4.1 Distance bounded network design convex programs 146

5.4.2 Distributed Algorithm 149

5.5 Distributed Approximation Algorithms for Network Design 154

5.5.1 Directed k-Spanner 155

5.5.2 Basic 3-Spanner and Basic 4-Spanner 157

5.5.3 Lowest-Degree k-Spanner 158

5.5.4 Directed Steiner Network with Distance Constraints 159

5.6 Conclusion and Future Work 160

Chapter 6 Conclusion and Future Direction **161**

Bibliography **164**

Chapter 1

Introduction

In recent years, due to the growth of massive datasets, the study of fundamental graphs problem in many big data and distributed platforms has gained significant attention. This motivates us to find efficient solutions to many fundamental algorithmic problems under various abstractions of modern computational systems.

A fundamental problem in all models of computation is the task of distance computation. In general, objects in many computational problems have an associated notion of distance or similarity. Hence in many problems involving graph analytics we need to compute some approximation of distances, or we need to *preserve* approximate distances in the graph while optimizing computational resources (such as space, communication, and time).

However in many situations computation of distances or preserving the distance structure in large-scale systems efficiently is quite challenging. This motivates us to study several different but related distance-based structures, such as distance sketches, spanners and hopsets in various computational models. As we will see, these structures provide us with different tradeoffs between size, accuracy and running time. These objects have gained significant attention as fundamental graph theoretic objects, in addition to related algorithmic applications such as shortest path computation, sparsification, routing, overlay networks, clustering, etc.

We study several different models each of which captures a different computational aspect of modern systems. Two main models we consider are the massively parallel computation model and the Congested Clique model, both of which are known to be closely connected with modern distributed platforms such as MapReduce, Hadoop, and Spark [50, 10, 54]. We also consider more classical distributed models, such as the LOCAL model and the CONGEST model, that capture theoretical restrictions on locality and congestion. Finally, we consider a dynamic model, that unlike other models we consider is sequential, but captures changing of input over time.

While the objects we study are of theoretical interest, we also use them to get state-of-the-art bounds in distance computation. In particular, in several of the models that we consider, our constructions have direct implications for single source, multi source, or all-pair shortest path computation. More broadly, many of the algorithms we study can be seen as a preprocessing step for constructing a sparse data structure that can be used for more efficient querying of distances. The preprocessing step may require more resources such as construction time.

Finally, while we mainly focus on constructing objects with guarantees that hold for an *arbitrary* input graph, in Chapter 5 we take a different perspective in which our goal is to find the best (minimal cost) solution for the specific instance considered. In particular, we consider distributed optimization of several network design problems, e.g. finding the minimal cost k-spanner in a given input graph. Solving these types of problems require a different set of techniques, such as linear programming. Interestingly, despite these differences a tool that we use for these optimization problems is a type of network decomposition that we also use in our sparse hopset algorithm in Chapter 3. Indeed throughout this thesis, we will see that similar ideas and techniques apply to seemingly different models and problems. One of our contributions is pointing out the significance of such techniques, by showing that they generalize to various models. This may in turn provide some insight into the connections between these models and their power and limitations.

First, we formally describe the models studied in this thesis, and then move on to formally defining the distance structures of interest, their applications, and connections between them.

Throughout this thesis, the input is an *undirected* graph $G = (V, E)$ with n vertices and m edges. In all chapters, except for Chapter 5, our results apply to *weighted* graphs.

1.1 Models

In all of the distributed models we study, time passes in *synchronous rounds* and in each round nodes exchange information in the network. The communication network, and the size of messages communicated in each model is different. The goal is to compute a solution to a problem in a small number of rounds of communication. Initially, each node knows its own portion of the input and the goal is for nodes to know their portion of the output (e.g. the incident edges in a desired subgraph or distance to a source). As we shall see, in the Massively Parallel Computation model, a node may be a collection of machines, or a machine may represent a collection of nodes.

LOCAL and CONGEST Models. In the LOCAL model [82], we are given an undirected graph $G = (V, E)$, and in each round every node can send an arbitrary message of unbounded size to each of its neighbors in G. In other words, the communication and input graph are the same in this model. As the name suggests, this model captures the locality of a problem, i.e., how many rounds of local computation is required to solve a problem globally.

In the CONGEST model [82], in each round nodes can send a message of $O(\log n)$-bits to each of their neighbors in G (different messages can be sent along different edges). In other words, similar to the LOCAL model, messages can only be sent along the input graph edges. However, unlike the LOCAL model, there is a bound on messages sent along each link in each round.

Congested Clique Model. In the Congested Clique model an input graph of $G = (V, E)$ is given, and initially each node $v \in V$ only knows its incident edges. However, the underlying communication graph is an undirected clique, and in each round every node can send a message of $O(\log n)$ bits to any other node. This model was introduced by [72], and has been studied extensively in recent years due to its connections with modern distributed platforms, such as MapReduce [54] and the Massively Parallel Computation model that we will define next.

Massively Parallel Computation Model. The *Massively Parallel Computation* or the MPC model was proposed by [10] as an abstraction of many modern distributed and parallel data processing platforms such as MapReduce, Spark and Hadoop. The model introduced by [10] refined previous abstractions proposed by [60, 50]. In this model, there is an input of size N distributed over N/S machines, each of which has $S = N^{\epsilon}, \epsilon < 1$ memory. Each machine can communicate with any other machine in each round but can have total I/O of at most S.

For graph problems, one particularly interesting and arguably more practical setting is the *low memory* regime in which each machine has memory strictly sublinear in the number of nodes (rather than edges). In other words, given a graph with n vertices and m edges, in low memory MPC, the memory per machine is $O(n^{\gamma}), 0 < \gamma < 1$, and the total memory is $O(m)$. Other memory regimes are the *linear memory* regime, in which, memory per machine is $\Theta(n)$, or the *super linear* memory regime, in which memory machine can be superlinear in n. For many graph problems the power of the model may significantly change in each of these three cases. We mostly focus on the low memory regime, which is the most challenging.

While in the standard MPC model the total memory over all machines is $O(m)$, we sometimes allow relaxations of this standard model in which the overall memory is slightly larger. We will see that such relaxations can sometimes significantly change the power of the model.

Dynamic Models. Next, we discuss a different type of model, which, unlike the previously described models, is centralized (sequential). Instead of communication limitations, the dynamic models capture changes of the input over time. A graph algorithm is called dynamic if it maintains a structure that supports answering *queries* about a graph which is undergoing modifications, or, as we say in the following, *updates* (see e.g. [46]). Each update is an edge deletion, insertion, or a weight change. We consider this model in Chapter 4.

The model that handles both insertions and deletions, called the *fully dynamic* model is quite challenging. That is why a large body of work (e.g. [15, 16, 21, 55]) focused on the the partially dynamic settings that can handle deletions only (the decremental model), or insertions only (the incremental model). In this thesis, we mainly focus on designing *decremental* algorithms for distance problems in weighted graphs in which, the updates are only edge deletions or weight increases.

PRAM and Streaming. We also come across two other models that are indirectly related to the distributed models described: the PRAM model (e.g. [49]) and the streaming model ([52, 44]). While they are not our focus, we give a brief discussion on their connection with the distributed models described, since we use some of the techniques in these models and in a few cases our results extend to these models.

In the PRAM model[1] parallel computation is performed by a group of processors

[1]We consider a simple abstraction without details of the exact parallel model (EREW, CRCW, etc), since PRAM is not our focus and there are reductions with small overhead between these

reading and writing on a shared memory. The number of parallel rounds of computation is called the *depth*, and the total amount of computation over all machines is called the *work*. In designing algorithms we wish to keep the depth and the work small, and there is often a tradeoff between these two performance measures.

In the streaming model, the input arrives based on a sequence of updates (insert only, delete only, or both) and we have space that is sublinear in the input size. In graph problems, there is often a treadeoff between number of passes and the space used, and our goal is to solve the problem in a small number of *passes* over the stream in a given space. While in the streaming model we also have a sequence of updates, unlike the dynamic model, we only require to maintain a solution after one or multiple passes are finished. Moreover, in the dynamic model we do not have a memory restriction, as we do in streaming model.

1.1.1 Connections and Comparison

In this section, we are going to give a high-level discussion on how these models with different goals and properties, relate to each other in some special settings. This discussion may be helpful in understanding the models better, but a reader may skip this section without loss of continuity.

However, we emphasize that one of the contributions of this thesis is to focus on the *techniques* used, rather than specific models. We aim to convey that for many distance-based graph problems, despite the differences, the techniques used to improve the performance of an algorithm in one model often leads to improvements in another model, even though we still need to use model-specific tools and modify the algorithms to optimize our bounds in each model.

It is known that the Congested Clique model is closely related to MPC. Specifically, when the graph is dense ($\Theta(n^2)$ overall memory), an algorithm in Congested Clique

variants.

that does not use more than $O(n)$ memory per machine can be implemented in the linear MPC model [12] in the same number of communication rounds. On the other hand an algorithm in linear MPC model can be implemented in the Congested Clique model in the same number of rounds. In general, the Congested Clique model is stronger than linear and low memory MPC, with total memory of $O(m)$. These reductions may no longer hold in variants where we relax the overall memory to $\tilde{O}(m)$ or larger.

It is easy to see that solving problems in the LOCAL and CONGEST model heavily rely on the network structure. On the other hand, in both MPC and Congested Clique the network topology is abstracted, and the input graph is not the same as the communication graph. Instead, communication is limited by size or number of messages exchanged.

The PRAM model also has close connections with the MPC model: [50] showed that a PRAM algorithm can be implemented in roughly the same number of rounds in MPC when the number of processors is (up to polylog factors) the same as the memory in MPC and polylogarithmic many rounds of computation is allowed.

While there are no known generic black-box reductions between streaming and distributed models, for some distance-based graph problems considered in this thesis the semi-streaming model (where memory is $O(n)$) is related to the Congested Clique model and the MPC regime with linear memory per machine. Intuitively, in this case the input stored in the streaming model can be stored in one machine (or node in case of Congested Clique), and depending on the problem, a pass of the stream can be implemented with one distributed round. We use this intuition in constructing distance oracles in streaming in Chapter 2. In related work on distance objects, similar connections between number of passes in streaming and distributed rounds was observed, such as for constructing hopsets [38] and spanners [17].

Despite the differences in what each model captures, in the context of graph problems, they sometimes have to deal with similar bottlenecks. For instance, in both CONGEST and Congested Clique, message size and the number of messages exchanged between nodes cause limitations that we need to overcome. But this type of congestion bottleneck also often translates to the communication limits among machines in MPC. Even though in more powerful models such as Congested Clique and MPC the number of rounds of communication is expected to be much smaller than in the CONGEST model, in designing algorithms for all of these models, we are interested in compressing the messages, restricting our focus to local neighborhoods with small overlap. This is why sparsification techniques, such as spanners, have significance in all of these models. On the other hand, as we will see while the techniques used for one model can be useful in implementing algorithm in another model, we often have to modify the structure of the objects we study to make them suitable to each specific model.

Finally, even though the LOCAL model seems to have very different focus than most models introduced here, there are still technical connections. Intuitively, this is because for many distance problems (and other graph problems such as connecitivity, or network design) many of the known algorithms in these models have a local growth structure (e.g. ball growing, neighborhood collection, Bellman-Ford, BFS). Hence the efficiency of such algorithms also depends on locality, and how fast local information can be gathered.

1.2 Distance Structures

In this section, we describe the fundamental objects that are going to be used throughout this thesis, and their application and connections. In different section, depending on the model, we may use different implementations of these objects. Throughout this section we assume that the input is a weighted and undirected graph $G = (V, E)$ with n vertices and m edges. We denote the distance between a pair of vertices $u, v \in V$ in G by $d_G(u, v)$.

1.2.1 Hopsets

An important object that we study through this thesis is a hopset. Informally, a hopset is a set of (weighted) edges added to a graph that while preserving distances approximately, reduces the maximum number of hops in the shortest paths among all pairs. In other words, these edges *shortcut* the shortest paths. More formally,

Definition 1.2.1. ((β, ϵ)- Hopset) For parameter $\epsilon, \beta > 0$, a graph $H = (V, E_H, w_H)$ is called a (β, ϵ)-hopset for the graph G, if in $G \cup H$, the graph obtained by adding E_H to G, we have: $d_G(u, v) \leq d^{(\beta)}_{G \cup H}(u, v) \leq (1 + \epsilon)d_G(u, v)$ for every pair $u, v \in V$ of vertices, where $d^{(\beta)}_{G \cup H}(u, v)$ is the length of the shortest path between u and v that has at most β hops in $G \cup H$. The parameter β is called the *hopbound* of the hopset.

Hopsets, originally introduced by Cohen [25], are widely used in distance related problems in various settings, such as parallel shortest path computation [25, 74, 35, 41], distributed shortest path computation [40, 76], routing tables [39] and distance sketches [39, 30]. In addition to their direct applications, hopsets have recently gained more attention (e.g. [13, 40, 1, 58]) as a fundamental object closely related to several other fundamental objects such as additive (or near-additive) spanners and emulators[43].

One of the common application of hopsets is in obtaining faster algorithms for shortest path computation in various models. For instance, in distributed (e.g. congested clique) or parallel (MPC, PRAM) models, after constructing a (β, ϵ)-hopset, we can compute $(1 + \epsilon)$-approximate single-source shortest paths in $O(\beta)$ rounds. In many applications we need to tradeoff the time needed for constructing a hopset, with the hopbound/stretch guarantees.

One limitation of hopsets in applications in $(1+\epsilon)$ single-source shortest paths is that their utility is limited for sparse graphs, i.e. they will only lead to suboptimal algorithms. Existential lower bounds imply that when $\epsilon < 1$, a hopset with polylogarithmic hopbound must have size at least $n^{1+\Omega(1)}$ [1]. This is why for very sparse graphs we cannot hope to obtain $\tilde{O}(m)$-time algorithms based on hopsets. However, when the graph is slightly denser ($|E| = n^{1+\Omega(1)}$), ϵ is larger ($\epsilon \geq 1$) [13, 35], or when we are computing distances from a large set of sources, this limitation no longer applies.

Throughout this thesis we use different types of hopset constructions, each of which is used for a specific model/application:

In Chapter 2, we use a hopset construction proposed by [38] to construct distributed distance oracles. In particular, we first construct a hopset with polylogarithmic hopbound, and then use the hopset to construct a distance oracle more efficiently in various models such as Congested Clique, MPC, and streaming. We also show applications of this MPC hopset algorithms in computing single-source shortest paths.

In Chapter 3, we propose a new sparse hopset construction in the Congested Clique model that uses a similar structure as the original hopsets proposed by Cohen [25].

In Chapter 4, we propose a new dynamic hopset construction that builds upon a construction by [41].

Different hopset constructions. In sequential/static settings there are three main efficient hopset constructions [25, 38, 41]. For applications in (sequential) shortest path computation these main hopset constructions lead to similar bounds (up to polylogarithmic factors).

In Chapter 2, we show how the hopsets of [38] can be implemented in MPC, but in other chapters we propose *new hopsets*. In particular, in Chapters 3, and 4, while our hopsets have structurally similarities with known hopsets, we need to modify the constructions to make them suitable for the specific models and applications that we consider.

1.2.2 Distance Oracles

In many situations we are interested in querying distances between a pair of nodes in a graph efficiently, but computing distance for each query is too slow, and we can not afford to store all distance pairs. This motivated Thorup and Zwick [88] to define the notion of an *approximate distance oracle*: a small data structure which can quickly report an approximation of the true distance for any pair of vertices. In other words, by spending some time up front to compute this data structure (known as the *preprocessing* step) and then storing it (which can be done since the structure is small), any algorithm used in the future can quickly obtain provably accurate distance estimates.

An approximate distance oracle is said to have *stretch* t if, when queried on $u, v \in V$, it returns a value $d'(u, v)$ such that $d(u, v) \le d'(u, v) \le t \cdot d(u, v)$ for all $u, v \in V$, where $d(u, v)$ denotes the shortest-path distance between u and v. The important parameters of an approximate distance oracle are the size of the oracle, the stretch, the query time, and the preprocessing time. For a parameter $k \ge 2$, Thorup and Zwick's construction (in the sequential setting) has expected size $O(kn^{1+1/k})$, stretch $(2k-1)$, query time $O(k)$, and preprocessing time $O(kmn^{1/k})$, where $n = |V|$ and $m = |E|$. Additionally, the Thorup-Zwick distance oracle consists of n smaller pieces corresponding to each node each of expected size $O(kn^{1/k})$. These are called *distance sketches* or *distance labelings*. The estimate between a pair of nodes u and v can be computed just from the sketch for u and the sketch for v, and there is no need for the rest of the structure. This property makes this structure particularly useful in distributed models in which each node has a memory or congestion limit.

Since the work of [88], there has been a large amount of followup work on improving the achievable tradeoffs, such as achieving query time of $O(1)$ with size $O(n^{1+1/k})$ [90, 20] or giving more refined bounds [80, 81]. However, with the notable exception of an interesting construction due to Mendel and Naor [73], the vast majority of followup work on this type of distance oracle has essentially been refinements and improvements to the approach pioneered by Thorup and Zwick. Thus understanding the Thorup-Zwick distance oracle is an important first step to understanding the limits and possibilities of distance oracles. This is our main focus in Chapter 2. We note that distance oracles with slightly different type of guarantees (such as allowing queries from multiple sources) have also recently been studied, for example in [35]. We also define a distance oracle with multi-source querying guarantees in Chapter 3.

In Chapter 2, we directly focus on computing Thorup-Zwick distance oracles and distance sketches. In this chapter our goal is to only keep a sparse data structure, and we do not need to store the original graph edges after the preprocessing step. But in Chapters 3 and 4 we still need to keep the original graph edges, and our main goal is to keep the query time small.

In particular, in Chapter 4 we maintain Thorup-Zwick distance oracles dynamically, and use this data structure for computing fast all-pairs-shortest path queries. But in order to maintain the data structure while keeping the update time small, we need to maintain a hopset, and the original graph. Similarly, in Chapter 3, we use a slightly different type of distance oracles. Again, our data structure supports fast (polylogarithmic rounds) distance queries, but we still need to store the original input graph edges. On the other hand, the data structure supports distance queries from a large number of sources (up to $O(\sqrt{n})$) simultaneously.

1.2.3 Spanners

Another related class of distance object are *graph spanners*. Informally, spanners are sparse subgraphs of a given graph that preserve the pairwise distances up to a multiplicative or additive (or both) factor. We mostly focus on multiplicative spanners, which were introduced by Peleg and Ullman [84] and Peleg and Schäffer [83], and are defined as follows.

Definition 1.2.2. Let $G = (V, E)$ be a graph (possibly directed), and let $k \in \mathbb{N}$. A subgraph H of G is a *k-spanner* of G if $d_H(u, v) \leq k \cdot d_G(u, v)$ for all $u, v \in V$. The value k is called the *stretch* of the spanner.

In Chapter 2, we use spanners to improve the round complexity of both our distributed distance oracle and hopset algorithms by trading off round complexity with accuracy. Informally, by first constructing a spanner, we can reduce an instance in the standard MPC model, to an instance in a stronger variant of MPC with extra memory. On the other hand, in this extra memory MPC variant, we can construct hopsets and distance oracles more efficiently.

We also consider spanners and several related objects in Chapter 5. As discussed, there we take the different perspective of finding the optimal spanner: given any graph, our goal is to find the best (minimum cost) spanner for that instance. While it is not our focus, it is observed (e.g. by [43]) that the algorithms for so-called near additive spanners (whose stretch has both an additive and $(1+\epsilon)$ multiplicative factor), are closely related to hopset constructions.

1.2.4 Connections

There are many structural and algorithmic connections between the objects defined. For instance, in this thesis we use hopsets to construct distance oracles more efficiently in various models: specifically we show this for MPC, Congested Clique, and the streaming model (all in Chapter 2), and for the dynamic model in Chapter 4.

Moreover, some hopset constructions have a structure similar to the sampling and clustering scheme of distance oracle construction by [88]. What leads to different guarantees, is the subsampling rates and the distance estimates stored (e.g. as edge weights). Going even further, as observed in [13] the Thorup-Zwick [88] distance oracles can be seen as hopsets with hopbound 2, and stretch $2k-1$. This intuition was extended in [13] to obtain a range of tradeoffs between stretch and hopbound. On the other hand, a (β, ϵ)-hopset can also be seen as a type of data structure that allow faster $(1+\epsilon)$-approximate distance queries: after adding hopset edges, in distributed and parallel models distances can be queried in $O(\beta)$ rounds.

Finally, as we discussed, we can use spanners as a general sparsification tool to construct hopsets, distance oracles or other distance objects, using less computational resources such as time, communication, or parallel work in exchange for accuracy.

Outline. We now give a brief outline of this thesis, which is divided in four main parts (Chapters 2 to 5):

In Chapter 2, we focus on constructing distance oracles and distances sketches in MPC, Congested Clique and Streaming models, and we use hopsets as tool for making our algorithms efficient. Depending on the model, and the approximation ratio, the round complexity of our algorithms are either a small polynomial, subpolynomial $n^{o(1)}$, or polylogarithmic. Once the distance oracles are constructed, the distance between a pair of nodes can be queried in at most 2 rounds of communication. In our final MPC algorithms in addition to hopsets, spanners are also used in obtaining polylogarithmic round complexity in MPC. As a side result we also obtain the first $(1+\epsilon)$-single source shortest path in low memory MPC. The results in this chapter are published in [30].

In Chapter 3, we focus on hopsets directly, and give a new efficient algorithm for constructing sparse hopsets in the Congested Clique model. In particular, we give the first algorithm for constructing a sparse hopset in Congested Clique in polylogarithmic rounds of communication. We also show an application of this construction in obtaining a distributed multi-source distance oracle that supports polylogarithmic round queries. The result of this chapter is published in [77].

In Chapter 4, we turn our attention to a dynamic model (specifically the decremental model), and provide algorithms for constructing hopsets in this model that have near-optimal (up to polylogarithmic factor) update time. These new hopsets lead to state-of-the-art update time for several important shortest path problems, such as $(1+\epsilon)$-approximate single-source shortest paths (SSSP), $(2k-1)$-approximate all-pairs-shortest-path (APSP), and $(1+\epsilon)$-approximate multi-source shortest path (MSSP). Our all-pairs-shortest-paths result is based on simultaneously maintaining low-hop hopset and a Thorup-Zwick [88] distance oracle decrementally. A manuscript of the results in this chapter can also be found in [59].

In order to get a more clear outline of the objects and models considered in the first three chapters, these are summarized in Table 1.2.4. Note that this table summarizes the main results of each chapter, whereas there may be side-results related to a different model/object not mentioned here.

Model \Problem	Hopset	Distance Oracle	Distance Computation
MPC	Chapter 2	Chapter 2	Chapter 2
Congested Clique	Chapters 2 & 3	Chapters 2 & 3	-
Dynamic	Chapter 4	Chapter 4	Chapter 4

Finally, in Chapter 5, we study a certain class of network design problems which we call *distance bounded*, in the LOCAL model. Our goal is to find a minimum cost subgraph with certain properties (e.g., connecting a pair of demands), but where the set of allowed paths have a bounded distance D. The round complexity of our algorithms depend on this distance bound D.

One specific example of this class of problems is the problem of finding the optimal k-spanner for the input graph. We provide an $O(\sqrt{n})$-approximation algorithm for the directed k-spanner problem that runs in $O(k \log n)$ rounds of the LOCAL model. This guarantee matches the best-known sequential approximation algorithm for this problem. Our results are based on a distributed algorithm for solving a special class of network design linear programs. The results of this chapter are published in [29].

Chapter 2

Massively Parallel Distance Sketches and Distributed Distance Oracles

2.1 Introduction

In this chapter, we focus on distance oracles, distance sketches, and their implementation and application in big data distributed platforms. In particular, we mainly focus on MPC and Congested Clique. The results in this chapter are published in [30].

To date, efficient distance computation in massive graphs remain a challenging task. While one side effect of our techniques is indeed a state of the art algorithm for shortest paths in MPC, the focus of this chapter is on getting around the limitations of these models by allowing preprocessing of the (distributed) graph. We can first spend some time building the *approximate distance sketches* (or an approximate distance oracle), as introduced in Chapter 1, which will then let us (approximately) answer any distance query using only 0, 1, or 2 rounds of network communication (depending on the precise model). Thus after this preprocessing, anyone who is interested in analyzing the massive graph has access to approximate distances essentially for free, making this a powerful tool for distributed graph analytics. For this we show that we can repurpose centralized data structures (in particular the Thorup-Zwick oracle [88]) by computing them efficiently in these new distributed models. Moreover, since our algorithms are derived from centralized data structures we even allow for efficient local computation in addition to efficient communication.

So our focus is on how to compute these data structures efficiently, since once they are computed, distance estimates become fast and easy. In the Congested Clique model, we can compute oracles a small polynomial number of rounds. In MPC, we can go even further and compute slightly suboptimal sketches in time that is only polylogarithmic. So while computing the data structure is still somewhat expensive, it is far more efficient than trivial approaches, and once it is computed, the analyst can receive approximate distances extremely quickly, allowing for low amortized cost or just the ability to do exploratory analysis without constantly waiting for expensive distance queries to complete.

It is also worth noting that in other related models such as PRAM, there are no known algorithms for (k-approximate) shortest paths in polylogarithmic depth (number of rounds), and use at most $\tilde{O}(m)$ processors. Hence, even for k-approximate shortest paths the bounds we achieve can not obtained by PRAM reductions directly- even though we indirectly use the PRAM algorithms in our final construction. For constructing distance oracles/sketches our MPC algorithms can also be extended to the PRAM model with additional polylog factors that depend on the exact type of PRAM.

Finally, while our main focus is on distributed models, as a side result, we also extend our algorithms to the (insert-only) streaming model.

Distance Oracles and Sketches. Even in many centralized applications, the time it takes to compute exact distances in graphs is undesirably slow, and similarly the memory that it would take to store all $\binom{n}{2}$ distances is also undesirable. This motivates the study of an approximate distance oracles. Recall the definition of an approximate distance oracle: a small data structure which can quickly report an approximation of the true distance for any pair of vertices. In other words, by spending some time up front to compute this data structure (known as the *preprocessing* step) and then storing it (which can be done since the structure is small), any algorithm used in the future can quickly obtain provably accurate distance estimates.

More formally, an approximate distance oracle is said to have *stretch t* if, when queried on $u, v \in V$, it returns a value $d'(u, v)$ such that $d(u, v) \leq d'(u, v) \leq t \cdot d(u, v)$ for all $u, v \in V$, where $d(u, v)$ denotes the shortest-path distance between u and v. The important parameters of an approximate distance oracle are the size of the oracle, the stretch, the query time, and the preprocessing time. For any constant k, Thorup and Zwick's construction (in the sequential setting) has expected size $O(kn^{1+1/k})$, stretch $(2k-1)$, query time $O(k)$, and preprocessing time $O(kmn^{1/k})$, where $n = |V|$ and $m = |E|$. Recall also, that the Thorup-Zwick distance oracle has the additional property that the data structure can be "broken up" into n pieces, each of size $O(kn^{1/k} \log n)$, that are called *distance sketches* or *distance labelings*,. The estimate $d'(u, v)$ can be computed just from the piece for u and the piece for v (the rest of the structure is unnecessary). Das Sarma et al. [86] initiated the study of Thorup-Zwick distance sketches in distributed networks, and in particular in the CONGEST model of distributed computing [82], which was later studied by [68], and [39].

2.1.1 Our Results

In this chapter we initiate the study of distance oracles and sketches in two popular computational models for "big data": Congested Clique and MPC. In addition, we show that our techniques can be used to give the first sublinear algorithm (and in fact polylogarithmic) for approximate single-source shortest paths for weighted graphs in (low memory) MPC, and moreover can be applied in straightforward ways to non-distributed models such as the streaming setting.

We discuss our results for each model in turn. At a high level, Congested Clique turns out to be relatively easy: we can essentially just combine the known CONGEST algorithm [86] with a slightly modified hopset construction. For MPC, the natural approach is to simulate the Congested Clique algorithm, since it is known [12] that under certain density and memory conditions, Congested Clique algorithms can be simulated in MPC. However, this simulation requires at least $\Omega(n)$ memory per machine. Our task becomes much more challenging if we allow $o(n)$ memory per machine, which we refer to as the *low memory* setting. Designing algorithms for this setting forms the bulk of this chapter.

Congested Clique. Since there is no memory restriction for Congested Clique, we assume that some node in the network is the *coordinator* at which the entire distance oracle will be stored (i.e., the machine with which users will interact with the distributed system). So at query time, the user can just query the coordinator locally (avoiding all network delay) rather than initiating an expensive distributed computation. The precise statements of our results are given in Section 2.3 and are somewhat technical, so for simplicity we state one particularly interesting corollary obtained by some specific parameter settings:

Theorem 2.1.1. *Given a weighted graph $G = (V, E, w)$, for all $k \geq 2$ and constant $\epsilon > 0$, we can construct a distance oracle with stretch $(1+\epsilon)(2k-1)$, (local) query time $O(k)$, and space $O(kn^{1+1/k}\log n)$ w.h.p. in the Congested Clique model. If $k = O(1)$, then the number of rounds for preprocessing is*[1] *$\tilde{O}(n^{1/k})$, and if $k = \Omega(\log n)$ then the number of rounds is $\tilde{O}(\log(n))$.*

Note that after a limited amount of preprocessing, distance queries can be computed

[1]The notation $\tilde{O}(f(n))$ stands for $O(f(n) \cdot \text{polylog}(f(n)))$, e.g. it is suppressing polyloglog(n) terms in $2^{\tilde{O}(\log n)}$.

without any network access whatsoever. Moreover, the computational query time is also extremely small, so these queries are extraordinarily efficient in the context of distributed algorithms. As an interesting extension, we show that the message complexity of computing this distance oracle can be reduced by adding an additional preprocessing step of computing a graph spanner.

MPC. In Section 3.5 we discuss the MPC model, which is the heart of this chapter. Since in the MPC model servers have small memory, it is impossible to fit an entire distance oracle at a single server as we did in the Congested Clique. So we instead focus on distance sketches. After the preprocessing algorithm, for each node $v \in V$, a distance sketch of size $O(kn^{1/k} \log n)$ will be stored and mapped to a machine with key v (this assumes that the memory at each server is at least $\Omega(kn^{1/k} \log n)$, which is reasonable in most settings). This means that after the preprocessing to construct these sketches, only two rounds of communication are needed for for approximating distance queries between a pair of nodes u and v: one for sending requests for the sketches of u and v and one for receiving them. We give the following result:

Theorem 2.1.2. *Given a weighted graph $G = (V, E, w)$ with polynomial weights*[2] *and parameters $\rho \leq \gamma \leq 1, 1/k \leq \rho, 0 < \epsilon < 1$, we can construct Thorup-Zwick distance sketches with stretch $(2k-1)(1+\epsilon)$ and size $O(kn^{1/k} \log n)$ w.h.p. in $\tilde{O}(\frac{1}{\gamma} \cdot n^{1/k} \cdot \beta)$ rounds of MPC(n^{γ}), where $\beta = \min(O(\frac{\log n}{\epsilon})^{\log(k)+k}, 2^{\tilde{O}(\sqrt{\log n})})$. In particular, if $k = O(1)$ and ϵ is a constant, then w.h.p. we require $\tilde{O}(n^{1/k})$ rounds, and if $k = \Theta(\log n)$ then w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ rounds.*

In the above theorem the distance sketches have the same guarantees as the centralized Thorup-Zwick distance oracles. However, in MPC a polynomial round complexity, while possibly of theoretical interest, is generally considered not practical.

[2]This assumption can be relaxed using reduction techniques (e.g. from [38]) in exchange for extra polylogarithmic factors in the hopbound and construction time.

So we give a different (but related) algorithm which achieves polylogarithmic round complexity, at the price of larger stretch.

Theorem 2.1.3. *Consider a graph $G = (V, E)$ where $m = \Omega(kn^{1+1/k}\log n)$, for any $k \geq 2$. Then there is an algorithm in MPC(n^γ) (with $0 < \gamma < 1$) that constructs Thorup-Zwick distance sketches with stretch $O(k^2)$ and size $O(kn^{1/k}\log n)$ and with high probability completes in $O(\frac{k}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k - 1})$ rounds.*

As a side effect of our techniques (which we discuss more in Section 2.1.2), we immediately get an algorithm for computing approximate single-source shortest paths (SSSP) in the MPC model, which is the problem of finding the (approximate) distances from a source node to all other nodes. Unlike Congested Clique, there do not seem to be any known nontrivial results for this problem in MPC. We first give an algorithm which computes a $(1+\epsilon)$-approximation in $n^{o(1)}$ time. Then we show that we can compute an $O(1)$-approximation in only polylogarithmic time, if we make an additional assumption about the density of the input graph. We will prove the following theorem in Section 2.4.2:

Theorem 2.1.4. *Given a weighted undirected graph $G = (V, E, w)$ with polynomial weights, a source node $s \in V$, and $0 < \gamma \leq 1, 0 < \epsilon < 1$ we can compute $(1+\epsilon)$-approximate SSSP w.h.p. in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine. Moreover, if $|E| \geq \Omega(n^{1+1/k}\log(n))$, we can compute $4k(1+\epsilon)$-approximate SSSP in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k - 1})$ rounds of MPC(n^γ), where $1/k < \gamma \leq 1, k \geq 2$. In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

Note that while the round complexity is polylogarithmic, it may still be somewhat slow for certain applications: an analyst who has to wait polylogarithmic rounds for every distance query would essentially be unable to perform any analysis which depended on large numbers of distance queries. On the other hand, our main results on distance sketches allows us to pay this round complexity only once, for constructing the sketch.

Streaming. Finally, we provide an algorithm for constructing distance oracles in the multi-pass streaming model. This is essentially a side-effect of our main results for Congested Clique and MPC, but we include it for completeness. For the specific settings of constant or logarithmic stretch, we have:

Corollary 2.1.5. *Given a graph $G = (V, E, w)$, there exists a streaming algorithm that constructs a Thorup-Zwick distance oracle of stretch $(2k - 1)(1 + \epsilon)$ of size $O(kn^{1/k} \log n)$ w.h.p. and expected space $O(n^{1+1/k} \cdot \log^2 n)$, such that if $k = O(1)$, w.h.p. we require $O(\log^k n)$ passes , and if $k = \Omega(\log n)$, w.h.p. we require $2^{\tilde{O}(\sqrt{\log n})}$ passes.*

Note that in case of $k = \Omega(\log n)$ we are in the so-called *semi-streaming* setting in which the total memory used is $O(n \cdot \text{polylog } n)$.

2.1.2 Our Techniques

Our main approach is to combine constructions of *hopsets* with efficient distributed constructions of Thorup-Zwick distance oracles/sketches. In particular, Das Sarma et al. [86] showed that Thorup-Zwick sketches could be computed in the CONGEST model, but the time depended on the graph diameter. So all that we really need to do is to reduce the diameter of the graph, since any CONGEST algorithm also works in the Congested Clique. This is what hopsets do: we discuss them in more detail in Section 2.2.2, but informally they allow us to reduce the diameter of the graph while preserving distances by adding in a carefully chosen set of weighted “shortcut" edges. Hopset constructions for the Congested Clique were given by Elkin and Neiman [38] (and more recently by[19]) so for Congested Clique we can essentially just combine result of [38] (or [19]) with [86] to get our result (modulo a small number of technicalities).

Moving to MPC introduces some significant technical difficulties, particularly when the space per machine is $o(n)$. Neither [86] nor [38] are written with MPC in mind, so we cannot simply “black-box" them as we could (mostly) in the Congested Clique. However, not surprisingly, both [86] and [38] use as a fundamental primitive a “restricted" version of the classical Bellman-Ford shortest-path algorithm that ends early, and it turns out that implementing this restricted Bellman-Ford is the main (although not the only) technical hurdle in adapting both of them to the MPC model.

When implementing restricted Bellman-Ford in low-memory MPC, the main difficulty is that since the memory at each server is $o(n)$, a single server cannot “simulate" a node in Bellman-Ford. It takes many machines to store the edges incident on any particular node, so we need to show that it is possible for many machines to simulate a single node in MPC without too much overhead. We show that this is indeed possible: Bellman-Ford and related algorithms can be implemented in low-memory MPC with very little additional overhead. Once we develop this tool, we argue that the hopsets of [38] can be constructed in low-memory MPC with essentially the same complexity as in the Congested Clique. Our implementation of Bellman-Ford and this hopset construction, as well as a few other primitives we develop for low-memory MPC (e.g., finding minimum or broadcasting on a range of machines), may be of independent interest.

Even after using hopsets, we would still need polynomial time for constructing constant stretch distance sketches. We overcome this issue and improve the running time using two ideas. First, we show that by relaxing the model to allow small additional total memory (either through extra space per machine or additional machines), we can run our algorithms in polylogarithmic number of rounds. So we just need to argue that there is a way of obtaining extra memory without actually changing the model assumptions. This is our second idea: by constructing a spanner we can sparsify the graph while keeping the memory per machine and number of machines the same. Thus from the perspective of the spanner, it will appear that we do indeed have “extra" memory. The idea of sparsifying the input to obtain extra resources has already proved to be powerful in related contexts (for example, [47] recently used spanners to give a work-efficient PRAM metric embedding algorithm). To the best of our knowledge, though, this idea has not yet appeared in the MPC graph algorithms literature.

Other hopset constructions. Throughout this thesis, we discuss several different hopset constructions for various models. While we used the construction of [38] in this chapter, we could also use the hopset construction in [41] for Congested Clique (which is close to what we use in Chapter 4), or hopsets of [25] (closer to the one used in Chapter 3) for MPC. The implementation of these hopsets in each model would be be different, and there are differences in the polylogarithmic factors in the hopbound.

2.1.3 Related Work

Distributed constructions of distance oracles and sketches have been studied extensively in the CONGEST model [86, 69, 39]. All of these algorithms have running times dependent on the graph diameter, while our algorithms run in time independent of the graph diameter. To the best of our knowledge, constructing distance oracles/sketches has not previously been studied for the Congested Clique or the MPC model. Similarly, hopsets have been used extensively in various models of computation for solving approximate SSSP ([56, 38]). Our result on hopset construction in low memory MPC also gives the first (approximate) SSSP algorithm in this model for weighted graphs (in Congested Clique there are more results known [38, 56, 11, 19], but these do not translate obviously to MPC when there is sublinear memory per machine). In a recent result, [19] gave an efficient Congested Clique algorithm that constructs hopsets of size $\tilde{O}(n^{3/2})$ with hopbound $O(\log^2(n)/\epsilon)$. Their hopsets are a special case of hopsets of [38]. In Section 2.3 we explain how their algorithm applies to our result.

In the PRAM model, shortest path computation is well studied (e.g. [25, 38]), and it is known that many PRAM algorithms can be simulated in the MPC model ([60, 50]). However, most of these algorithms use $\omega(|E|)$ number of processors, in which case the simulations of [60] and [50] do not directly apply as they assume that the number of processors is at most the input size. As we argue in Section 2.4.1 we will still utilize an extension of this simulation.

Finally, we note that distance problems have also been studied in related models such as the k-machine model ([63]). In this model they show a low bound of $\Omega(n/k)$ for computing shortest paths, where k is the number of machines. To the best of our knowledge, the exact connection between this model and the MPC model has not yet been studied[3].

Subsequent work. After [30], there has been several results with implications on shortest path computation in MPC and Congested Clique. Li [70] and Andoni et al. [3] presented algorithms for computing $(1+\epsilon)$-approximate single-sources shortest paths in polylogarithmic depth and $\tilde{O}(m)$ work. This translates to a polylogarithmic round algorithm for $(1+\epsilon)$-SSSP in low memory MPC. Results of [70] and [3] are based on continuous optimization techniques and are mainly suitable for SSSP, whereas hopset based solutions like ours are more suitable for multi-source shortest path. We will discuss this further in Chapter 3. These algorithms cannot be extended to construct distance sketches/oracles efficiently, which is our main focus in this chapter.

Finally, in [17], we proposed an efficient algorithm for constructing $O(k)$-spanners

[3]In the k-machine model generally the number of machines considered is small. The computational power of this model therefore seems very different from the low-memory MPC setting where there are many machines (possibly more than n), but each one has small memory. Moreover, the k-machine model does not bound the space on each machine and the IO bound is slightly different with MPC.

in low and linear memory MPC and Congested Clique. One consequence of the result of [17] is an $\tilde{O}(\log n)$-approximate APSP algorithm for weighted graphs that runs in $O(\log\log^2 n)$ rounds of linear memory MPC and Congested Clique. This result also does not lead to efficient algorithms for constructing distance sketches in MPC. However, [17] and other spanner constructions can be used to efficiently construct distance oracles in Congested Clique with weaker stretch/size tradeoff than the algorithm used in this chapter (for a fixed size using spanners leads to stretch $O(k^2)$, whereas we get $O(k)$).

2.2 Preliminaries and Notation

2.2.1 Notation

In a given weighted graph $G = (V, E)$, we denote the (weighted) distance between a pair of nodes $u, v \in V$ by $d_G(u, v)$. We may drop the subscript G when there is no ambiguity. We define the h *hop-restricted* distance between u and v to be the weight of the shortest path between u and v that uses at most h hops and denote this by $d^{(h)}(u, v)$.

We will denote the set of neighbors of a node $v \in V$ by $N(v)$. In a weighted graph G, we define the *shortest-path diameter* of G, denoted by Λ, to be the maximum over all $u, v \in V$ of the number of edges in the shortest $u - v$ path (so if the graph is unweighted this is the same as the diameter, but in weighted settings it can be larger than the unweighted diameter). Finally, a t-spanner of G is simply a subgraph which preserves distances up to a multiplicative t factor.

2.2.2 Algorithmic Building Blocks

In this section we describe the algorithms of [88], [86] and [38], that are the building blocks used in next sections.

Thorup-Zwick Distance Oracle. In this section, we briefly describe the centralized construction of the well-known Thorup-Zwick distance oracle [88]. Given an undirected weighted graph $G = (V, E, w)$ and $k > 1$, in the preprocessing phase of their algorithm they first create a hierarchy of subsets $A_0, A_1, ..., A_k$ by sampling from nodes of V in the following manner: set $A_0 = V$, and for $1 \leq i \leq k-1$, add every node $v \in A_{i-1}$ to the set A_i independently with probability $n^{-1/k}$. Set $A_k = \emptyset$ and for all $u \in V$ define $d(u, A_k) = \infty$. Let $B_i(u) = \{w \in A_i : d(u, w) < d(u, A_{i+1})\}$ for all $u \in V$ and $0 \leq i \leq k-1$, where $d(u, A_i)$ is the minimum distance between u and a node in the set A_i, and set $B(u) = \cup_{i=0}^{k-1} B_i(u)$. We also denote the node that has the minimum distance to u among all nodes in A_i by $p_i(u)$ and call this the i-center of u, and so $d(u, A_i) = d(u, p_i(u))$. The distance sketch for u consists of $\{p_i(u)\}_{i=0}^{k}$, the set $B(u)$, and the corresponding distances between these nodes and u. The distance oracle is just the union of the sketches for all $u \in V$. Thorup and Zwick showed that this data structure has size $O(kn^{1+1/k} \log n)$ w.h.p., and access to these sketches is enough for approximating distances between every pair of vertices in $O(k)$ time with stretch $2k - 1$. In all the settings we consider, after preprocessing the distance oracle/sketches, we can *locally* perform the query algorithm of [88] in $O(k)$ time.

Next, we explain a distributed construction of Thorup-Zwick distance *sketches* as described by Das Sarma et al. [86] for the CONGEST model. The sampling phase can easily be done in distributed settings. Then for finding $p_i(v), 1 \leq i \leq k$ for all nodes $v \in V$, we will do the following: in iteration i, define a virtual source node s_i, and for all nodes in $u \in A_i$ add an edge between u and s_i where $w(u, s_i) = 0$. Then we will only need to run the Bellman-Ford algorithm from s_i, and after $O(k\Lambda)$ time every node $u \in V$ knows $p_i(u)$ and $d(u, A_i)$. Finally, for all $1 \leq i \leq k$ we need to compute the distance from $w \in A_i \setminus A_{i+1}$ to all the nodes v for which $w \in B(v)$. Simply running a distributed Bellman-Ford independently from all the sources $w \in A_i \setminus A_{i+1}$ would be slow since due to congestion limit on each edge we cannot run all these in parallel at the same time. However, [86] argue that this can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ rounds in total (w.h.p), since each node v needs to forward messages in the runs of Bellman-Ford algorithm for a source w only if $w \in B(v)$. This means that, roughly speaking, each node v participates in $|B(v)| = O(kn^{1/k} \log n)$ runs of Bellman-Ford. Then by a simple round-robin scheduling scheme they show that running these Bellman-Fords for all sources in $A_i \setminus A_{i+1}$ can be done in $O(\Lambda \cdot kn^{1/k} \log n)$ without violating the congestion bound on each edge.

Query algorithm. The querying step can be performed locally in one machine after only two rounds of communication for storing sketches of a pair of nodes in one machine. However for completeness, we briefly review the (sequential) query algorithm of [88]. Given sketches of a pair of nodes $(u, v) \in V$ the query algorithm proceeds as follows: For each $0 \leq i \leq k-1$, we check if $p_i(u) \in B_i(v)$ or $p_i(v) \in B_i(u)$. Let j be the smallest level at which one of these conditions occur. Note that by construction $p_{k-1}(u) \in B(v)$ and $p_{k-1}(v) \in B(u)$, and this implies that $j \leq k-1$ exists. Then if the first condition holds, the distance estimate $\tilde{d}(u, v) = d(u, p_j(u)) + d(v, p_j(u))$ and if the second conditions holds we set $\tilde{d}(u, v) = d(u, p_j(v)) + d(v, p_j(v))$. Note that these distance are stored with the sketch and can be computed. This clearly takes $O(k)$ time (sequentially), and it can be shown (see [88]) that this estimate satisfies $\tilde{d}(u, v) = (2k-1)d(u, v)$-stretch.

Hopsets. Recall, that for parameter $\epsilon, \beta > 0$, a graph $G_H = (V, H, w_H)$ is called a (β, ϵ)-hopset for the graph G, if in graph $G' = (V, E \cup H, w')$ obtained by adding edges of G_H, we have $d_G(u, v) \leq d^{(\beta)}_{G'}(u, v) \leq (1+\epsilon)d_G(u, v)$ for every pair $u, v \in V$ of vertices. The parameter β is called the *hopbound* of the hopset.

We first give a high level overview of the (sequential) hopset construction of [38] here. In their algorithm, they consider each distance scale $(2^k, 2^{k+1}], k = 0, 1, 2, ...$ separately.

For a fixed distance scale $(2^k, 2^{k+1}]$ the algorithm consists of a set of *superclustering*, and *interconnection* phases. Initially, the set of clusters is $\mathcal{P} = \{\{v\}_{v \in V}\}$. Each cluster in $C \in \mathcal{P}$ has a cluster center which we denote by r_C. The algorithm uses a sequence $\delta_1, \delta_2, ...$ of distance thresholds and a sequence $\deg_1, \deg_2, ...$ of degree thresholds that determines the sampling probability of clusters. At the i-th iteration, every cluster $C \in \mathcal{P}$ is sampled with probability $1/\deg_i$. Let S_i denote the set of sampled clusters. Now a single shortest-path exploration of depth δ_i (weighted) from the set of centers of sampled clusters $R = \{r_C \mid C \in S_i\}$ is performed. Let $C' \in \mathcal{P} \setminus S_i$ be a cluster whose center $r_{C'}$ was reached by the exploration and let r_C be the center in R closest to r'_C. An edge $(r_C, r_{C'})$ with weight $d_G(r_C, r_{C'})$ is then added to the hopset. A supercluster $\hat{C}$ with center $r_{\hat{C}} = r_C$ is now created that contains all the vertices of C and the clusters C' for which a hopset edge was added. In the next stage of iteration i, all clusters within distance $\delta_i/2$ of each other that have not been superclustered at iteration i will be interconnected. In other words, a *separate* exploration of depth $\frac{\delta_i}{2}$ is performed from each such cluster center r_C and if center of cluster C' is reached, an edge (r_C, r'_C) with weight $d_G(r_C, r_{C'})$ will be also added to the hopset. The final phase of their algorithm only consists of the interconnection phase. We denote the hopset edges added for distance scale $(2^k, 2^{k+1}]$ by H_k. One important property of this hopset construction (proved in Lemma 3.3 of [38]) that we will need for our analysis in Section 3.5) is the following:

Lemma 2.2.1 ([38])**.** *In the i-th iteration of a given distance scale $(2^k, 2^{k+1}]$, for each node $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(deg_i \cdot \log n)$, where deg_i is the sampling probability of the superclustering phase.*

Now we turn our attention to efficient construction of hopsets in distributed settings (such as CONGEST and Congested Clique) also proposed by [38]. Note that each superclustering phase can be performed by a distributed Bellman-Ford exploration of depth δ_i. For an interconnection phase, a separate distributed Bellman-Ford explorations of depth $\delta_i/2$ from cluster centers is performed. These Bellman-Ford algorithms can easily be implemented sequentially, however, in distributed settings, $O(n)$ rounds may be needed for each of the explorations of the larger scales. To overcome this issue, [38] propose to use the hopsets $\cup_{\log\beta-1<j\leq k-1} H_j$, for constructing hopset edges H_k. More precisely, they observe that for any pair of nodes with distance less than 2^{k+1}, hopsets $\cup_{\log\beta-1<j\leq k-1} H_j$ provide a $(1+\epsilon)$-stretch approximate shortest path with $2\beta+1$ hops between these pair of nodes. In other words, it is enough to run each Bellman-Ford exploration only for $O(\beta)$ rounds.

2.3 Distance Oracles in Congested Clique

In this section, as a warm-up, we will explain how a distance oracle can be efficiently constructed in the Congested Clique model. We will use the algorithm described in Section 2.2.2 by Das Sarma et al. [86] that constructs Thorup-Zwick distance *sketches* with stretch $2k-1$ and size $kn^{1+1/k}$ in $O(k\Lambda n^{1/k})$ rounds in the CONGEST model, where Λ is shortest-path diameter. Our algorithm is similar to their algorithm, with the difference that we first construct a hopset. This will allow us to terminate the algorithm earlier while preserving the distances within a $(1+\epsilon)$ factor. Constructing hopsets in the Congested Clique model can be done more efficiently than in CONGEST model. Hence, unlike the known algorithms in the CONGEST model, we can build a distance oracle in time independent of the shortest path diameter.

In this section, we can use the Congested Clique hopset algorithm of [38] directly as a black-box. Hence we mainly need to describe the distance oracle construction after adding the hopset edges. First we formally state a theorem proved in [38] for hopset construction in Congested Clique.

Theorem 2.3.1 ([38]). *For any graph $G=(V,E,w)$ with n vertices, and parameters*

$2 \le \kappa \le (\log n)/4, 1/2 > \rho \ge 1/\kappa$ and $0 < \epsilon < 1$, there is a distributed algorithm for the Congested Clique model that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{n^\rho}{\rho} \cdot \log^3 n \cdot \beta)$ rounds whp, where $\beta = O(\frac{\log(n)\cdot(\log\kappa + 1/\rho)}{\epsilon\cdot\rho})^{\log\kappa + \frac{1}{\rho}}$.

Roughly speaking, adding a (β, ϵ)-hopset to a graph is as if the shortest path diameter is reduced to β in exchange for a small loss in the stretch. In other words, hopsets will let us cut of distance computation after exploring β hops. Later on we will explain how we can set the parameters ρ and κ depending on the stretch parameter for the distance oracle, to get our desired running time.

We need the *h-restricted* distributed Bellman-Ford subroutine (Algorithm 1) which is widely used in previous work on distributed distance estimation (e.g. see [69], or [86]). We use the following lemma that follows from basic properties of Bellman-Ford algorithm.

Algorithm 1: Distributed Bellman-Ford with hopbound h.

```
Input : Undirected weighted graph $G = (V, E, w)$, and source node $s \in V$.
Output: $h$-hop restricted distances from the source $s$ to all nodes $u \in V$,
        $d^{(h)}(s, v)$.
1 Set $\forall v \in V : \hat{d}(s, v) = \infty$.
2 for Rounds $i = 0$ to $h$ do
3     for $\forall v \in V$ do
4         if $\exists u \in N(v) : \hat{d}(s, v) > \hat{d}(s, u) + w(u, v)$ then
5             Set $\hat{d}(s, v) := \min_{u \in N(v)}(\hat{d}(s, u) + w(u, v))$, and send $\hat{d}(s, v)$ to all
              neighbors.
```

Lemma 2.3.2. *There is a distributed variant of the Bellman-Ford algorithm runs in $O(h)$ rounds in Congested Clique and for all nodes $u \in V$, computes $d_h(s, u)$, the length of the shortest path between s and u among the paths that have at most h edges.*

In order to compute the shortest path from s to all nodes, we will need to set $h = \Lambda$, the shortest path diameter. But this can be as large as $\Omega(n)$. Hence we will use a (β, ϵ)-hopset to approximately find the distance in $O(\beta)$ time only. In other words, by constructing a (β, ϵ)-hopset H, we would know that there is a path of hopbound β with length $(1+\epsilon)d(u,v)$ among any pair of nodes $u, v \in V$, and hence Algorithm 1 can approximate the distances $d(s,v)$ up to a factor of $(1+\epsilon)$ for all $v \in V$.

We now argue that a distance oracle can be constructed by first preprocessing the input graph by constructing a (β, ϵ)-hopset (by Theorem 2.3.1) and then running the algorithm of [86] that was described in Section 2.2.2 for $O(\beta)$ rounds. Let us first state the result of [86] in the following theorem.

Next we review the distance oracle algorithm for the Congested Clique model. This algorithm was proposed by [86] for constructing distance sketches in the CONGEST model, which can clearly also be implemented in the Congested Clique model.

By extending the result of [86] to Congested Clique, we get the following result:

Theorem 2.3.3 ([86])**.** *Given undirected graph $G = (V, E, w)$ with shortest path diameter Λ, there is an algorithm that runs in $\tilde{O}(\Lambda \cdot kn^{1/k} \log n)$ rounds of Congested Clique w.h.p. and outputs a Thorup-Zwick distance oracle with stretch $(2k-1)$ at the coordinator with high probability.*

Since the algorithm in [86] is for the CONGEST model, we are not yet using the extra power of the Congested Clique model here, rather, we will use this power for constructing hopsets more efficiently. Moreover, in [86] distance *sketches* are constructed at each node. It is easy to see that nodes can then send their sketches to the coordinator within a constant factor of total number of rounds required to build a distance oracle consisted of the sketches for all nodes. Next, we will utilize the hopset construction of [38] to make the preprocessing algorithm more efficient with respect to time and message complexity. Let $G' = (V, E \cup H, w')$ be the graph obtained by adding a (β, ϵ)-hopset H to the undirected graph $G = (V, E, w)$. By running the algorithm in Theorem 2.3.3 on G' we will get the following result.

Corollary 2.3.4. *Given a graph $G = (V, E, w)$ and a (β, ϵ)-hopset H for G, there* *is an algorithm that runs in $\tilde{O}(\beta \cdot kn^{1/k})$ rounds of Congested Clique and outputs a Thorup-Zwick distance oracle with stretch $(2k-1)(1+\epsilon)$ on the graph $G' = (V, E \cup H, w')$ at the coordinator with high probability.*

Next we will analyze the message complexity of algorithm of Theorem 2.3.3 and show that w.h.p. $\tilde{O}(kmn^{1/k}\beta)$ messages need to be exchanged. It is not hard to see that the number of messages exchanged for constructing a (β, ϵ)-hopset is $\tilde{O}(\beta n^{1+\rho}/\rho)$ (this follows by analysis of [38]). Hence the dominant number of messages exchanged is for running the algorithm of Theorem 2.3.3.

Lemma 2.3.5. *Total number of messages exchanged for constructing a Thorup-Zwick distance oracle (with parameters specified in Corollary 2.3.4) on graph $G' = (V, E \cup H, w')$ is w.h.p. $O(\beta m \cdot kn^{1/k} \log n)$.*

Proof. The algorithm of Corollary 2.3.4 runs in $O(\beta kn^{1/k} \log n)$ rounds w.h.p. and overall for each edge in the graph $O(1)$ messages are exchanged. □

We now combine the hopset construction and Theorem 2.3.3 together to obtain our main result. We will use Theorem 2.3.1 to construct a hopset H on graph $G = (V, E, w)$, and then run the algorithm of Theorem 2.3.3 on the obtained graph $G' = (V, E \cup H, w')$ and get the following:

Theorem 2.3.6. *Given a graph $G = (V, E, w)$, polynomial weights[4] and parameters $2 \leq \kappa \leq (\frac{\log n}{4})$, $1/\kappa \leq \rho \leq 1/2, 0 < \epsilon < 1$, we can construct a Thorup-Zwick distance oracle with stretch $(2k-1)(1+\epsilon)$ and size $O(kn^{1+\frac{1}{k}} \log n)$ w.h.p. in $O(\beta(\frac{n^\rho}{\rho} \cdot \log^3 n + n^{\frac{1}{k}} \log n))$ time, where $\beta = O(\frac{\log(n) \cdot (\log \kappa + 1/\rho)}{\epsilon})^{\log \kappa + \frac{1}{\rho}}$.*

The running time depends both on the parameter ρ and stretch k. In other words, there is a tradeoff between the stretch k and the running time of this algorithm. When

[4]Same as in other models this assumption can be relaxed using techniques of [38] in exchange for extra polylogarithmic factors.

stretch k is smaller, we can choose a larger value for ρ and the dominant part of the running time would still be the distance oracle construction. On the other hand, for larger values of stretch k, since the distance oracle construction algorithm can be performed more efficiently, we need to set ρ smaller to balance out the running time of constructing a hopset and that of constructing the distance oracle over the new graph. The parameter κ mostly just impacts the hopset size and the constant factor in the exponent of hopbound β. Let us consider two special cases of $k = O(1)$ and $k = \Omega(\log n)$ to understand these bounds better. In the special case of $k = \Omega(\log(n))$ the hopset construction step takes more time, and so we use the recent result of [19] for the hopset construction to get a polylogarithmic running time. They construct a hopset of size $\tilde{O}(n^{3/2})$ with hopbound $O(\log^2(n)/\epsilon)$ in $O(\log^2(n)/\epsilon)$ rounds.

Corollary 2.3.7. *Given a graph $G = (V, E, w)$, and constant $0 < \epsilon \leq 1$, we can construct a Thorup-Zwick distance oracle with stretch $(2k-1)(1+\epsilon)$ in the Congested Clique model, s.t.,*

- *In case $k = O(1)$, w.h.p. we require $\tilde{O}(n^{1/k})$ rounds.*
- *In case $k = \Omega(\log n)$, w.h.p. we require $\tilde{O}(\log(n))$ rounds.*

Proof. For stretch $k = O(1)$ we use Theorem 3.1.1 and set $\rho = 1/\kappa$, and $\kappa = k$ to get $\beta = \tilde{O}(\log(n))$ and total running time $\tilde{O}(n^{1/k})$. In case $k = \Theta(\log n)$, we will set $1/\kappa = \rho = \sqrt{\frac{\log\log n}{\log n}}$. In both cases, by setting ρ to be a smaller constant, such as $\rho = 1/2$ we can have a smaller β (but still polylogarithmic), but the preprocessing algorithm will use the larger space of $\tilde{O}(m + n^{1+\rho})$ space and communication, rather than $\tilde{O}(m + n^{1+1/k})$. In the special case $k = \Omega(\log(n))$, we use the hopset algorithm of [19], which takes polylogarithmic time.

□

Communication Reduction with Spanners In this section, we will describe how spanners can be used as a tool for reducing communication in exchange for an extra factor in the stretch. Recall, A t-spanner of a graph G is a subgraph H such that $d_G(u,v) \leq d_H(u,v) \leq d_G(u,v)$ for all $u, v \in V$. We will use the spanner construction of [9] which computes spanners efficiently in the more restricted CONGEST model.

Theorem 2.3.8 ([9]). *For any weighted graph, a $(2t-1)$-spanner of expected size $O(tn^{1+1/t})$ can be computed in the CONGEST model in $O(t^2)$ rounds and $O(tm)$ message complexity.*

This construction allows us to turn the input graph for algorithms described in this section into a sparser graph. By doing so we will lose a factor of t in the approximation ratio but we only need to run algorithm of Theorem 2.3.3 on a graph with $O(n^{1+1/t})$ edges. Hence, we first run the Algorithm of Theorem 2.3.8 to get a spanner G_t, and then run the algorithm of Theorem 3.1.1 on G_t. Then by Lemma 2.3.5 we have,

Theorem 2.3.9. *Given a graph $G = (V, E, w), t, k > 1$, we can construct a Thorup-Zwick distance oracle of size $O(kn^{1+1/k} \log n)$ with stretch $t \cdot (2k-1)(1+\epsilon) = O(kt)$ w.h.p. with total communication of $\tilde{O}(kn^{1/t+1/k}\beta + tm)$, where β and the running time are the same as in Theorem 3.1.1.*

This implies that there is a direct tradeoff between the approximation ratio and the amount of communication when size of the distance oracle is fixed. In other words, when $n^{1/t} = o(m)$ the amount of communication required for distance oracles of stretch $O(kt)$ is smaller than the amount required for building distance oracles of stretch $O(k)$, where the size is in both cases $O(kn^{1+1/k} \log n)$.

2.4 Distance Sketches in Massively Parallel Computation Model

In this section we will focus on the MPC model. First we provide MPC algorithms for constructing distance sketches that have the same guarantees (with respect to the stretch/size tradeoff) as the centralized construction of Thorup-Zwick that run in polynomial (or slightly subpolynomial) time. Then in Section 2.4.1 we show how we can bring down the running time to polylogarithmic in exchange for a loss in accuracy.

First, we note that it is known from [12] that for *dense graphs* with $O(n^2)$ edges every Congested Clique algorithm (in which nodes use local memory of $O(n)$) can be implemented in the MPC(n) model. Therefore, when memory per machine is $\Omega(n)$ and the graph is dense all the Congested Clique results discussed in Section 2.3 also hold, except that we store the distance sketches rather than a central distance oracle. The more interesting case is when memory per machine is strictly sublinear in n. For the rest of this section we will turn our attention to the case where the memory is n^γ, where $0 < \gamma \leq 1$ (i.e., strictly sublinear). For simplicity we assume that we can store the sketches in a single machine. Namely, we require $\tilde{O}(n^{1/k})$ memory per machine for stretch $O(k)$ distance sketches. This assumption can be relaxed (and in exchange the query algorithm will take $O(k)$ rounds instead of 2 rounds).

One main subroutine that we need is the *restricted Bellman-Ford* algorithm. We then need to run many instances of this algorithm in parallel and handle other technicalities both for constructing hopsets, and then the distance sketches. First, we require following subroutines that will allow us to simulate one round of Bellman-Ford in MPC(n^γ):

Sorting [**50**]. Given a set of N comparable items, the goal is to have the items sorted on the output machines, i.e. the output machine with smaller ID holds smaller items.

Indexing [2]. Suppose we have sets $S_1, S_2, ..., S_k$ of N items stored in the system. The goal is to compute a mapping f such that $\forall i \in [k], x \in S_i$, x is the $f(S_i, x)$-th element of S_i. After running this algorithm the tuple $(x, f(S_i, x))$ is stored in the machine that stores x.

Find Minimum (x, y). Finds the minimum of N values stored over a contiguous set of machines given ID x of the first machine and ID y of the last machine.

Broadcast (b, x, y). Broadcasts a message b to a contiguous group of machines given ID x of the first machine and ID y of the last machine.

The sorting and indexing subroutines can be performed in $O(1/\gamma)$ rounds of MPC(n^γ) ([2, 50]). We argue that we can solve the Find Minimum and Broadcast problems also in $O(1/\gamma)$ rounds of MPC(N^γ) in the following theorem. At a high-level we use an *implicit* aggregation tree of depth $O(\log_{N^\gamma} N) = \frac{1}{\gamma}$.

Theorem 2.4.1. *Given N items over a contiguous range of machines x to y, subroutines Find Minimum(x, y) can be implemented in $O(1/\gamma)$ rounds of MPC(N^γ). Moreover, the subroutine Broadcast(x, y) can also be implemented in $O(1/\gamma)$ rounds of MPC(N^γ).*

Proof. We will first define a rooted *aggregation tree* $\mathcal{T}$ with branching factor N^γ where the machines $M_x, ..., M_y$ are placed at the leaves (here M_x denotes the machine with ID x). W.l.o.g assume that the machines in this range have increasing and sequential IDs. Note that we don't need to store this tree explicitly, and we only need each node to know its parent. Consider level ℓ of the tree (leaves have $\ell = 0$). Each node in this level is a machine associated with the label ℓ. For each node in level $\ell - 1$ that has the i-th machine in its subtree, we set as its parent $M_{p(i,\ell)}$ where $p(i, \ell) = x + \lfloor \frac{i}{N^{\ell\gamma}} \rfloor$. Thus each machine can compute its parent given the label ℓ. Similarly, each machine can compute the indices of its children (as a range). In other words, at each level ℓ, we assign each group of N^γ nodes of this tree to a parent node at level $\ell + 1$.

The algorithm Find Minimum proceeds as follows: at each round ℓ, each machine first computes minimum over its the values it knows, and then sends the outcome to the parent machine. Finally, the minimum will be computed and stored at the root machine, which may forward the value to another destination. The algorithm Broadcast will similarly use an aggregation tree, but this time it routes the message top-down. First message b is sent to the first machine M_x, and then starting from M_x in each round any machine that receives message b sends this value to all of its children, which can be determined from the machine's ID and y. Eventually all the machines at the leaves will receive b. The number of rounds each of these subroutines take are the height of the aggregation tree which is $O(\log_{N^\gamma} N) = \frac{1}{\gamma}$. $\square$

Running the (restricted) Bellman-Ford algorithm in MPC is not as straightforward as it is in the Congested Clique. One challenge is that for high-degree nodes, the edges corresponding to a single node are distributed over a set of machines. Therefore, for each round of Bellman-Ford these machines must communicate for computing and updating the distance estimates. Another hurdle is the fact that since nodes have different degrees, we do not have the range in which edges corresponding to a given node are stored a priori. To overcome these challenges we need to use the described subroutines, and for that we need to perform some preprocessing to append each edge with a tuple that we will describe shortly.

We will show how we can create and maintain the following setting: Given a graph $G = (V, E)$, the goal is to store all the edges incident to each node v in a contiguous group of machines, which we denote by $M(v)$. More precisely, let $M_1, ..., M_P$, where $P= O(\frac{m}{n^\gamma})$, be the list of machines ordered by their ID, and let $v_1, ..., v_n$ be the list of vertices sorted by their ID. $M(v_i)$ consists of the i-th smallest contiguous group of machines, such that $|M(v_i)| = \lceil \frac{\deg(v_i)}{n^\gamma} \rceil$.

Throughout the algorithm, let $M_{(u,v)}$ denote the machine that stores the edge

(u, v). Also, for all $u \in V$, let r_u be the first machine in $M(u)$, and for any edge $(u, v) \in E$ let $i_u(v)$ be the index of (u, v) (based on the lexicographic order) among all the edges incident to v. We need to compute and store the following information at $M_{(u,v)}$: $\deg(u), \deg(v)$, r_u, r_v, i_u, i_v (here by storing r_u we mean ID of r_u, and for simplicity we refer to $i_u(v)$ as i_u). We first explain how these labels can be computed for all edges in $O(\frac{1}{\gamma})$ rounds in the following lemma.

Lemma 2.4.2. *Let $M_{(u,v)}$ be the machine that stores a given edge (u, v). We can create tuples of the form $((u, v), \deg(u), \deg(v),\ r_u, r_v, i_u, i_v)$, stored at $M_{(u,v)}$ for all edges in $O(\frac{1}{\gamma})$ rounds in MPC(n^γ), where $\gamma < 1$.*

Proof. Let $N(v)$ be the set of edges incident on node v. Without loss of generality, let us assume that both tuples of form (u, v) and (v, u) are present in the system for each edge and we assume $(u, v) \in N(u)$ and $(v, u) \in N(v)$ (note that the graph is still undirected). First, we use the indexing subroutine of [2] on the sets $\{N(v)\}_{v \in V}$ to store index i_u at $M_{(u,v)}$ and index i_v at $M_{(v,u)}$. After this step tuples of form $((u, v), w(u, v), i_u)$ are stored at $M_{(u,v)}$.

Then we sort the tuples based on edge IDs lexicographically, using sorting algorithm proposed in [50]. This will result in the setting described above in which edges incident to each node u are stored in a contiguous group of machines $M(u)$. Now in order to compute $\deg(u)$, machines will check whether they are the last machine in $M(u)$ either by scanning their local memory or communicating with the next machine. Then the last machine in $M(u)$ sets $\deg(u)$ to the maximum index i_u it holds. This machine can also compute r_u, ID of the first machine in $M(u)$ (using $\deg(u)$), and then broadcasts $\deg(u)$ and r_u to all machines in $M(u)$. At the end of these computations, each tuple $((u, v), w(u, v), i_u)$ will be replaced by the tuple $((u, v), w(u, v), r_u, i_u, \deg(u))$. Next, we sort these tuples again but this time based on the ID of the smallest endpoint. In other words, for each edge $(u, v) \in E$, both tuples $((u, v), w(u, v), i_u, \deg(u))$ and

$((v,u), w(v,u), i_v, \deg(v))$ will be at the same machine. Now we can easily merge these two tuples to create tuples of form $((u,v), w(u,v), i_u, i_v, \deg(u), \deg(v))$. □

After computing the tuples, we use the sorting subroutine again to redistribute the edges into the initial setting of having contiguous group of machines $M(u)$ for all $u \in V$. After these preprocessing steps, we are ready to perform updates required for the restricted Bellman-Ford algorithm. A summary of this algorithm is presented in Algorithm 2.

Algorithm 2: Restricted Bellman-Ford in MPC(n^γ).

```
Input  : Graph $G = (V, E)$ distributed among machines $M_1, ..., M_P$ and
         source $s$.
Output : $h$-hop restricted distances from the source $s$ to all nodes $u \in V$,
         $d^{(h)}(s, v)$.
1 Create the tuple $((u,v), i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each edge
    $(u,v) \in E$ (by Lemma 2.4.2).
2 Sort the edges lexicographically so that edges incident to $v$ are stored in a
    contiguous group of machines $M(v)$ (by [50]).
3 for $i = 0$ to $h$ do
4     for $v \in V$ do
5         Compute $\hat{d}(s,v)$ by finding (using Theorem 2.4.1
            $\min_{u \in N(v)} \hat{d}(s,u) + w(u,v)$).
6         Broadcast updated distances to everyone in $M(v)$ (also by Theorem
            2.4.1).
7         Each machine in $M_{(v,u)}$ sends $\hat{d}(s,v)$ to $M_{(u,v)}$ (located at $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$).
```

Theorem 2.4.3. *Given a graph $G = (V, E)$ and a source node $s \in V$ the restricted Bellman-Ford algorithm (Algorithm 2) computes distances $d^{(h)}(s, v)$ for all $v \in V$ in $O(\frac{h}{\gamma})$ rounds of MPC(n^γ).*

Proof. After storing the tuples $(i_u, i_v, r_u, r_v, \deg(u), \deg(v))$ at $M_{(u,v)}$ for each $(u,v) \in E$, the restricted Bellman-Ford algorithm proceeds as follows: in each round, for each node v, we first find the minimum distance estimate for v and send it to r_v. Then r_v will broadcast the minimum distance found to all the machines in $M(v)$. By Theorem

2.4.1 both of these operations take $O(1/\gamma)$ rounds. Then for each $(v,u) \in N(v)$, $M_{(v,u)}$ sends the updated distance directly to $M_{(u,v)}$, which islocated at index $r_u + \lfloor \frac{i_u}{n^\gamma} \rfloor$. All the operations for each of the h iterations of Bellman-Ford take $O(1/\gamma)$ rounds. □

We now need to argue that hopsets of [38] can be constructed in MPC(n^γ). We show this in the following theorem. Here we assume that the weights are polynomial in n, which is not unrealistic since in MPC the total memory is assumed to be $\tilde{O}(m)$ bits.

Theorem 2.4.4. *For any graph $G = (V, E, w)$ with n vertices, and parameters $\rho \leq \gamma \leq 1, 1 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa$ and $0 < \epsilon < 1$, there is an algorithm in MPC(n^γ) model that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{n^\rho}{\rho} \cdot \log^2 n \cdot \beta)$ rounds whp, where $\beta = O((\frac{\log n}{\epsilon} \cdot (\log \kappa + 1/\rho))^{\log \kappa + \frac{1}{\rho}})$.*

Proof. As explained in Section 2.3, the distributed implementation of this algorithm just performs multiple restricted Bellman-Ford algorithms in each phase. Recall also that it is enough to run each of the Bellman-Ford instances only for $O(\beta)$ rounds, by using the fact that for constructing hopset edges H_k for a distance scale of $(2^k, 2^{k+1}]$, the hopsets $\cup_{\log \beta - 1 < j \leq k-1} H_j$ can be used recursively.

Each round of a single Bellman-Ford algorithm can be simulated in $O(\frac{1}{\gamma})$ rounds of MPC(n^γ) by running the algorithm of Theorem 2.4.3 on each node, whose edges may be distributed over multiple machines. Hence each superclustering phase can be performed in $O(\frac{\beta}{\gamma})$ rounds. But at each interconnection phase multiple separate Bellman-Fords will run from each cluster center remaining. Thus we need to argue that these runs of Bellman-Ford will not violate the memory (and IO memory) limit of each machine. This can be shown using Lemma 2.2.1, which states that for each vetex $v \in V$, w.h.p. the number of explorations of interconnection phase that visit v is at most $O(\deg_i \cdot \log n)$. In other words, each node only forwards messages to at most $O(\deg_i \cdot \log n)$ in each depth $\delta_i/2$ Bellman-Ford explorations performed for an interconnection phase. Moreover, the parameters of their construction is set so that $\deg_i = O(n^\rho)$ throughout the algorithm. Hence, each node $v \in V$ need to store and forward distance estimates corresponding to at most $O(n^\rho \log n)$ sources for $O(\log(\kappa\rho) + \frac{1}{\rho})$ iterations, and each Bellman-Ford runs for $O(\beta)$ rounds. These separate Bellman-Ford runs can be pipelined. Overall, all of the Bellman-Ford explorations can be implemented in $O(\frac{\beta}{\gamma} \cdot n^\rho \log n)$. □

We can now construct a hopset first and then run the distributed variant of the algorithm in Section 2.2.2 due to [86] for constructing the distance sketches on the new graph. The sketch of a given node v can be stored at a machine in $M(v)$.

Proof of Theorem 2.1.2. After constructing a (β, ϵ)-hopset (by setting $\kappa = k$), we store the edges added to each node v by redistributing them among machines $M(v)$ that simulate v. Let $G' = (V, E \cup H, w')$ be the graph obtained by adding hopset edges. For constructing distance sketches with stretch $2k - 1$, we run the algorithm of [86] (described in Section 2.3) on G'. We run the restricted Bellman-Ford algorithm (Algorithm 2) in $O(\frac{\beta}{\gamma})$ rounds. Overall, $O(\frac{\beta n^\rho \log^2 n}{\rho\gamma})$ rounds are needed for the hopset construction (by Theorem 2.4.4), and $O(kn^{1/k} \log n \cdot \frac{\beta}{\gamma})$ rounds for building the distance sketches on G'. In case $k = O(1)$ we set $\rho = 1/\kappa$, and $\kappa = k$ to get $\beta = O(1)$ and total running time $\tilde{O}(n^{1/k})$. In case $k = \Theta(\log n)$, we will set $1/\kappa = \rho = \sqrt{\frac{\log\log n}{\log n}}$. □

2.4.1 Polylogarithmic Round Complexity

In this section we describe how we can modify our algorithm to run in a polylogarithmic number of rounds in exchange for increasing the stretch. We do this by first constructing a spanner, which sparsifies the graph (“shrinking" the input) and thus allows us to act as if we have “extra" total space. It turns out that this extra space is incredibly powerful, and will let us build distance sketches in polylogarithmic time. But in the end we have to pay for both the stretch of the spanner and the stretch of the sketch, so we only achieve stretch $O(k^2)$ rather than stretch $2k-1$ for sketches of size $\tilde{O}(n^{1/k})$.

There are intuitively two reasons why this extra space is so helpful. First, in MPC having extra space (or extra machines) is equivalent to having larger total communication bandwidth. This intuitively allows us to speed up the main construction algorithm by running the Bellman-Ford algorithms "in parallel". There are some technical details but it is not surprising that extra bandwidth is helpful.

The second reason why extra space is helpful is less obvious. Goodrich et al. [50] gave a powerful simulation argument, showing that PRAM algorithms can be efficiently simulated in MPC as long as the total number of processors used and the total space used by the PRAM algorithm are bounded by the size of the input. This is a very useful theorem, but the requirement that the number of processors is only the size of the input is very restrictive. For example, the state of the art PRAM algorithms for constructing hopsets use $\Omega(mn^{\rho})$ processors rather than $O(m)$ (for some value ρ determined by the parameters of the hopset). It turns out to be easy to extend [50] to show that if we have extra total space, we can use that extra space and communication to simulate PRAM algorithms that use slightly more processors or space. Thus by using a spanner first to sparsify the input, we give ourselves extra space and thus the ability to efficiently simulate a wider class of PRAM algorithms (hopsets in particular).

MPC with Extra Space. First we define a variant of MPC with extra machines (and thus extra space) denoted by MPC(S, S') where S is memory per machine, the number of machines is $\Theta(\frac{mS'}{S})$ and m is the total input size. This also implies the total memory available is $\Theta(mS')$ rather than $\Theta(m)$. We are first going to analyze our algorithm in this variant of MPC, and then switch back to the standard setting.

In [50] it was shown that with a small overhead PRAM algorithms can be simulated in MPC under certain assumptions on the number of processors and the memory used. We use a simple extension of their result for our new MPC variant.

Theorem 2.4.5. *Given a PRAM algorithm using $\mathcal{P} = O(m\alpha)$ processors that runs in time $\mathcal{T}$, and uses $O(m\alpha)$ total memory at any time, this algorithm can be simulated in $O(\mathcal{T}/\gamma)$ rounds of MPC(m^γ, α), for any $0 < \gamma < 1$.*

This stronger variant of MPC also lets us extend Theorem 2.4.1 for larger message sizes. We define a generalized variant of Find Minimum that takes a collection of vectors and computes their coordinate-wise minimum, and a generalizes version of Broadcast which broadcasts a vector of messages (rather than just a single message). We get the following lemma.

Lemma 2.4.6. *We can compute generalized Find Minimum(x, y) over N vectors of length α stored on a contiguous range of machines x to y in $O(1/\gamma)$ rounds of MPC(N^γ, α). Moreover, the generalized Broadcast$(\mathbf{b}, x, y)$ subroutine can also be implemented in $O(1/\gamma)$ rounds.*

Proof. In the new settings we have $\Theta(N^{1-\gamma} \cdot \alpha)$ machines that can be used for computation over N items in range (x, y), rather than $\Theta(N^{1-\gamma})$ machines used in Theorem 2.4.1. Therefore we can assign each coordinate to a group of $N^{1-\gamma}$ machines and then use a similar aggregation tree argument as in Theorem 2.4.1 on all the coordinates in parallel in $O(1/\gamma)$ rounds for both problems. □

Next, we describe how the algorithm of Theorem 2.3.3 can be modified to utilize the extra resources in MPC($n, n^{1/k}\log n$) to improve the round complexity. We use an argument similar to [86] with a few changes.

Theorem 2.4.7. *Given a graph $G = (V, E)$ with shortest path diameter Λ, there is an algorithm in MPC($n^\gamma, n^{1/k}\log n$) that runs in time $O(k\Lambda)$ w.h.p. and constructs Thorup-Zwick distance sketches of size $O(kn^{1/k}\log n)$ with stretch $2k - 1$.*

Proof. The algorithm is as follows: we have k *phases* for each level of Thorup-Zwick. Sampling sets $A_{k-1} \subseteq ... \subseteq A_1$ is straightforward. We start from the k-th phase, and we run Bellman-Ford (Algorithm 2) with the following modification: each node u keeps a vector of size $O(n^{1/k}\log n)$ of distance estimates $\tilde{d}(v,u)$ for all $v \in B(u)$. Then we run modified variants of the Broadcast and Find Min subroutines (Lemma 2.4.6) to update distances $\tilde{d}(v,u)$ based on a message received from a neighbor $u' \in N(u)$ if and only if $\tilde{d}(v,u) + w(u,u') < d(u, A_{i+1})$ and $\tilde{d}(v,u') + w(u,u') < \tilde{d}(v,u)$.

Based on Lemma 5 in [86], we know that at the end of phase i, each node $u \in V$ knows $B_i(u)$ and its distance to all nodes in $B_i(u)$. In particular, inductively each node u knows $d(u, A_{i+1})$ before starting phase i. Note that algorithm of [86] keeps a queue for all possible source nodes for their scheduling. We do not have space to store such a queue for all nodes. Here we simply only store a map of size $O(kn^{1/k}\log n)$ for each node u that corresponds to distance estimates for all $v \in B(u)$.

Next, we argue that each phase takes $O(\frac{\Lambda}{\gamma})$ rounds based on an inductive argument similar to Lemma 6 in [86]. Let $v \in B_i(u)$, and assume that there is a shortest path with j hops between v and u which we denote by $v = v_0, ..., v_j = u$. We use an induction on j. In the base case, u and v are neighbors and in $O(1/\gamma)$ rounds the aggregations can be performed as in Theorem 2.4.1. By inductive hypothesis v_{j-1} received a message after $O(\frac{(j-1)}{\gamma})$ rounds. If v_{j-1} had found its shortest path to v before the $(j-1)$-st iteration of Bellman-Ford, it would have already sent an update to v_j. Otherwise, v computes and broadcasts the updated distance using $O(1/\gamma)$ rounds of $\mathrm{MPC}(n^{\gamma}, n^{1/k}\log n)$. We showed in Lemma 2.4.6 that this can be done in parallel for all messages corresponding to sources in $B(u)$ in $O(1/\gamma)$ rounds. Hence v_j receives the updated distance after $O(j/\gamma)$ rounds, where $j \leq \Lambda$ by definition. Finally, all nodes will receive the distances from nodes in their bunches after $O(k\Lambda/\gamma)$ rounds. $\square$

A straightforward extension of Theorem 2.4.7 implies that given a (β, ϵ)-hopset for a graph, we can compute distance sketches with stretch $(1+\epsilon)(2k-1)$ in $O(\frac{\beta}{\gamma})$ rounds of $\mathrm{MPC}(n^{\gamma}, n^{1/k}\log n)$. Next, we show that in addition to proving Theorem 2.4.7, the extra memory also lets us improve the number of rounds for the hopset construction. To show this, we use a result in [38] that constructs hopsets in PRAM, which is as follows:

Theorem 2.4.8 ([38])**.** *For any graph $G = (V, E, w)$ with n vertices, and parameters $2 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa$ and $0 < \epsilon < 1$, there is a PRAM algorithm that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log n)$ in $O(\frac{1}{\rho} \cdot \log^2 n \cdot \log \kappa \cdot \beta)$ PRAM time whp, where $\beta = O(\frac{\log n(\log \kappa + 1/\rho)}{\epsilon})^{\log \kappa + \frac{1}{\rho}}$ using $\tilde{O}((m + n^{1+1/\kappa})n^{\rho})$ processors.*

We now argue that by having more space/machines, we are can implement the algorithm in Theorem 2.4.8 with the same guarantees in low-memory MPC settings. We will not discuss the details of the PRAM construction but the intuition here is similar to Theorem 2.4.7. At a high level, having more communication/memory will allows us to perform all the $\tilde{O}(n^{\rho})$ Bellman-Ford explorations required in the algorithm of Theorem 2.4.8 in parallel.

Corollary 2.4.9. *For any graph $G = (V, E, w)$, and parameters $0 < \epsilon < 1, 1/\kappa < \gamma \leq 1, \kappa \geq 2$, there is an algorithm that computes a (β, ϵ)-hopset with size $O(n^{1+\frac{1}{\kappa}} \log n)$ w.h.p. in $O((\kappa/\gamma) \cdot \log^2 n \cdot \log \kappa \cdot \beta)$ rounds of MPC($n^{\gamma}, n^{1/\kappa}$), where $\beta = O(\frac{\log n(\log \kappa)}{\epsilon})^{\log \kappa + \kappa + 1}$.*

Proof. The claim directly follows by setting $\rho = 1/\kappa$ in Theorem 2.4.8 and then applying the simulation in Theorem 2.4.5 in MPC($n^{\gamma}, n^{1/\kappa}$) on the new graph. □

Obtaining Extra Space. Our modified algorithm for MPC(n^{γ}) now proceeds as follows: we first construct a spanner, then construct a hopset on this spanner, and then use Theorem 2.4.7. Intuitively, by sparsifying the graph we can “buy" more memory and hence more communication. In other words, by building a spanner we can extend the results of the extra memory setting to the standard MPC setting.

There are several efficient PRAM algorithms for constructing spanners that we can simulate in MPC. We use an algorithm proposed by [9] that constructs a $(2k - 1)$-

spanner of size $O(kn^{1+1/k}\log n)$ with high probability. We then use Theorem 2.4.5 with $\alpha = 1$ (i.e. the original simulation of [50]) to construct the spanner in $O(\frac{k}{\gamma}\log n \log^* n)$ rounds of MPC(n^γ), and then redistribute the spanner edges (e.g., by sorting), to make the input distribution uniform over all the machines. We can now put everything together to get the polylogarithmic construction.

Proof of Theorem 2.1.3. We first construct a $4k-1$-spanner with size $O(kn^{1+\frac{1}{2k}})$. We denote this spanner by G'. Since G' has size $m' = O(n^{1+\frac{1}{2k}})$, while our total memory (and consequently overall communication bound) is still based on the original graph. Equivalently, the number of machines is $\frac{m}{n^\gamma} = \Omega(\frac{m' n^{1/2k}\log n}{n^\gamma})$ (since $m = \Omega(kn^{1+1/k}\log n)$), and therefore we are exactly in the MPC($n^\gamma, n^{\frac{1}{2k}}$) setting, but where the input graph is G'. Then we use Corollary 2.4.9 to construct a (β,ϵ)-hopset for G' with $\beta = O(\frac{k}{\gamma}\cdot(\frac{\log n\cdot \log k}{\epsilon})^{\log k+1+k})$ rounds of MPC(n^γ). Finally, after adding the hopset edges to G' we use Theorem 2.4.7. The new stretch is clearly $O(k^2(1+\epsilon))$. □

2.4.2 Single-source shortest path

In various models (such as PRAM, CONGEST and Congested Clique) hopsets are used for solving shortest path problems (e.g. [25, 56, 38]), and thus it is natural to see how they can be used for this application in the MPC model. In particular, we discuss application of Theorem 2.4.4 in solving the (approximate) single-source shortest path problem. As stated earlier, while this problem is well-studied in many distributed models, including the Congested Clique model, we are not aware of any non-trivial results for this problem in the low memory MPC setting.

Theorem 2.4.10. *Given a weighted undirected graph $G = (V, E, w)$, a source node $s \in V$, and $0 < \gamma \leq 1, 0 < \epsilon < 1$ we can compute $(1+\epsilon)$-approximate distances from s to all nodes in V w.h.p. in $O(\frac{1}{\gamma})\cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC with $\Theta(n^\gamma)$ memory per machine.*

Proof. We first construct a hopset using Theorem 2.4.4 by setting $\rho = \sqrt{\frac{\log n}{\log\log n}}$, and $\kappa = \Theta(\log n)$. This will let us build a hopset with hopbound $2^{\tilde{O}(\log n)}$ in time $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\log n)}$. We then run the restricted Bellman-Ford algorithm (Algorithm 2) in $O(\frac{1}{\gamma}) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ rounds of MPC(n^γ). The idea behind this choice of parameters is the following: any attempt to improve the running time by getting a smaller hopbound (e.g. constant) will increase the time required to construct the hopset. In other words, this choice of parameters will make the time required for preprocessing (construction of the hopset) almost the same as the time required for running the Bellman-Ford algorithm. □

Finally, we show that we can used the technique in Section 2.4.1 to find constant approximation to single source shortest path in polylogarithmic time for graphs with a certain density. In particular, by first constructing a spanner and then using Corollary 2.4.9, we can also solve $4k(1+\epsilon)$-approximate SSSP (for any $2 \leq k \leq O(\log n)$) on any graph with $m = \Omega(n^{1+1/k} \log n)$ edges in fewer number of rounds. After constructing a $4k-1$-spanner, we construct a (β, ϵ)-hopset for an appropriate hopbound β using the extra space and then run a single restricted Bellman-Ford (Algorithm 2) from the source in $O(\beta/\gamma)$ rounds of MPC(n^γ). By setting $\kappa = k$ we get,

Corollary 2.4.11. *For any graph $G = (V, E, w)$ with n vertices, $m = \Omega(n^{1+1/k})$ edges, and $0 < \epsilon < 1, 1/k < \gamma \leq 1, k > 2$, and a source node $s \in V$, there is an algorithm that w.h.p. finds a $4k(1+\epsilon)$-approximation of shortest path distance from s to all nodes in $O(\frac{1}{\gamma} \cdot (\frac{\log n \cdot \log k}{\epsilon})^{\log k + k + 1})$ rounds of MPC(n^γ). In particular, for $k = O(1)$ the algorithm runs in $O(\frac{1}{\gamma} \cdot (\frac{\log n}{\epsilon})^{O(1)})$ rounds.*

2.5 Distance Oracles in the Streaming Model

In this section we will describe how the Thorup-Zwick distance oracles can be constructed in the insert-only streaming model. For graph problems, the stream is a sequence of edges (and their weights), and the goal is to solve the problem in space strictly sublinear in number of edges. For some problems we might need to see multiple *passes* of the stream. Similar to the distributed settings, we will use the hopset construction of [38]. They show that in streaming settings a (β, ϵ)-hopset, where with the following guarantees can be constructed.

Theorem 2.5.1 ([38]). *For any graph $G = (V, E, w)$ with n vertices, and any $2 \leq \kappa \leq (\log n)/4, 1/2 > \rho \geq 1/\kappa, 1 \leq t \leq \log n$ and $0 < \epsilon < 1/2$, there is a streaming algorithm that computes a (β, ϵ)-hopset with expected size $O(n^{1+\frac{1}{\kappa}} \log^2 n)$, where $\beta = O(\frac{1}{\epsilon} \cdot (\log(\kappa) + \frac{1}{\rho}) \log n))^{\log(\kappa)+\frac{1}{\rho}}$ requiring either of the following resources:*

- *$O(\beta \log n)$ passes w.h.p. and expected space $O(\frac{n^{1+\rho}}{\rho} + n^{1+\frac{1}{\kappa}} \log^2 n)$,*
- *$O(n^{\rho} \cdot \beta \cdot \log^2 n)$ passes w.h.p. and expected space $O(n^{1+\frac{1}{\kappa}} \log^2 n)$.*

We will next explain how the distance oracle can be constructed in $O(\beta)$ passes given a hopset with hopbound β. First, we need a variant of *restricted* Bellman-Ford for streaming settings. The idea of using Bellman-Ford in streaming settings has been previously used for shortest path computation (e.g. [38], [56]). This algorithm is similar to the distributed variant: on receipt of each edge $(u, v) \in E$ we will check to see if the distance from any of the sources in S should be updated. After i passes of the algorithm, all nodes have the i-restricted distance to nodes in s. The restricted Bellman-Ford in streaming is presented in Algorithm 3. Note that unlike centralized Bellman-Ford nodes do not store an initial distance estimate (due to space limitation in the streaming model). This algorithm uses $O(|S| \cdot nh)$ total space.

After constructing a hopset, using the restricted Bellman-Ford algorithm, we can construct a Thorup-Zwick distance oracle of stretch $(2k - 1)(1 + \epsilon)$ in $O(\beta)$ passes.

First, we will explain how distance oracles can be constructed in $O(\Lambda)$ passes, where Λ is the shortest path diameter. The details of this algorithm is presented in Algorithm

Algorithm 3: Restricted Bellman-Ford in the Streaming Model

```
Input  : Undirected weighted graph $G = (V, E, w)$, and source node $s \in V$.
Output : $h$-hop restricted distances from the source $s$ to all nodes $u \in V$,
         $d^{(h)}(s, v)$.
1 for $O(h)$ passes do
2     for $(u, v) \in E$ do
3         if $\hat{d}(s, v) = \emptyset$ or $\hat{d}(s, u) + w(v, u) < \hat{d}(s, v)$ then
4             $\hat{d}(s, v) = \hat{d}(s, u) + w(v, u)$
5         if $\hat{d}(s, v) = \emptyset$ or $\hat{d}(s, v) + w(v, u) < \hat{d}(s, u)$ then
6             $\hat{d}(s, u) = \hat{d}(s, v) + w(v, u)$
```

4. This algorithm is again similar to the distributed algorithm. Sets $A_1, .., A_{k-1}$ can easily be sampled in sublinear space. Here again for finding the distances from each set A_i to all nodes, we will add a virtual node a_i and add an edge of weight 0 between a_i and all the nodes in A_i. We then run the restricted Bellman-Ford algorithm from each of these sources a_i separately. This phase requires $O(kn\Lambda)$ space and $O(\Lambda)$ passes. In the final phase, we need to find the distance from each node in $s \in A_i \setminus A_{i+1}$ to all nodes in $C(s) = \{v \mid v \in B(s)\}$. We will run a variant of the Bellman-Ford algorithm in which each node v only stores a distance only if $\hat{d}(s, v) < d(v, A_{i+1})$ or if this condition holds after receiving an update from a neighbor. We will get the following lemma.

Lemma 2.5.2. *There is an algorithm that runs in $O(\Lambda)$ passes, and w.h.p. constructs a $2k - 1$ stretch Thorup-Zwick distance oracle of size $O(kn^{1+1/k} \log n)$ using $O(kn^{1+1/k} \log n)$ total space.*

Proof. It is clear that described algorithm takes $O(\Lambda)$ passes, and correctly updates all the distances required for building a Thorup-Zwick distance oracle. We also show that the space required is the same as the distance oracle size. This follows from the fact that for each node v, we are only storing distances to the nodes that are in v's bunch $B(v)$, and we know from [88] that w.h.p. $|B(v)| = O(kn^{1/k} \log n)$. Thus the

Algorithm 4: Preprocessing distance oracle of stretch $2k-1$ in the streaming model.

```
Input  : Undirected graph $G = (V, E, w)$ of shortest path diameter $\Lambda$.
Output : Approximate distance oracle.
1  Set $A_0 = V, A_k = \emptyset$.
2  for $i = 1$ to $k - 1$ do
3      If $v \in A_{i-1}$ with probability $n^{-1/k}$ add $v$ to $A_i$.
4  Run Algorithm 3 in parallel out of each set $A_i, 1 \leq i \leq k$, to find
     $p_i(v) = \text{argmin}_{u \in A_i} d(u, v)$, and set $d(v, A_i) := d(p_i(v), v)$.
5  for $O(\Lambda)$ passes do
6      for $(u, v) \in E$ do
7          for $i = k - 1$ down to 1 do
8              for $s \in A_i \setminus A_{i+1}$ do
9                  if $\hat{d}(s, v) < d(v, A_{i+1})$ or $\hat{d}(s, u) + w(u, v) < d(v, A_{i+1})$ then
10                     if $\hat{d}(s, u) + w(v, u) < \hat{d}(s, v)$ then
11                         $\hat{d}(v, s) = \hat{d}(s, u) + w(v, u)$
12                 if $\hat{d}(s, u) < d(u, A_{i+1})$ or $\hat{d}(s, v) + w(u, v) < d(u, A_{i+1})$ then
13                     if $\hat{d}(s, u) + w(v, u) < \hat{d}(s, u)$ then
14                         $\hat{d}(s, u) = \hat{d}(s, v) + w(v, u)$
```

total space is w.h.p. $O(kn^{1+1/k} \log n)$. □

Similar to the distributed case, given a graph $G = (V, E, w)$, we can use the (β, ϵ)-hopset construction of Theorem 2.5.1 to obtain a graph $G' = (V, E \cup H, w')$ which has shortest path diameter $O(\beta)$, and the distances in G are preserved up to a factor of $(1 + \epsilon)$. Then by running Algorithm 4 on G', we would require $O(\beta)$ passes to build a distance oracle with stretch $(2k - 1)(1 + \epsilon)$. We set the parameters in such a way that we have space to store all the hopset edges locally. Thus while running the algorithm of Theorem 2.5.3 we also consider the hopset edges to decide when to update the distance. However, for readability of our algorithm, here we assume that the hopset edges are also appearing in the stream. Hence, by first running the hopset construction algorithm of Theorem 2.5.1, and then running the algorithm of Theorem 2.5.3, we will get the following result:

Theorem 2.5.3. *Given a graph $G = (V, E, w)$, there exists a streaming algorithm that constructs a Thorup-Zwick distance oracle of stretch $(2k-1)(1+\epsilon)$ of size $O(kn^{1/k} \log n)$ w.h.p. using either of the following resources*[5]*:*

- *$O(\beta \log n)$ passes w.h.p. and expected space $O(\frac{n^{1+\rho}}{\rho} + n^{1+\frac{1}{k}} \log^2 n)$,*
- *$O(n^\rho \cdot \beta \cdot \log^2 n)$ passes w.h.p. and expected space $O(n^{1+\frac{1}{k}} \log^2 n)$,*

where $\beta = O(\frac{1}{\epsilon} \cdot (\log(k) + 1/\rho) \log n))^{\log(k)+\frac{1}{\rho}}$ and $\frac{1}{k} \le \rho \le \frac{1}{2}$.

In particular, when $k = O(1)$ we will use the first case of Theorem 2.5.3 and set $\rho = 1/k$, and when $k = \Omega(\log n)$ we will use the second case and set $\rho = \sqrt{\frac{\log\log n}{\log n}}$. We have,

Corollary 2.5.4. *Given a graph $G = (V, E, w)$, there exists a streaming algorithm that constructs a Thorup-Zwick distance oracle of stretch $(2k-1)(1+\epsilon)$ of size $O(kn^{1/k} \log n)$ w.h.p. and expected space $O(n^{1+1/k} \cdot \log^2 n)$, such that:*

- *If $k = O(1)$, we require $O(\log^k n)$ passes with high probability.*
- *If $k = \Omega(\log n)$, we require $2^{\tilde{O}(\sqrt{\log n})} = n^{o(1)}$ passes with high probability.*

2.6 Comparison with Alternative Methods

In this section we discuss how storing a distance oracle is compared with alternative methods for distance computation in distributed settings. In the Congested Clique, rather than computing a Thorup-Zwick distance oracle we could instead compute a graph spanner and store this at the coordinator node. A $(2k-1)$-spanner of G is simply a subgraph which preserves distances up to a $(2k-1)$ factor, so once such spanner is at the coordinator, a classical centralized shortest-path algorithm would

[5]All the bounds expressed in expectation can be turned into high probability bound with an additional factor of $\log n$ in the number of passes.

yield a distance estimate that is accurate up to $(2k-1)$ (as with our distance oracle). A similar approach can be used in the streaming model. While a reasonable approach, there are a few drawbacks.

First, the local computation time becomes superlinear, rather than $O(k)$ as in our oracle. While computation is generally extremely cheap compared to network communication, there is still an enormous gap between superlinear and $O(k)$ (since k is at most logarithmic). And for large graphs, this may indeed rise to the level of network delay time scales.

The more important drawback, though, is that spanners cannot be used in the MPC model. Even an extraordinarily sparse spanner would not fit into the memory of a single server in low-memory MPC, so the spanner would (just like the original graph) have to be stored in a distributed fashion. So we would still have the same problem that we started with: how to compute distance estimates in a distributed graph. Only distance sketches allow us to answer such queries in such a small number of rounds (in particular, two rounds after the sketches have been computed).

Another direction that one could take is running an all-pairs shortest path algorithm (APSP). This approach has multiple drawbacks: First, for fast queries we will need to store the whole adjacency matrix, which clearly uses much more space. Secondly, such algorithms are slower and use more resources.

2.7 Discussion and Future Work

Fist we note that our results in this section can be extended in two directions: After constructing a (β, ϵ)-hopset in MPC, we can use it to compute a $(1+\epsilon)$-approximate multi-source-shortest path from s-sources in $O(\beta)$ rounds, and with an extra factor of s in total memory consumption. Depending on the number of sources and desired total memory we can find the appropriate β. For instance, if the number of sources is large, we can use a total of $O(sm)$ memory and set β to be polylogarithmic.

Another application of our techniques is in constructing distance oracles also in PRAM, and dynamic streams with some additional technicalities within the same bounds given here for MPC and streaming respectively.

One future direction is in computing k-approximate shortest paths using hopsets with larger stretch directly. One would hope that such hopsets can have smaller hopbound, and lead to better tradeoffs.

In a recent related work [17], we obtain a $\tilde{O}(\log\log k)$ round algorithm for constructing $O(k)$-spanners in low and linear memory MPC. This leads to a $\log n^{1+o(1)}$-approximate all-pairs-shortest-path algorithm in $\tilde{O}(\log\log n)$-rounds of *linear memory MPC*. While this algorithm is exponentially faster than other known results, the approximation ratio is larger than desired. One natural open problem is to see whether hopsets could lead to better tradeoffs in the linear memory MPC regime.

Chapter 3

Sparse Hopsets in Congested Clique

3.1 Introduction

In this chapter we propose a new algorithm for constructing sparse hopsets in the Congested Clique model. Our construction time improves over the previous state-of-the-art sparse hopsets of Elkin and Neiman [38](used in Chapter 2), and [41] in this model. The results in this chapter are published in [77]. Recall the definition of a hopset. Informally, given a graph $G = (V, E)$, a (β, ϵ)-hopset H with hopbound β, is a set of edges such that for any pair of nodes u and v in G, there is a path with at most β hops in $G \cup H$ with length within $(1 + \epsilon)$ of the shortest path between u and v in G (see the more formal definition in Chapter 1). We generally want to have sparse hopsets with small hopbound. As we discussed in Chapter 1 hopsets are theoretically interesting, and are connected to many other important objects, but in this chapter the following perspective on their application is helpful: In the distributed models that we consider, once a hopset is preprocessed, we can use it for computing distance queries, and the query time (number of distributed rounds) will be the hopbound.

There is a natural tradeoff between the size and the hopbound (or the distributed query time) of a hopset. In an extreme case one could store the complete adjacency list-or equivalently add $O(n^2)$ edges, and then query distances in constant time. Other than the fact that computing all-pairs shortest-path is generally slow, we often do not have enough space to store the whole adjacency list for large-scale graphs. There is a line of work that focuses on designing data structures with small size, say $\tilde{O}(n^{1+1/k})$, in which distances can be estimated up to $O(k)$ stretch in small query time. Examples of such structures are Thorup-Zwick distance oracles [88], which we focused on in Chapter 2. Hopsets offer a different tradeoff: a hopset gives an accuracy of $1+\epsilon$ (rather than $O(k)$) at the expense of a larger query time (polylogarithmic in distributed settings instead of a small constant in centralized settings). It is therefore crucial to keep the hopbound as small as possible, since the hopbound will basically determine the query time and is more important than the preprocessing time. However, even in centralized settings there are existential limitations in this tradeoff. The lower bound result by [1] states that there are graphs for which we can not have a hopbound of $o(\log(k)/\epsilon)^{\log(k)}$ and size $O(n^{1+1/k})$, for arbitrary $0 < \epsilon < 1$.

In a recent result, Censor-hillel et al. [19] gave a fast Congested Clique algorithm that constructs a hopset with hopbound $O(\log^2(n)/\epsilon)$ and size $\tilde{O}(n^{3/2})$. While we can use their hopsets to compute distances efficiently, one shortcoming of such a construction is the large space. In particular, if the original graph has size $o(n^{3/2})$, their algorithm requires storing more edges than the initial input. This is undesirable for large scale graphs, where we do not have any additional space available. It is therefore natural to find algorithms that use less space, possibly in exchange for a slightly weaker hopbound (but still polylogarithmic). This is our main goal in this chapter. We extend the result of [19] by constructing sparse hopsets with size $\tilde{O}(n^{1+1/k})$ for a constant $k \geq 2$ and polylogarithmic hopbound in polylogarithmic time in Congested Clique. This is the first Congested Clique construction of sparse hopsets with polylogarithmic hopbound that uses only *polylogarithmic* number of rounds. Hence we can store a sparse auxiliary data structure that can be used later to query distances (from multiple sources) in polylogarithmic time.

Our hopset construction is based on a combination of techniques used in Cohen [25] (with some modifications) and the centralized construction of Huang and Pettie [58]. We also use another result of [19] that allows us to efficiently compute $(1+\epsilon)$-approximate multi-source shortest path distances from $O(\sqrt{n})$ sources.

One tool that we use in our construction is a *hop-limited neighborhood cover* construction, which may be of independent interest. Roughly speaking, a W-neighborhood cover is a collection of clusters, such that there is a cluster that contains the neighborhood of radius W around each node, and such that each node is contained in at most $O(\log(n))$ clusters. In an ℓ-limited W-neighborhood cover only balls with radius W using paths with at most ℓ-hops are contained in a cluster.

Connections with other chapters. One technique that we use in our construction is a scaling idea by [64] that allows us to construct neighborhood covers more efficiently. Interestingly, we also use this idea in our dynamic hopset construction in Chapter 4. While this idea is used in a different primitive and model in these two places, in both cases it is based on the intuition that by restricting our attention to pairs of nodes in a specific distance scale, scaling based ideas can improve the performance of these algorithms.

We also note that our neighborhood cover algorithm is based on constructing *padded decompositions*, a tool that we also use in Chapter 5 (see Section 5.3), for a very different purpose of solving linear programs in the LOCAL model.

3.1.1 Our contribution

While our main focus on hopsets in the Congested Clique model, we also get several side results in other models. In this section, we summarize our results and their connection with related work.

Congested Clique. Before this work, the state-of-the-art construction of sparse hopsets in Congested Clique was the results of Elkin and Neiman [41] (and similar bounds in [38]), but these algorithms require polynomial number of rounds for constructing hopsets with polylogarithmic hopbound. The construction of Censor-Hillel et al. [19] is a special case of hopsets of [41]. They construct hopsets of size $\tilde{O}(n^{3/2})$ with $O(\log^2(n)/\epsilon)$ hopbound. They can construct such a hopset in $O(\log^2(n)/\epsilon)$ rounds of Congested Clique using sparse matrix multiplication techniques. However, [19] does not provide any algorithm for sparser hopsets. Here we use a new hopset construction that has a very different structure than hopsets of [41] and with improved guarantees. Not only does our hopset construction run in polylogarithmic rounds, but it also yields a better size and hopbound tradeoff over the state-of-the-art Congested Clique construction of [41]. Prior to [38] and [19] the hopsets proposed for Congested Clique had superpolylogarithmic hopbound of $2^{\tilde{O}(\sqrt{\log(n)})}$ [56] or polynomial [76] hopbound. More formally, we provide an algorithm with the following guarantees:

Theorem 3.1.1. *Given a weighted*[1] *graph* $G = (V, E, w)$*, for any* $k \geq 2, 0 < \epsilon \leq 1$*, there is a Congested Clique algorithm that computes a* (β, ϵ)*-hopset of size* $O(n^{1+\frac{1}{2k}} \log(n) + n \log^2(n))$ *with hopbound* $\beta = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1})$ *with high probability in* $O(\beta \log^2(n))$ *rounds.*

To see how this result compares to the *efficient variant* of the Congested Clique hopsets of [38], we note that for a hopset of size $O(n^{1+\frac{1}{k}} \log(n))$, we have a hopbound of $O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-2})$, whereas [38] gets a hopbound of $\Omega((\frac{\log(n)\log(k)}{\epsilon})^{\log(k)+2})$. Thus we get a $\Theta(\log^{2-o(1)}(n)(\frac{\log(k)}{\epsilon})^{4-o(1)})$ factor improvement over the construction of [38]. Also, their more efficient algorithm runs in $\tilde{O}(n^{\rho})$ rounds of Congested Clique, where $0 < \rho \leq \frac{1}{2}$ is a parameter that impacts the hopbound (ρ is a constant when their hopbound is polylogarithmic). They also propose a second algorithm that uses

[1]For simplicity, we assume that the weights are polynomial. This assumption can be relaxed using standard reductions that introduce extra polylogarithmic factors in time (and hopbound) (e.g. see [64],[74], [38]).

an extra polynomial factor in the running time to obtain constant hopbound[2]. We note that the construction of [41] has similar guarantees to [38], with two differences: it eliminates a $\log(n)$ factor (or more generally the dependence on aspect ratio) in the hopset size, but has a slightly worse hopbound in their fastest regime.

Our construction is mainly based on the ideas of [25] with a few key differences that take advantage of the power of Congested Clique. While the hopsets of [38] improve over hopsets of [25] existentially in centralized settings, the construction of [25] has certain properties that makes it adaptable for a better Congested Clique algorithm. In particular, [25] uses a notion of small and big clusters, and we can utilize this separation in Congested Clique. We change the algorithm of [25], in such a way that leads to adding fewer edges for small clusters. This leads to sparser hopsets and improves the overall round complexity. The key idea is that by using the right parameter settings, in Congested Clique we can send the whole topology of a small cluster to a single node, *the cluster center*, and then compute the best known hopset locally. It is possible to perform these operations specifically in Congested Clique due to a well-known routing algorithm by Lenzen [68]. We can then combine Theorem 3.1.1 with a *source detection* algorithm by [19] (formally stated in Lemma 3.2.2) to get the following result for computing multiple-source shortest path queries.

Corollary 3.1.2. *Given a weighted graph $G = (V, E, w)$ there is a Congested Clique algorithm that constructs a data structure of size $O(m+n^{1+\frac{1}{2k}})$ in $O(\beta \log^2(n))$ rounds, where $\beta = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1})$, such that after construction we can query $(1+\epsilon)$-stretch distances from $O(\sqrt{n})$ sources to all nodes in V in $O(\beta)$ rounds with high probability.*

[2]If we allow extra polynomial factors in the running time we may also get a constant hopbound (we would need to change the parameters of the neighborhood cover construction, and change how we iteratively use smaller scales). However, we do not study this regime, as it is not aligned with our main motivation of getting polylogarithmic round complexity.

Neighborhood and Pairwise Covers. In Section 3.3, we focus on an efficient construction of a *limited* pairwise cover (or neighborhood cover) in the CONGEST model, which is a tool that we use in our hopset construction. Given a weighted graph $G = (V, E)$, a W-pairwise cover, as defined by [24], is a collection $\mathcal{C}$ of subsets of V with the following properties.

1. The diameter of each cluster is $O(W \log n)$;
2. We have $\sum_{C \in \mathcal{C}} |C| = O(n)$, $\sum_{C \in \mathcal{C}} E(C) = \tilde{O}(m)$, in other words, the sum of the sizes of all clusters is $O(n)$, and the sum of all edge occurrences in the clusters is $\tilde{O}(m)$;
3. For every path p with (weighted) length at most W, there exists a cluster C where $p \subseteq C$.

Pairwise covers are similar to neighborhood covers of Awerbuch and Peleg [4] with two differences: in a W-neighborhood cover, there must be a cluster that contains the neighborhood of radius W around each node rather than only paths of length W. Neighborhood covers also need an additional property that *each node* is in at most $O(\log(n))$ clusters. While for our purposes the path covering property is enough, in distributed settings we need the property that each node overlaps with few clusters to ensure that there is no congestion bottleneck. The main subtlety in constructing a general W-pairwise (or neighborhood) cover is that we may need to explore paths of $\Omega(n)$ hops, and thus it is not clear how this can be done in polylogarithmic time. To resolve this, [24] proposed a relaxed construction called *ℓ-limited W-pairwise cover*. This structure has all the above properties but only for paths with at most ℓ-hops. More formally, the third property will be relaxed to require that for every path p of weight at most W with at most ℓ hops there exists a cluster C where $p \subseteq C$. We can define an ℓ-limited W-neighborhood cover similarly.

A randomized algorithm that constructs ℓ-limited pairwise covers with high probability in $O(\ell)$ depth in the PRAM model was given by [24]. The ideas used in [74] for constructing work-efficient PRAM hopsets, can also be used to construct ℓ-limited pairwise and neighborhood covers in PRAM. However, they do not explicitly construct limited pairwise covers. In distributed settings, a recent construction for *sparse* neighborhood covers in *unweighted graphs* in the CONGEST model was given by [79]. However, even by generalizing their result to weighted graphs, in order to cover distances for large values of W the algorithm would take $\tilde{\Omega}(W)$ rounds for reasons described above.

To the best of our knowledge, an efficient algorithm for constructing ℓ-limited pairwise-covers (or limited neighborhood covers) is not directly studied in the CONGEST and Congested Clique literature. Our first contribution is such an algorithm: we use the low-diameter decomposition construction of Miller et al. [74] for *weighted graphs*, combined with a rounding technique due to [64] to construct ℓ-limited W-pairwise covers in $O(\ell \log^2(n))$ rounds in the CONGEST model. Importantly, ℓ is a parameter independent of W, which we will set to a polylogarithmic value throughout our hopset construction. Our algorithm is similar to the algorithm of [74], but with some adaptations needed for implementation in the CONGEST model. Formally, we get the following result:

Theorem 3.1.3. *Given a weighted graph $G = (V, E, w)$, there is an algorithm that constructs an ℓ-limited W-pairwise cover in $O(\ell \log^2(n) \log(W))$ rounds in the CONGEST model, with high probability. Moreover, a pairwise cover for paths with ℓ-hops with length in $[W, 2W]$ [3] can be constructed in $O(\ell \log^2(n))$ rounds with high probability.*

[3]The algorithm and analysis can easily be extended to paths with length $[W, cW]$ for any constant c.

MPC. As a side result[4], we note that pairwise covers can also be constructed efficiently in the Massively Parallel Computation model (MPC) (even when memory per machine is strictly sublinear). This in turn leads to a better running time for $(1+\epsilon)$-MSSP from $O(\sqrt{n})$ sources (and consequently SSSP), in $O(\log^2(n)/\epsilon)$ rounds in a variant of the model where we assume the overall memory of $\tilde{O}(m\sqrt{n})$ (equivalently, we have *more machines* than in the standard MPC model). We consider this variant since in practice it is plausible that there are more machines, while due to the large-scale nature of data in these settings, using less memory per machine is often more crucial.

In Chapter 2 (which is based on [30]) we constructed a (β, ϵ)-hopsets (based on hopsets of [38]) with polylogarithmic hopbound when the overall memory is $\tilde{O}(m)$, but we argued that using the existing hopset constructions, this would take polynomial number of rounds in MPC. We further showed that if the overall memory is by a polynomial factor larger (i.e. if the overall memory is $\tilde{\Theta}(mn^{\rho})$ for a constant $0 < \rho \leq 1/2$), then hopsets with polylogarithmic hopbound can be constructed in polylogarithmic time. We can also use this extra-memory idea, first to argue that using the hopsets of [25] instead of the hopsets of [38] we can get a smaller hopbound when the overall memory is $O(m\sqrt{n})$. Then we observe that if we use a faster ℓ-limited pairwise cover algorithm (based on the construction of [74]) instead of the pairwise covers that [25] uses, we can shave off a polylogarithmic factor in the construction time. This ℓ-limited W-pairwise cover construction may also be of independent interest in MPC. More formally, we get faster algorithms for $(1+\epsilon)$-MSSP:

Theorem 3.1.4. *Given an undirected weighted graph G, we can compute $(1+\epsilon)$-MSSP from $O(n^{1/2})$ sources in $O(\frac{\log^2(n)}{\gamma\epsilon})$ rounds of MPC, when memory per machine is $\tilde{O}(n^{\gamma}), 0 < \gamma \leq 1$ and the overall memory is $\tilde{O}(mn^{1/2})$ (i.e. there are $\Theta(mn^{1/2-\gamma})$*

[4]Our MPC results can be seen as a straight-forward combination of results of [25], [74], [30] (Chapter 2) and simulation of [50]. But since both the construction and the model are closely relevant to our Congested Clique algorithms, we find it useful to include this discussion.

machines).

The difference between this result and that of Chapter 2 [30] is that they give a more general result where the overall memory is $\tilde{O}(mn^{\rho})$ for a parameter $\rho > 0$. But in the special case of $\rho = 1/2$, we get a hopbound of $O(\frac{\log^2(n)}{\epsilon})$, whereas in this case they get a hopbound of $O((\frac{\log(n)}{\epsilon})^3)$. We also note that the main focus in Chapter 2 is constructing Thorup-Zwick distance sketches. As explained earlier, these structures offer a different tradeoff: much weaker accuracy ($O(k)$-stretch), but better query time (constant rounds rather than polylogarithmic) and less space after preprocessing ($\tilde{O}(n^{1+1/k})$ instead $\tilde{O}(m)$ in the case of hopsets). More details on the MPC algorithm can be found in Section 3.5.

Further related work on distributed distance computation. In a related result, the problem of single-source shortest path (SSSP) in Congested Clique was also studied in [11], where they use continuous optimization techniques for solving transshipment. Firstly, their algorithm takes a large polylogarithmic round complexity, and has a high dependence on ϵ. But we can have a significantly smaller running time depending on the hopset size. In other words, for hopsets with a reasonable density (e.g. with size $n^{1+\mu}$, where $\mu < 0.1$) we get a much smaller polylogarithmic factor for computing $(1 + \epsilon)$-SSSP. This can be further reduced if we allow denser hopsets.

More importantly, an approach such as [11] is mainly suited for SSSP. One limitation with their approach is that for computing multiple distance queries we need to repeat the algorithm for each query. For example, for computing the shortest path from s sources to all nodes, we have to repeat the whole algorithm s times. But constructing a hopset will let us run many such queries in parallel in $O(\beta + s)$ rounds, where β is the hopbound. Moreover, we can compute multi-source shortest path from $O(\sqrt{n})$ sources in parallel for all the sources using the source detection algorithm of [19].

Subsequent work. After this work [77] was published, [33] proposed a Congested Clique algorithm for *unweighted graphs* that runs in $\tilde{O}(\log\log(n))$ rounds and computes $(2+\epsilon)$-approximate all-pairs-shortest-paths.

In this work, we are not taking full advantage of sparse hopsets since we use the result of [19] as a black-box. A natural direction is to combine the sparse matrix multiplication techniques of [19] with our sparse hopset algorithm. This may lead to an efficient approximate multi-source shortest path algorithm for a larger set of sources, and further results for approximate all-pairs-shortest paths. Indeed, in a very recent work, Elkin and Neiman [42] used sparse matrix multiplication, combined with a different hopset construction to obtain a $(1+\epsilon)$-MSSP algorithm from up to roughly $n^{0.65}$ sources in polylogarithmic rounds of Congested Clique. We leave it as an open problem whether our hopset techniques allows a further increase in the number of sources.

3.1.2 Overview of techniques.

Our hopset has a similar structure to hopsets of [25], but with some changes both in construction and the analysis. We also take advantage of multiple primitives that are specific to Congested Clique such as message routing algorithm by Lenzen [68] and a recent result of [19]. First, we explain the ℓ-limited W-neighborhood cover construction and then we explain the hopset algorithm.

ℓ-limited neighborhood covers. As described earlier, our algorithm for constructing a W-neighborhood cover is based on a combination of the low-diameter decomposition of [74], and a rounding technique originally proposed by [64]. At a high level, in the low-diameter decomposition algorithm of [75, 74], each node u chooses a radius r_u based on an exponential random variable. Then each node u joins the cluster of node v that minimizes *the shifted distance* from u, which is defined as

$d(u, v) - r_v$. This leads to a partition of the graph, and we can show that by repeating this process we will get a W-neighborhood cover. Since partitions for constructing a W-neighborhood covers directly will be slow for large values of W, we focus on the ℓ-limited W-pairwise covers. To construct these, consider all pairs of nodes within distance $[w, 2w], w \leq W$ in each iteration. We round up the weights of each edge in the graph based on values w and ℓ. We then construct a low-diameter-decomposition *based on the new weights*, such that the diameter of each cluster is $O(\ell \log(n))$ (rather than $O(W \log(n))$ based on the original weights). The rounding scheme is such that the ℓ-limited paths with length $[w, 2w]$ in the original graph will be explored. Intuitively, this means that on the rounded graph we need to explore a neighborhood with fewer hops, which will lead to a faster construction. We can then repeat this process for $O(\log(W))$ times for different distance intervals. The details of this rounding scheme can be found in Section 3.3.

Hopset Construction. First we describe the sequential hopset construction and will then choose the parameters appropriately for our distributed construction. Let μ be a a parameter that we will set later. The (sequential) structure of the hopset is as follows: In each iteration we consider pair of nodes $u, v \in V$ such that $R \leq d(u, v) < 2R$, and we call the interval $[R, 2R)$ a *distance scale*. Then for distance scales $[R, 2R)$ we set $W = O(\epsilon R/(\log n))$ and construct a W-pairwise cover. We let big clusters be the clusters that have size at least n^{μ} and small clusters have size less than n^{μ}, where $0 < \mu < 1$ is a constant parameter. Then we construct a hopset with small hopbound on each of the small clusters. This is the main structural difference with the construction of [25] that adds a clique for the smallest hopsets. We then add a star from the center node of each big cluster to every other node in that cluster, and add a complete graph at the center of large clusters. Whenever we add an edge, we set the weight to be the distance between the two endpoints. In the distributed construction, the weight will be an estimate of this distance that we will describe later.

Roughly speaking, constructing a hopset on small clusters rather than constructing a clique as [25] does, allows us to set the size threshold of small clusters larger, while keeping the number of edges added small. This in turn reduces the number of big cluster centers we have to deal with. Such a modification can be very well tuned to the Congested Clique model. By setting $\mu = 1/2$, we will have small cluster that will at most $O(n)$ edges. Then a well-known routing algorithm by Lenzen [68] can be used to send all these edges to the cluster center. The cluster center can then compute a hopset locally. For this we use current best-known centralized construction by [58]. The other challenge is that we need to compute pairwise distances between all big cluster centers. In [25] this step is done by running Bellman-Ford instances from different sources in parallel. But directly implementing this in distributed settings would need $\Omega(\sqrt{n})$ rounds due to congestion. This is where we use a recent result by [19] stating that we can compute $(1+\epsilon)$-approximate distances from $O(\sqrt{n})$ sources in $O(\log^2(n)/\epsilon)$ time. We point out that in [25], in order to get sparse hopsets, they use a recursive construction for small clusters. Such a recursion would introduce significant overhead in the hopbound guarantee. Here we show that in Congested Clique by using the tools described above we can avoid using the recursive construction and still compute sparse hopsets.

We explain briefly why the constructed hopset has the size and hopbound properties stated in Theorem 3.1.1. To see this, we use similar arguments as in [25]: for a distance scale $[R, 2R)$ consider a shortest path of length at most $2R$, and consider $O(\log(n)/\epsilon)$ segments of length W on this path. By definition of a W-pairwise cover, each such segment is contained in a cluster. If this segment is in a small cluster, there is a corresponding path with at most β' edges, where β' is the hopbound of the local construction. For big clusters, we either add a single edge, or if there is more than one big cluster, the whole segment between these clusters has a corresponding edge in the hopset. By similar considerations and by the triangle inequality we can show that the stretch of the replaced path is $(1+\epsilon)$. We need a tighter size analysis than the one used in [25] to prove the desired sparsity. We use a straight-forward bucketing argument as follows: for each cluster of size $\Theta(s)$, $\Theta(s^{1+1/k})$ edges will be added. Then by noting that are at most $O(n/s)$ clusters with this size we can bound the overall size.

Bounding the exploration depth. For large values of R, the shortest path explorations up to distance R could take $\Omega(n)$ rounds in distributed settings. To keep the round complexity small, we use the following idea from [25] (also used in [38] and [19]): we can use the hopset edges constructed for smaller distance scales for constructing hopset edges for larger distance scales more efficiently. The intuition behind this idea is that any path with length $[R, 2R)$ can be divided into two segments, such that for each of these segments we already have a $(1+\epsilon)$-stretch path with β hops using the edges added for smaller distance scales. This allows us to limit the explorations only to paths with $2\beta+1$ hops in each iteration. This process will impact the accuracy, and so in order to keep the error small we have to construct the hopsets for a fixed scale at a higher accuracy. This is where a factor polylogarithmic in n will be introduced in the hopbound, which can generally be avoided in the centralized constructions (e.g. see [25, 38]). This idea is formalized in Lemma 3.2.1.

3.1.3 Preliminaries

We review the notation and review the models considered here again. Given a weighted undirected graph $G = (V, E, w)$, and a pair $u, v \in V$ we denote the (weighted) shortest path distance by $d(u, v)$. We denote by $d^{(\ell)}(u, v)$ the length of the shortest path between u and v among the paths that use at most ℓ hops, and call this the ℓ-hop limited distance between u and v. For each node $v \in V$, we denote the (weighted) radius r neighborhood around v by $B(v, r)$, and we let $B^{\ell}(v, r)$ be the set of all nodes $u \in V$ such that there is path π of (weighted) length at most r between u and v such that π has at most ℓ hops.

Models. We construct limited neighborhood covers in the more classical *CONGEST* model, in which we are given an undirected graph $G = (V, E)$, and in each round nodes can send a message of $O(\log(n))$-bits to each of their neighbors in G (different messages can be sent along different edges). Recall, that in the *Congested Clique* model, we are given a graph with n nodes, where all nodes can send a message with $O(\log(n))$-bits to *every* other node in the graph in each round [72]. In other words, this is a stronger variant of the CONGEST model, in all nodes can communicate with each other directly.

We also consider a *variant* of the low memory Massively Parallel Computation model. In the standard MPC model, for graph problems the total memory N is $O(m), m = |E|$ words. But here we allow the total memory to be larger, while the memory per machine is still strictly sublinear in n. In other words, each machine has $O(n^{\gamma}), \gamma < 1$ memory, where $n = |V|$, but the overall memory is going to be $\Omega(mn^{\alpha})$, for a specific constant $0 < \alpha < 1$.

Even though we do not give any new PRAM results, we use multiple tools from PRAM literature. In the PRAM model[5], a set of processors perform computations by reading and writing on a shared memory in parallel. The total amount of computation performed by all processors is called the *work*, and the number of parallel rounds is called the *depth*.

[5]We just use a simple abstraction without details of the exact parallel model (EREW, CRCW, etc), since PRAM is not our focus and there are reductions with small overhead between these variants.

3.2 Algorithmic Tools.

In this section we describe several algorithmic tools from previous work that we will be using.

Bounding the shortest path exploration. As explained earlier, for an efficient hopset construction, we need to first compute hopsets for smaller distance scales and then use the new edges for computing future distances. This will let us limit the shortest path explorations to a logarithmic number of hops in each round. More formally,

Lemma 3.2.1 ([25, 38])**.** *Let H^k be the hopset edges for distance scale $[2^{k-1}, 2^k)$ with hopbound β. Then for any pair u, v where $d(u, v) \in [2^k, 2^{k+1})$, there is a path with $2\beta + 1$ hops in $G \cup (\cup_{i=\log(\beta)}^{k} H^i)$ with length $(1 + \epsilon)$-approximate of the shortest path between u and v.*

Roughly speaking, the above lemma implies that we can use previously added edges and only run Bellman-Ford for $2\beta + 1$ rounds for each iteration of our algorithm.

Lenzen's routing. Given a set of messages such that each node is source and destination of at most $O(n)$ messages, these messages can all be routed to their destination in $O(1)$ time in Congested Clique [68].

Multi-source shortest path and source detection in Congested Clique. We use the following two results by [19]. First result is a multi-source shortest path algorithm that we use as a subroutine in our hopset construction:

Lemma 3.2.2 (MSSP, [19])**.** *Given a weighted and undirected graph, there is an algorithm that computes $(1 + \epsilon)$-approximate distances distances from a set of $O(\sqrt{n} \log n)$ sources in $O(\frac{\log^2(n)}{\epsilon})$ rounds in the Congested Clique model.*

The second result solves a special case of the so-called *source-detection* problem that we use to prove Corollary 3.1.2:

Lemma 3.2.3 (Source detection, [19])**.** *Given a fixed set of $O(\sqrt{n})$ sources S, we can compute ℓ-hop limited distances from all nodes to each of the nodes in S in $O(\ell)$ rounds in the Congested Clique model.*

3.3 Neighborhood covers using low-diameter decomposition

In this section, we describe an algorithm for constructing pairwise covers in the CONGEST model. We first give an algorithm for W-pairwise covers in weighted graphs that runs in $O(W \log^2(n))$ rounds. We then provide an ℓ-limited W-pairwise cover that runs in $O(\ell \log^2(n))$ rounds. Clearly, the CONGEST algorithm can also be used in Congested Clique with the same guarantees. We will use the low-diameter decomposition algorithm that was proposed in [75] and extended (to weighted graphs) in [74] for computing pairwise covers in PRAM. First we state their PRAM result:

Theorem 3.3.1 (MPX [74, 75])**.** *Given a weighted an undirected graph $G = (V, E, w)$, there is a randomized parallel algorithm that partitions V into clusters $\mathcal{X}_1, \mathcal{X}_2, \ldots$ such that w.h.p. the (strong) diameter of each cluster $\mathcal{X}_i$ is at most $O(\frac{\log(n)}{\alpha})$. This algorithm has $O(\alpha^{-1} \log(n))$ depth*[6] *w.h.p. and $O(m)$ work.*

We denote the algorithm of [74] for a parameter $0 < \alpha < 1$ by LDD(α), which is as follows: each node $u \in V$ first chooses a random radius r_u based on an exponential distribution $\exp(\alpha)$. Each node $v \in V$ joins the cluster of node $u = \arg\min_{x \in V}(d(v, x) - d_x)$. Ties can be broken aribtrarily. It is easy to see that based on simple properties of exponential random variables the weak diameter of each cluster

[6]Depending on the exact PRAM model considered the depth may have a small extra factor of $O(\log^*(n))$.

is $O(\alpha^{-1}\log(n))$ with high probability. But it can be shown that the clusters also have strong bounded diameter of $O(\alpha^{-1}\log(n))$ (as argued in [75, 74]). This means the diameter of the subgraph induced by each cluster is $O(\alpha^{-1}\log(n))$ as opposed to the weak diameter guarantee, which bounds the diameter with between each pair of nodes in the cluster based on distaces in G. The second property is that we can lower bound the probability that the neighborhood around each node is fully contained inside one cluster by a constant. This was shown in [74], but we give a proof sketch for completeness.

Lemma 3.3.2 (Padding property[7], [74])**.** *Let $\mathcal{X}$ be a partition in support of the LDD(α) algorithm. For each node $u \in V$, the probability that there exists $C \in \mathcal{X}$ such that $B(u,r) \subseteq C$ is at least* $\exp(-2r\alpha)$*.*

Proof sketch. For each node u we will consider the subgraph induced by $B(u,r)$. For each node $v \in V$, consider the random variable $Y_v := r_v - d(u,v)$. Let Y_1 denote the largest Y_v over $v \in B(u,r)$, and let Y_2 denote the second largest value. We argue that the probability that $B(u,r)$ intersects more than one cluster is at most $1-\exp(-2r\alpha)$. This event occurs when Y_1 and Y_2 are within $2r$ of each other. Therefore we only need to bound the probability that $Y_1 - Y_2 < 2r$. This now follows from Lemma 4.4. of [75] that claims the following: given a sequence of exponential random variables $r_1, r_2, ..., r_n$, and arbitrary values $d_1, d_2, ..., d_n$ the probability that largest and second largest values $r_i - d_i$ are within δ of each other is at most $1-\exp(-\delta\alpha)$. This implies the probability that $B(u,r)$ intersects more than one cluster is at most $1-\exp(-2r\alpha)$ and this proves the claim. For more details see [75], [74]. $\square$

In order to compute W-neighborhood covers *sequentially*, we can use the above theorem by setting $\alpha = 1/W$ and repeating the partition algorithm $O(\log(n))$ times. It follows from a standard Chernoff bound that the desired properties hold with high

[7]Lemma 2.2 in [74] upper bounds the probability that a ball overlaps with k or more clusters, but Lemma 3.3.2 is a straightforward corollary of this claim.

probability. Implementing this algorithm in distributed settings may take $\Omega(n)$ rounds for large values of W. To resolve this, we use a relaxed notion similar to the notion of *ℓ-limited W-pairwise cover* proposed in [25]. This structure has all the properties of a W-pairwise cover but the path covering property only holds for paths with at most ℓ-hops. More formally, the third property will be relaxed to require that for every path p of weight at most W with at most ℓ hops there exists a cluster C where $p \subseteq C$. We define an ℓ-limited W-neighborhood cover similarly: for each node u, there is a cluster C such that $B^{\ell}(u, W) \subseteq C$. In [25], Cohen shows that we can construct ℓ-limited W-pairwise covers in $O(\ell)$ parallel depth, independent of W. We will show that this concept can also be utilized to limit the number of rounds for LDD(α) partitions to $O(\ell)$.

ℓ-limited W-pairwise cover. Since running LDD(α) by setting $\alpha = 1/W$ will require many rounds, we cannot directly use the weighted variant of LDD(α). Instead, we use a rounding idea that allows to run LDD(α) on the graph obtained from rounded weights, only for $\alpha = O(1/\ell)$, at the cost of a small loss in accuracy. This idea was proposed by [64] and is used widely in PRAM literature (e.g. [25], [74]). In the context of distributed algorithms a similar approach was used by [76] in CONGEST, but directly applying the result of [76] to our settings will require a polynomial running time, since we would need to run the algorithm from many (polynomial) sources. The idea is based on the following observation: consider a path π with at most ℓ hops, such that $R \leq w(\pi) \leq 2R$ for a fixed $R > 0$. Then by slightly changing the weights of each edge $e \in \pi$ by a small additive factor such that for the new weight $\hat{w}$ it holds $w(e) \leq \hat{w}(e) \leq w(e) + \frac{\epsilon_0 R}{\ell}$ for an arbitrary $\epsilon_0 > 0$. We then get $\hat{w}(\pi) \leq w(\pi) + R\epsilon_0 \leq (1+\epsilon_0)w(\pi)$. This can be achieved by setting $\hat{w}(e) = \lceil \frac{w(e)}{\eta} \rceil$, where $\eta = \frac{\epsilon_0 R}{\ell}$. We note that this scaling lemma is also going to be used in Chapter 4, Lemma 4.2.4, for constructing hopsets more efficiently in dynamic settings.

Lemma 3.3.3 ([64]). *Given a weighted graph $G = (V, E, w)$, and a parameter R, there is a rounding scheme that constructs another graph $\hat{G} = G = (V, E, \hat{w})$ such that any path π with at most ℓ hops and weight $R \leq w(\pi) \leq 2R$ in G, has $\hat{w}(\pi) \leq \lceil 2\ell/\epsilon_0 \rceil$ in $\hat{G}$. Moreover, $w(\pi) \leq \eta(R, \ell) \cdot \hat{w}(\pi) \leq (1 + \epsilon)w(\pi)$, where $\eta(R, \ell) = \epsilon_0 R/\ell$.*

We can now run LDD(α) for $\alpha = O(\ell)$ on $\hat{G}$, and each path π with at most ℓ hops will be fully contained in some cluster with probability at least $\exp(-\ell \cdot O(1/\ell)) = \Omega(1)$. We can then recover an estimate to the original length $w(\pi)$ by setting $\tilde{w}(\pi) = \eta(R, \ell) \cdot \hat{w}(\pi)$, and we have $w(\pi) \leq \tilde{w}(\pi) \leq (1 + \epsilon_0) \leq w(\pi)$. Same as before, by repeating the LDD(α) algorithm $O(\log(n))$ times we will get an ℓ-limited W-neighborhood cover. We first argue that this algorithm can be implemented $O(\alpha \log^2(n))$ rounds of the CONGEST model. A similar construction was used in [79] for W-neighborhood covers in CONGEST. But the result of [79] only focuses on unweighted graphs, and would take $O(W \log(n))$ rounds.

Theorem 3.3.4. *Given a weighted graph $G = (V, E, w)$, there is an algorithm that constructs an ℓ-limited W-pairwise cover in $O(\ell \log^2(n) \log(W))$ rounds in the CONGEST model, with high probability. Moreover, a pairwise cover for paths with ℓ-hops with length in $[W, 2W]$ can be constructed in $O(\ell \log^2(n))$ rounds with high probability.*

Proof. As we argued by using the rounding technique of [64], for any pair of nodes u, v such that $d^{(\ell)}(u, v) \in [W, 2W)$ we can restrict our attention to another graph $\hat{G}$ with rounded weights. We construct a pairwise cover on $\hat{G}$ by running the LDD(α), $\alpha = \epsilon_0/2\ell = \Theta(1/\ell)$ algorithm $O(\log(n))$ times independently.

We argue that each run of LDD($O(1/\ell)$) takes $O(\ell)$ rounds in the CONGEST model. First we observe that each node u only needs to broadcasts the value r_u to all the nodes within its r_u neighborhood, since a node x will not join the cluster of u if $r_u - d(u, x) < 0$. We can now use a simple induction to prove the claim. In each round, each node u will forward the radius and distances corresponding to the node $u_{\max}$ that maximizes

$r_{u_{\max}} - d(u, u_{\max})$ among over all the messages that u has received. We now argue that each node u will receive the message from the node $c = \arg\max_{v \in V} r_v - d(u, v)$ in r_c rounds. Consider any path $\pi = \{c = u_0, u_1, ..., u_j = u\}$, where $j \leq r_c$. If $j = 1$ then in one single round c sends $(r_c, w(c, u_1))$ to u_1. Assume now that u_i receives the message $(r_c, d(u, u_{i-1}))$ in round i. Then u_i will compute $d(u, u_i)$ (after receiving distance estimates from all neighbors), and forwards $(r_c, d(c, u_i))$ to all neighbors including u_{i+1}. Therefore in round $i = j \leq r_c$), u_j has received the message $r_c - d(u, c)$, and can compute $d(u, c)$. Therefore this algorithm will terminate after $\max_{v \in V} r_v$ rounds. Since r_u is an exponential random variable with parameter $O(1/\ell)$, we know that maximum of these $O(n)$ exponential random variables is $O(\ell \log(n))$ with high probability. Now we need to repeat the partition algorithm $O(\log(n))$ times and will pipeline the broadcasts for different runs. Clearly, each node is in at most in $O(\log(n))$ clusters. A standard Chernoff bound in combination with Lemma 3.3.2 implies that with high probability after $O(\log(n))$ repetition of the ℓ-limited LDD$(O(1/\ell))$ algorithm, for each path π with at most ℓ hops and length $w(\pi) \in [R, 2R]$, there will be a cluster C such that $\pi \subseteq C$. We then repeat this process for $O(\log(W))$ distance scales to get an ℓ-limited W-pairwise cover. □

As we will see, since in our hopset construction we consider different distance scales and need to compute pairwise for a fixed scale, this step takes only $O(\ell \log(n))$ rounds.

Diameter guarantee. For constructing pairwise covers, we need the diameter guarantee of $O(W \log(n))$ for all clusters. While running LDD$(O(1/\ell))$ gives a diameter guarantee of $O(\ell \log(n))$ on $\hat{G}$, we note that the construction ensures that clusters have diameter $O(W \log(n))$ on G. Since we argued that every ℓ hop path with length W will fall into a cluster with high probability, the diameter guarantee of $O(\ell \log(n))$ on $\hat{G}$ will imply that the corresponding cluster in G will have length $O(W \log(n))$.

More formally, for any pair of nodes there is a path with length $O(\ell \log(n))$ in $\hat{G}$. Let $\mathcal{C}$ denote the cluster that contains this path. Consider each segment of length $O(\ell)$ in $\hat{G}$ is in $\mathcal{C}$ and will be have length $O(W)$ in G (by Lemma 3.3.3), and thus there will be a path of length $O(W \log(n))$ based on weights in G in $\mathcal{C}$.

Extension to neighborhood covers. While Cohen shows that for the parallel construction of hopsets pairwise covers are enough, for the distributed implementation we need one more property: each vertex should overlap with at most $O(\log(n))$ clusters. Moreover, the algorithm used in Theorem 3.3.4 provides the stronger guarantee that there will be a cluster that contains the neighborhood of (weighted) radius W from each vertex with high probability, rather than only containing paths of length W. In other words, a similar analysis shows that with high probability an ℓ-limited neighborhood cover can be constructed in $O(\ell \log(n))$ rounds of the CONGEST model. That is, for each node u, the ℓ-limited W-neighborhood of u will be fully contained in a cluster with high probability. However, for our purposes the path covering property suffices.

3.4 Congested Clique Hopset Construction

In this section we describe our main algorithm. Similar to the sequential construction described we consider different distance scales $[R, 2R)$, and handle each scale separately. In each iteration, we construct a *sparse* ℓ-limited $2R$-neighborhood cover as described in Section 3.3. Then the clusters will be divided into small and big clusters, and each case will be handled differently. So far the construction is similar to [25]. The key new idea is that for Congested Clique, by setting the parameters carefully we can send the topology corresponding to a small cluster to the cluster center, and build a hopset locally. Here we need to use the fact that each node is in at most $O(\log(n))$ clusters, which is a property that we get from our neighborhood cover construction. We will also need to compute pairwise distances between big clusters centers. For this step, we use the algorithm of [19] that computes $(1+\epsilon)$-multi-source shortest path from $O(\sqrt{n}\log n)$ sources (Lemma 3.2.2). We note that while during our construction we construct the denser hopsets of [19] as auxiliary structure, these extra edges will be removed at the end of each distance scale.

Finally, we use Lemma 3.2.1 to use the hopset edges added for smaller distance scales to construct the larger distance scales. For this to give us a $(1+\epsilon)$ for an arbitrary ϵ, we first let ϵ' be the error parameter. Since we use paths with error $(1+\epsilon')$ for each scale, to compute distances for the next scale, a multiplicative factor in the stretch will be added in each iteration. This means that after i iterations the error will be $(1+\epsilon')^i$. We can simply rescale the error parameter by setting $\epsilon' = O(\frac{\epsilon}{\log(n)})$ to get arbitrary error overall of $\epsilon > 0$.

Throughout our analysis w.l.o.g we assume the minimum edge weight is one. Otherwise, we can scale all the edge weights by the minimum edge weight. We also assume the aspect ratio is polynomial. Otherwise, we can use reductions from previous work to reduce the aspect ratio in exchange in polylogarithmic depth (this will be a preprocessing step and will not dominate the overall running time).

An overview of the construction is presented in Algorithm 8. By defining small clusters to have size at most $\sqrt{n}$, we have that the number of edges in each small cluster C is $O(n)$, and hence all the nodes C can send their incident edges to the cluster center in constant rounds using Lenzen's routing [68]. Then the cluster center computes a hopset with size $O(n^{1/2+1/2k})$ and hopbound $\beta_0 = O(\log(k)/\epsilon')^{\log(k)-1}$ *locally* using Huang-Pettie [58] centralized construction. The center of a small cluster C can send the edges incident to each node in that clusters. Since the size of the hopset on small clusters is always $O(n^{\frac{1}{2}+\frac{1}{2k}}) = O(n)$, this can also be done in constant time using Lenzen's routing.

As explained in Lemma 3.2.1, using hopset edges added for smaller scales we can limit all the shortest path explorations to $2\beta+1$. So we can add the star edges, by

Algorithm 5: Congested Clique construction (β, ϵ)-hopset of size $\tilde{O}(n^{1+\frac{1}{2k}})$

```
1 Let $H_i$ denote the hopset edges for scale $(2^i, 2^{i+1}]$, and set $\epsilon' = O(\frac{\epsilon}{\log(n)})$.
2 for $(R, 2R]$, where $R = 2^\kappa, \log(\beta) \leq \kappa \leq O(\log(n))$, on $G \cup_{i \geq \log(\beta)}^{\kappa-1} H_i$ do
3     Set $W = \frac{\epsilon' R}{4(\log n)}$, and and build $\beta$-limited $W$-pairwise covers (by Theorem 3.3.4).
4     Let $\mathcal{C}_b$ be the set of big clusters that have size at least $\sqrt{n}$ and $\mathcal{C}_b$ small clusters with size less than $\sqrt{n}$.
5     for each $C \in \mathcal{C}_s$ do
6         All the nodes in cluster $C$ send their incident edges to the center.
7         The center locally computes a hopset of size $O(n^{1/2+1/2k})$ (construction of [58]) with $(\beta_0, \epsilon')$-hopsets with $\beta_0 = O(\frac{\log k}{\epsilon'})^{\log(k+1)-1}$.
8         The center sends the new hopset edges to the corresponding nodes (endpoints) in $C$.
9     for each $C \in \mathcal{C}_b$ do
10        Add a star on $C$ by adding edges from the cluster center $x$ to all $v \in C$ (that are within $\ell = 2\beta + 1$ hops), and set the weight of $(x, v)$ to $d^{(\ell)}(x, v)$.
11    Add an edge between any pair $u_1, u_2$ of centers of big clusters that are within $\ell = 2\beta + 1$ hops of each other, and set the weight to $d^{(\ell)}(u_1, u_2)$.
```

running $2\beta + 1$ rounds of Bellman-Ford. For adding a clique between centers of large clusters, we will use the $(1+\epsilon)$-MSSP algorithm of Censor-Hillel et al. 2019 [19] (using Lemma 3.2.2. This is possible since there are at most $O(\sqrt{n} \log n)$ big cluster centers. We disregard all the other edges added in this step for computing these distances after the computation. We now analyze the algorithm and show that it has the properties stated in Theorem 3.1.1.

Size. Recall that large clusters have size at least $\sqrt{n}$. The stars added for each big cluster will add $O(n \log^2(n))$ edges overall since they are consisted of unions of $O(\log(n))$ forests for each scale. The (clique) edges added between centers of big clusters will add $O(n)$ edges overall. For small clusters of size $s = O(\sqrt{n})$, we added a hopset of size $s^{1+1/k}$ (this is the guarantee we get by using Huang-Pettie hopsets), for a parameter $k \geq 2$. On the other hand, we have at most $O(\frac{n}{s})$ clusters of size within

$[s, 2s]$. Therefore we can estimate the overall number of edges added for these small clusters in each scale by summing over different values $s \in [2^r, 2^{r+1})$ for small clusters as follows:

$$\sum_{s \in [\sqrt{n}]} O(\frac{n}{s} \cdot s^{1+1/k}) = \sum_{r=1}^{\log(\sqrt{n})} O(\frac{n}{2^r} \cdot (2^r)^{1+1/k}) = \sum_{r=1}^{\log(\sqrt{n})} O(n \cdot 2^{\frac{r}{k}}) = O(n^{1+\frac{1}{2k}}).$$

Therefore, the overall size for all scales is $O(n^{1+\frac{1}{2k}} \log(n) + n \log^2(n))$.

Hopbound and Stretch. Fix a distance scale $(R, 2R], R = 2^k$ and consider a pair of nodes $u, v \in V$ where $d(u, v) \in (R, 2R]$. If $R \leq \log \beta$, since we assumed the minimum edge weight is one, this implies that the shortest path has at most $O(\beta)$ hops and no more edges is needed for this pair.

We argue that for any pair of nodes u, v, where $d(u, v) \leq (R, 2R]$. We argue that after the hopset edges added for scale $(R, 2R]$, means there is a path with at most β-hops and stretch $1 + \epsilon'$.

Let π be the shortest path between u and v in G. We divide π to $\frac{\log(n)}{8\epsilon'}$ roughly equal (of at most $W = \epsilon' R/4 \log n$) length segments. Note that we may have some segments containing a single edge.

The properties of a neighborhood cover imply that each of these segments are w.h.p. contained in one cluster. First assume that all these clusters are small. Then the segment in this cluster has a corresponding path with hopbound at most β', obtained by local hopsets for the cluster, where $\beta_0 = O((\log(k)/\epsilon')^{\log(k+1)-1})$. In this case, we have replaced each segment of length W with a path of stretch $(1+\epsilon')$. By the triangle inequality, overall we get a $(1 + \epsilon')$-stretch.

Otherwise assume that there is only one big cluster, and all other segments correspond to small clusters. Then the segment corresponding to this cluster can be replaced with two edges, going through the center of this cluster. The segment corresponding to this single cluster will just add a single additive $W \log n = \epsilon' R$ cost to our distance estimate.

Finally, assume that there are more than one big clusters corresponding to π. Consider the two furthest big clusters (based on their centers) on π, and let their centers be x (corresponding to the segment closest to u and y (corresponding to the segment closest to v). Since x corresponds to the first big cluster, we can use an argument similar to the first case to show that there each cluster over the whole segment between u and x, has a corresponding path with stretch $(1 + \epsilon')$ of length β_0. We can use a similar argument for the small clusters that appear after y. What remains is therefore the segment between x and y on π. But note that by construction, we have added one single edge with $(1 + \epsilon')^k$-stretch of $d(x, y)$ that covers the whole segment between these two centers (the errors multiply, since we are using smaller hopsets) . Therefore, we have shown that all segments of π have a corresponding path within $(1 + \epsilon')$-stretch.

As argued, combining this with the idea that we are using hopsets for smaller scales to compute distances on larger scales, we see that each scale incurs a multiplicative factor of $(1 + \epsilon')$ in the stretch, and thus by setting $\epsilon' = O(\epsilon / \log(n))$, and since we assumed that the weights are polynomial we can get $(1 + \epsilon)$-stretch for all scales.

Hence the hopbound for all scales is,

$$\beta = O(\frac{\log(n)}{\epsilon'}(\frac{\log(k)}{\epsilon'})^{\log(k+1)-1})) = O(\frac{\log^2(n)}{\epsilon}(\frac{\log(n)\log(k)}{\epsilon})^{\log(k+1)-1}).$$

Round Complexity. For each of the $O(\log(n))$ distance scales, it takes $O(\beta \log^2(n))$ rounds to compute $2\beta + 1$-limited neighborhood covers (Lemma 3.3.4). Once the covers are constructed for small clusters we need to run a Bellman-Ford with $O(\beta)$ hops from the center of each big cluster and since each node may overlap with at most $O(\log(n))$ clusters this phase takes $O(\beta \log(n))$ (each node can pipeline the computation over the clusters it overlaps with). For small clusters, we argued that in

$O(1)$ rounds (using Lenzen's routing) the whole small cluster topology can be sent to the cluster center, and after local computation another $O(1)$ rounds will be enough for cluster center to send back the new hopset edges to the destination node. Finally, using the result of [19] we can compute $(1+\epsilon')$-approximation from big cluster centers ($O(\sqrt{n}\log n)$ sources) in $O(\frac{\log^2(n)}{\epsilon'}) = O(\frac{\log^3(n)}{\epsilon})$ time. Therefore the overall running time is $O(\beta\log^2(n))$.

Application to multi-source queries. We can now combine our hopset construction with Lemma 3.2.2 (source detection algorithm of [19]) to show that we can compute queries from $O(\sqrt{n})$ sources in $O(\beta)$ time, by maintaining a sparse hopset of size $\tilde{O}(n^{1+\frac{1}{2k}})$, while [19] has to store a hopset of size $\tilde{O}(n^{3/2})$. Corollary 3.1.2 follows from this observation.

3.5 Massively Parallel Hopsets and MSSP

In this section, we argue that in a variation of the MPC model where the overall memory is $O(mn^{1/2})$ we can construct hopsets with small hopbound efficiently, and this in turn gives us a fast algorithm for multi-source shortest path in this case. This result relies on an observation made in Chapter 2, stating that the PRAM hopset constructions (e.g. [25], [38]) that use $O(m\alpha)$ processors with depth t can be implemented in MPC, even when the memory per machine is strictly sublinear, in $O(t)$ rounds if we assume that the overall memory available is $O(m\alpha)$. Once a (β,ϵ)-hopset is constructed, the Bellman-Ford subroutine described in Chapter 2 can be used to compute $(1+\epsilon)$-stretch distances from $O(\sqrt{n}\log n)$ nodes to all other nodes in V.

The construction we have in Chapter 2 are based on hopsets of [38], and their constructions may use less overall memory in general, but they get a worse hopbound than ours in the special case that the total memory is $\tilde{\Omega}(m\sqrt{n})$. In this case, we get an improved hopbound of $O(\log^2(n)/\epsilon)$, whereas their result gives a hopbound of

$O((\log(n)/\epsilon)^3)$. In particular, we use the PRAM hopset construction of [25] (instead of [38]), which can be simulated in the MPC model with strictly sublinear memory per machine (using a reduction of [50]) to construct hopsets with hopbound $O(\log^2(n)/\epsilon)$. The only difference between our construction and [25] is using a faster algorithm for constructing ℓ-limited pairwise covers based on the algorithm of [74]. First we note that our ℓ-limited W-neighborhood cover construction can be constructed in MPC based on a very similar algorithm and analysis as in Section 3.3. This step can be done only using $O(m \log^2(n))$ overall memory (or $O(m\log(n))$ memory for a single-scale) in $O(\ell \log(n))$ rounds. Observe that the construction of W-neighborhood covers for different scales $[W, 2W]$ can all be done in parallel with an extra logarithmic overhead in the *total memory*. Similarly since each of the low diameter partitions are independent, the repetitions of the LDD algorithm can also be parallelized. This result is also implied by results in [74], combined with goodrich [50]. We have,

Lemma 3.5.1. *There is an algorithm that runs in $O(\frac{\ell}{\gamma} \cdot \log(n))$ rounds of MPC and w.h.p. computes an ℓ-limited W-neighborhood cover, where memory per machines is $O(n^\gamma), 0 < \gamma \leq 1$ and the overall memory is $O(m\log^2(n))$.*

$(1+\epsilon)$-MSSP. Given a pairwise cover, assuming that in MPC we have $\tilde{O}(mn^{1/2})$ total memory, we can construct a $(\log^2(n)/\epsilon, \epsilon)$-hopset of size $O(n^{3/2}\log(n))$. This hopset is a special case of hopsets of [25]: we add a clique for small clusters, a star centerd at each big cluster, and a clique between big cluster centers. As stated, the main difference in our algorithm is that we use the algorithm of Lemma 3.5.1 for constructing pairwise covers, rather than the algorithm of [24]. This leads to a construction time of $O(\beta \log(n))$, whereas a direct reduction from [25] would have construction time of $O(\beta \log^3(n))$, which is how long it takes to construct their limited pairwise covers. Hence combining Lemma 3.5.1 with simulating the PRAM construction of [25], and the Bellman-Ford primitives described in Chapter 2, we can construct a hopset of size

$O(n^{3/2}\log(n))$ in $O(\beta\log(n))$ time with hopbound $\beta = O(\log^2(n)/\epsilon)$.

Theorem 3.5.2. *Given an undirected weighted graph G, and parameters $\epsilon > 0, 0 < \gamma \leq 1$, we can w.h.p. construct an (β,ϵ)-hopset of size $O(n^{3/2}\log(n))$ in $O(\frac{\log^3(n)}{\gamma\epsilon})$ rounds of MPC, using $O(n^\gamma)$ memory per machine, and the overall memory of $O(mn^{1/2})$ (i.e. there are $O(mn^{1/2-\gamma})$ machines), where hopbound is $\beta = O(\frac{\log^2(n)}{\gamma\epsilon})$.*

The analysis is very similar to the arguments in previous sections and previous work. Similarly, for $(1+\epsilon)$-MSSP we get,

Theorem 3.5.3. *Given an undirected weighted graph G, after a preprocessing step of $O(\frac{\log^3(n)}{\gamma\epsilon})$ rounds, we can w.h.p. compute $(1+\epsilon)$-multi source shortest path queries from $O(n^{1/2})$ sources in $O(\frac{\log^2(n)}{\gamma\epsilon})$ rounds of MPC, when the memory per machine is $O(n^\gamma), 0 < \gamma \leq 1$, and the overall memory required for preprocessing is $O(mn^{1/2})$.*

At a high-level since we have overall memory of $\tilde{O}(mn^{1/2})$, to each node u, we can assign a block of memory of size $O(\deg(u).n^{1/2})$. Then using aggregations primitives (e.g. see Chapter 2), we can store and update the distances from up to $O(n^{1/2})$ sources. Therefore given a hopset with hopbound $O(\log^2(n)/\epsilon)$, we can compute distances from $O(n^{1/2})$ sources by running parallel Bellman-Ford.

3.6 Conclusion and Open Problem

In this chapter we used the results by [19] as a black-box to construct sparse hopsets with low hopbound. While this leads to a sparser data structure, such an approach does not allow us to handle shortest path computation from a larger number of sources than [19]. A natural direction is to use more fine-grained properties of their sparse matrix multiplication techniques, and try to combine them with our sparse hopset construction. This may let us handle more sources, as a recent work by [42] showed using a different algorithm.

Chapter 4

Dynamic Hopsets

4.1 Introduction

In this chapter, we consider a model of computation that captures a different phenomenon in modern massive data sets. In many practical scenarios, the input is not accessible in one shot. Instead it is evolving over time, and we only have access to the changes or *updates* to the input over time. We consider a *centralized* dynamic model, i.e. as opposed to most the other models in this thesis, the input is not distributed over different machines.

Recall, a graph algorithm is called dynamic if it supports answering *queries* about a graph which is undergoing modifications, or, as we say in the following, *updates*. Each update is an edge deletion, insertion, or a weight change. In this chapter, we focus on designing *decremental* algorithms for distance problems in weighted graphs. In the decremental setting, the updates are only edge deletions or weight increases. This is as opposed to an *incremental* setting in which edges can be inserted, or a *fully dynamic* setting, in which we have both insertions and deletions. For simplicity in stating our results, we only consider edge deletions rather than weight increases, but it is not hard to see that our amortize update time bounds also hold for a sequence of weight increases. We use the term total update time to refer to the update time of a sequence of m deletions.

Our focus is on fast decremental algorithms for constructing hopsets. We provide a near-optimal algorithm for maintaing decremental hopset in a wide range of settings. We then apply our hopset construction to obtain faster shortest path algorithms. In particular, we consider the problem of maintaining shortest paths from a fixed set S of sources. We consider different variants of the problem which differ in the size of S: the single-source shortest paths (SSSP) problem ($|S| = 1$), all-pairs shortest paths (APSP) problem ($|S| = n$, where n is the number of vertices of the input graph), as well as the multi-source shortest paths (MSSP) problem (S is of arbitrary size), which is a generalization of the previous two. Specifically, given a weighted graph $G = (V, E, w)$, we want to support the following operations: $\textsc{Delete}((u, v))$, where $(u, v) \in E$, which removes the edge (u, v), $\textsc{Distance}(s, u)$, which returns an (approximate) distance between a source s and any $u \in V$, and $\textsc{Increase}((u, v), \delta)$, which increases the weight of the edge (u, v) by $\delta > 0$.

The best known algorithm for maintaining exact distances under deletions takes $O(mn)$ total update time [87, 62], even if we limit ourselves to unweighted and undirected graphs. In fact, this bound matches a widely believed conditional lower bound [57]. Hence a large body of work [15, 16, 21, 55] focused on maintaining *approximate* distances. Allowing approximate distances enabled significant speedups in the running time.

Following this line of work, we provide efficient decremental algorithms for maintaining $(1 + \epsilon)$-approximate SSSP and MSSP and $(2k - 1)(1 + \epsilon)$-approximate APSP in *weighted undirected graphs*.

While prior to this work, hopsets were extensively studied in other models of computation (e.g. distributed and parallel settings), their applicability in dynamic settings was not very well-understood. The few exceptions include, utilizing hopsets in the state-of-the art decremental SSSP algorithm for undirected graphs by Henzinger, Krinninger and Nanongkai [55], and implicit hopsets considered in [15, 21]. We use some of the hopset techniques from parallel and distributed settings, and show that they can be useful for obtaining efficient dynamic algorithms. A manuscript of the results in this chapter is also available [67].

Technical connection to previous chapters. In this chapter, similar to Chapter 2, we use hopsets to get better algorithm for maintaining distance oracles. In particular, our APSP result relies on simultaneously maintaining a low-hop hopset and a Throup-Zwick [88] distance oracle. The hopset algorithm used in this section is new, and is tuned to the dynamic settings. Interestingly, we use the same scaling idea as we used in Chapter 3 in improving the running time. The difference is that there we used this scaling idea to obtain faster distributed algorithms for limited neighborhood covers, whereas here we use them to get a faster algorithm for *approximately* maintaining Thorup-Zwick style clusters.

4.1.1 Preliminaries and Notation

We review the notation that is needed for this section. Given a weighted undirected graph $G = (V, E, w)$, and a pair $u, v \in V$ we denote the (weighted) shortest path distance by $d_G(u, v)$. We denote by $d_G^{(h)}(u, v)$ the length of the shortest path between u and v among the paths that use at most ℓ-hops, and call this the h-hop limited distance between u and v. We have defined hopsets in previous chapters, but in this section we define a notion of *restricted hopset* (also used in Chapter 3). Formally:

Definition 4.1.1. Let $G = (V, E, w)$ be a weighted undirected graph. Fix $d, \epsilon > 0$ and an integer $\beta \geq 1$. A $(d, \beta, 1+\epsilon)$-*hopset* is a graph $H = (V, E(H), w_H)$, such that: for all $u, v \in V$, such that $d_G(u, v) \leq d$, we have $d_G(u, v) \leq d^{(\beta)}_{G \cup H}(u, v) \leq (1+\epsilon) d_G(u, v)$. We say that β is the *hopbound* of the hopset and $1+\epsilon$ is the *stretch* of the hopset. We also use $(\beta, 1+\epsilon)$-hopset to denote a $(\infty, \beta, 1+\epsilon)$-hopset. Finally, for any finite d, we say that a $(d, \beta, 1+\epsilon)$-hopset is a *d-restricted hopset*.

In analyzing dynamic algorithms we sometimes also use a time subscript t to denote a distance (or a weight) after the first t updates. In particular we use $d_{t,G}(u,v)$ to denote the distance between u and v after t updates, and similarly use $d^{(h)}_{t,G}(u,v)$ to denote h-hop limited distance between u and v at time t.

4.1.2 Summary of Results

We first state a summary of our results before describing a high-level overview of our techniques.

Decremental Hopsets. The main technical component of our result is a new decremental hopset algorithm. Formally we show the following.

Theorem 4.1.1. *Given an undirected graph $G = (V, E)$ with polynomial weights*[1]*, subject to edge deletions, we can maintain a $(\beta, 1+\epsilon)$-hopset of size $\tilde{O}(n^{1+\frac{1}{2^k-1}})$ in total expected update time $\tilde{O}(\frac{\beta}{\epsilon} \cdot (m + n^{1+\frac{1}{2^k-1}})n^{\rho})$, where $\beta = O(\frac{\log n}{\epsilon} \cdot (k + 1/\rho))^{k+1/\rho+1}$, $0 < \epsilon < 1$ and $\frac{2}{2^k-1} < \rho < 1$.*

Note that the above algorithm, as well as all our results, which use the above construction, are randomized and work against an oblivious adversary.

Our decremental hopset construction covers a wide range of time/hopbound tradeoffs. Importantly, by setting ρ to be a constant, we get the first hopset with *polylogarithmic* hopbound, with a total update time of $\tilde{O}(mn^{\rho})$ which matches (up to polylogarithmic factors) the running time of the best known static algorithm [41, 40] for computing a hopset with polylogarithimic hopbound and $(1 + \epsilon)$ stretch.

In the decremental setting, to the best of our knowledge, the state-of-the art hopset construction [55] has a hopbound of $2^{\tilde{O}(\log^{3/4} n)}$, and can be maintained with $2^{\tilde{O}(\log^{3/4} n)}$ amortized update. By setting $\rho = \frac{\log\log n}{\sqrt{\log n}}$, we can maintain a hopset with hopbound

[1]If weights are not polynomial a factor the $\log n$ factor will be replaced with $\log W$ in the hopbound, and a factor of $\log W$ will be added to the update time, where W is the ratio between largest to smallest edge weight.

$2^{\tilde{O}(\sqrt{\log n})}$ in $2^{\tilde{O}(\sqrt{\log n})}$ amortized time. Thus, compared to [55] we improve both the hopbound and the amortized update time.

The starting point of our decremental hopset algorithm is a static hopset construction by [41]. However, since computing this hopset in the static setting requires computing potentially long shortest paths, it is not clear how to efficiently maintain this hopset in the decremental setting. To deal with that, we construct a new hopset that combines some of the properties of [41] with various dynamic tools. Specifically, to compute our hopset, it suffices to run a number of single-source shortest paths computations up to small depth in a sequence of graphs that we build iteratively. We provide an overview of this construction in Section 4.2.

SSSP. By using our decremental hopset construction, we obtain an algorithm for decremental single-source shortest paths.

Theorem 4.1.2. *Given an undirected and weighted graph $G = (V, E)$, there is data structure for maintaining $(1 + \epsilon)$-approximate distances from a source $s_0 \in V$ under edge deletions, where $0 < \epsilon < 1$ is a constant and $|E| = n \cdot 2^{\tilde{\Omega}(\sqrt{\log n})}$. The total expected update time of the data structure is $\tilde{O}(m \cdot 2^{\tilde{O}(\sqrt{\log n})})$, and the query time is $O(1)$.*

The amortized update time of our algorithm over all m deletions is $2^{\tilde{O}(\sqrt{\log n})}$. This improves upon the state-of-the art algorithm of [55], whose amortized update time is $2^{\tilde{O}(\sqrt{\log n})}$. While the improvement is only by a $n^{o(1)}$ factor, it is *super-polynomial*, since for any constant $c > 0$, $2^{\tilde{O}(\sqrt{\log n})} = O((2^{\tilde{O}(\log^{3/4} n)})^c)$.

MSSP. Our next result is a near-optimal algorithm for multi-source shortest paths.

Theorem 4.1.3 (MSSP)**.** *There is a data structure which given a weighted undirected graph $G = (V, E)$* explicitly *maintains $(1 + \epsilon)$-approximate distances from a set of s sources in G under edge deletions. Assuming that $|E| = n^{1+\Omega(1)}$ and $s = n^{\Omega(1)}$, the* *total expected update time is $\tilde{O}(sm)$. The data structure is randomized and works against an oblivious adversary.*

The total update time matches (up to polylogarithmic factors) the running time of the best known *static* algorithm for computing $(1+\epsilon)$-approximate distances from s sources for a wide range of graph densities. While for very dense graphs, using algorithms based on fast matrix multiplication is faster, the running time of our decremental algorithm matches the best known results in the static settings (up to polylogarithmic factors) whenever $ms = n^{\delta}$, for a constant $\delta \in (1, 2.37)$.

In the dynamic setting, our algorithm improves upon a solution obtained by running the algorithm of Henzinger, Krinninger and Nanongkai [55] independently from each source, giving a total update time of $O(sm \cdot 2^{\tilde{O}(\log^{3/4} n)})$. The advantage of our algorithm is that it decrementally maintains a hopset of polylogarithmic hopbound in $mn^{o(1)}$ time, which then allows it to maintain approximate SSSP in $\tilde{O}(m)$ time. In contrast, the algorithm of [55] maintains a hopset of hopbound $2^{\tilde{O}(\log^{3/4} n)}$, which, if one simply applies existing techniques, results in a total update time of $m2^{\tilde{O}(\sqrt{\log n})}$. In the general case, i.e., for sparse graphs, the update bound of our algorithm is $sm2^{\tilde{O}(\sqrt{\log n})}$, which is still better than the bound obtained by [55].

APSP. Finally, we show that by maintaining both a hopset and a Thorup-Zwick distance oracle we can get the following tradeoffs for approximating distance between any pair of nodes.

Theorem 4.1.4 (Approximate APSP). *For any constant integer*[2] *$k \geq 2$, there is a data structure that can answer $(2k-1)(1+\epsilon)$-approximate distance queries in a given a weighted undirected graph $G = (V, E, w)$ subject to edge deletions. The total expected update time over any sequence of edge deletions is $\tilde{O}(mn^{1/k})$ and the expected size of*

[2]The k here should not be confused with the parameter k in the hopset size.

the data structure is $\tilde{O}(m + n^{1+1/k})$. Each query for the distance between two vertices is answered in $O(k)$ worst-case time.

The currently best known bound for decremental APSP with a similar stretch is due to Chechik [21]. The total update time in [21] is $\tilde{O}(mn^{1/k})(1/\epsilon)^{O(\sqrt{\log n})}$, and the query time is $O(\log\log(nW))$, where W is the largest weight. Our update time improves over this bound by eliminating the $(1/\epsilon)^{\sqrt{\log n}}$. Note that the improvement in the running time holds for constant k. When $k = \omega(1)$, the running time of our algorithm roughly matches the one obtained in [21]. Our results match the best known static algorithm with the same tradeoff (up to $(1+\epsilon)$ in the stretch and polylog in time) by Thorup-Zwick [88]. In addition, the query time of our algorithm is *independent* of n or the aspect ratio (the ratio between largest and smallest edge weight).

Prior to [21], Roditty and Zwick [85] gave an algorithm for maintaining Thorup-Zwick distance oracles in total time $\tilde{O}(mn)$, stretch $(2k-1)(1+\epsilon)$ and $O(k)$ query time for *unweighted graphs.* Later on, Bernstein and Roditty [16] gave a decremental algorithm for maintaining Thorup-Zwick distance oracles in $O(n^{2+1/k+o(1)})$ time using emulators also only for *unweighted graphs.*

Hopsets vs. emulators Unlike our work, most of the previous work on dynamic distance computation are based on algorithms constructing a sparse *emulator* (e.g. [15, 16, 21]). For a graph $G = (V, E)$, an emulator $H' = (V, E')$ is a graph such that for any pair of nodes $x, y \in V$, there is a path in H' that approximates the distance between x and y on G (possibly with both multiplicative and additive factors). While there are some similarities between construction of these objects, their analysis is different. More importantly, the efficient *dynamic algorithms* for maintaining emulators and hopsets have some significant differences. At a high-level, an emulator approximates distances without using the original graph edges and hence we can restrict the computation to a sparser graph, whereas for using and maintaining hopsets we need to use the edges in the original graph as well. On the other hand, hopsets allow for faster computation because we can restrict our attention to paths with few hops, which is utilized differently than an emulator in dynamic settings. One challenge in the decremental implementation of both of these objects is the fact that removing an edge from the graph may lead to both edge deletions and *insertions* in the hopset or emulator. How we handle such insertions is also different for these two objects. In case of emulators, one would argue that the overall decrease in distances after each update is bounded. This is harder to argue for hopsets since we are not only considering a sparse graph. But we see that there are structural properties of the insertions in our specific hopsets that along with ideas similar to the monotone ES tree of [55] can be utilized to handle insertions.

4.2 Overview of Our Algorithms

The starting point of our algorithm is a known static hopset construction [41, 58]. We first review this construction. As we shall see, maintaining this data structure dynamically directly would require update time of up to $O(mn)$. We therefore give another *new decremental hopset* that captures some of the properties of the hopsets of [41, 58], but can be maintained efficiently in a decremental setting. Our new hopset is consisted of the union of restricted hopsets on a sequence of scaled graphs. Our main contribution is this new hopset and a hierarchical algorithm for maintaining a sequence of data structures that together lead to a near-optimal time and stretch tradeoff.

4.2.1 Static Hopset Construction

In this section we outline the (static) hopset construction of Elkin and Neiman [41][3] (which is similar to [58]). We will later explain how we can make modifications that allows us to maintain a *similar* hopset dynamically.

Given a weighted graph $G = (V, E, w)$, an integer $1 \leq k \leq \log\log n$ and $\rho > 0$, we show the construction a (β, ϵ)-hopset of size $O(n^{1+\frac{1}{2^k-1}})$ and hopbound $\beta = O((\frac{k+1/\rho+1}{\epsilon})^{k+1/\rho+1})$.

We define sets $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{k+1/\rho+1} = \emptyset$. Let $\nu = \frac{1}{2^k-1}$. Each set A_{i+1} is obtained by sampling each element from A_i with probability $q_i = \max(n^{-2^i \cdot \nu}, n^{-\rho})$, where ρ is a parameter that determines a tradeoff between hopbound and running time.

Fix $0 \leq i \leq k + 1/\rho + 1$. Then, for every vertex $u \in A_i \setminus A_{i+1}$, let $p(u) \in A_{i+1}$ be the node of A_{i+1}, which is closest to u. We define a *bunch* of u to be a set $B(u) := \{v \in A_i : d(u, v) < d(u, A_{i+1})\}$. Also, we define a set $C(v)$, called the *cluster* of $v \in A_i \setminus A_{i+1}$, defined as $C(v) = \{u \in V : d(u, v) < d(u, A_{i+1})\}$. Note that if $v \in B(u)$ then $u \in C(v)$, but the converse does not necessarily hold. The way we define the bunches and clusters here follows [41], but differs slightly from the definitions in [88, 85], where each vertex has a separate bunch and cluster defined for each level i (and stores the union of these for all levels).

The clusters are *connected* in a sense that if a node $u \in C(v)$ then any node z on the shortest path between v and u is also in $C(v)$ since $d(z, v) \leq d(u, v) < d(u, A_{i+1})$.

Claim 4.2.1. *Let $u \in C(v)$, and let $z \in V$ be on a shortest path between v and u. Then $z \in C(v)$.*

Proof. Let $v \in A_i$. If $z \notin C(v)$ then by definition $d(z, A_{i+1}) \leq d(v, z)$. On the

[3]In [41] two algorithms with different sampling probabilities are given, where one removes a factor of k in the size. This factor does not impact our overall running time, so we will use the simpler version.

other hand, since z is on the shortest path between u and v: $d(u, A_{i+1}) \leq d(z, u) + d(z, A_{i+1}) \leq d(u, z) + d(z, v) = d(u, v)$, which contradicts the fact that $u \in C(v)$. $\square$

As we will see, this property is important for bounding the running time. The hopset is then obtained by adding an edge (u, v) for each $u \in A_i \setminus A_{i+1}$ and $v \in B(u) \cup \{p(u)\}$, and setting the weight of this edge to be $d(u, v)$. These distance can be computed by maintaining the clusters. As we will see in maintaining the clusters (rather than bunches) we scan more edges than what is stored in the hopset. Hence the update time of our dynamic algorithms is determined by the number of clusters a node belongs to, rather than the size of the hopset. This is because unlike an emulator, for maintaining the distances using a hopset, we also need to consider the edges in G, and the small hopbound is the key to efficiency rather than sparsity.

Theorem 4.2.2 ([41]). *There is an algorithm that given a weighted and undirected graph $G = (V, E)$, and $2 \leq k \leq \log\log n - 2$, $\frac{2}{2^k - 1} < \rho < 1$ computes a $(\beta, 1+\epsilon)$-hopset of size $O(n^{1+\frac{1}{2^k-1}})$, where $\beta = O((\frac{k+1/\rho}{\epsilon})^{k+1/\rho+1})$. It runs in $O(\frac{n^\rho}{\rho}(m + n\log n))$ expected time.*

We do not directly use this static construction, so we do not give proofs of stretch and hopbound guarantees. As we shall see, the stretch analysis of our decremental hopset uses a similar structure, but combined with stretch arguments needed for scaling and monotone ES tree techniques.

Size. We rely on the fact that the hopset size is not denser than the original graph. This is why our main bounds do not hold for very sparse graphs. For analyzing the size, [41] argues that for each $u \in A_i \setminus A_{i+1}$ we have $E[|B(u)|] \leq 1/q_i$ for the following reason: Consider an ordering of vertices in A_i based on their distance to u. By definition, size of $B(u)$ is bounded by the number of vertices in this ordering until the first vertex in A_{i+1} is visited. This corresponds to a geometric random variable with parameter q_i and thus in expectation it is $1/q_i = n^{2^i \nu}$. Hence for all i the number of edges added is in expectation

$$\sum_{i=1}^{k-2} E[|A_i|] n^{2^i \cdot \nu} = O(kn^{1+\nu}).$$

Efficient Construction via Modified Dijsktra's algorithm. For an efficient construction of these hopsets, [41] used the *modified Dijsktra's algorithm*, which was proposed by Thorup-Zwick [88]. This algorithm the bunches for level i can be constructed in $O(m+n\log n)/q_i$. At a high-level this is done by making a modification to Dijkstra. In the original Dijkstra for each source $u \in A_i \setminus A_{i+1}$, at each iteration we consider an unvisited vertex v, and "relax" each incident edge (v, z) by setting $d(u, z) = \min\{d(u, z), d(u, v) + w(v, z)\}$. But in the modified algorithm this is done only if $d(u, v) + w(v, z) < d(z, A_{i+1})$. In other words each node z only "participates" in a shortest-path exploration from a source u only if $z \in B(u)$. Note that if $z \in B(u)$, all the nodes on the shortest path between u and z are considered. Since $|B(u)| \leq \frac{1}{q_i}$, this allows us to bound the running time by $O(mn^{\rho})$.

Related objects. The hopset of [41] has some structural similarities with emulators of [89]. One main difference, as we discussed, is that the sampling probabilities are adjusted (bounded by $n^{-\rho}$) to allow for efficient construction of these hopsets in various models, at the cost of slightly weaker size/hopbound tradeoffs. We also need these adjustment for our efficient decremental algorithms.

4.2.2 Maintaining Restricted Hopsets Dynamically

Before we give our full hopset construction, we show how we can construct a d-restricted hopset, i.e. a hopset that guarantees hopbounded paths only between nodes within distance d. We then use this algorithm to construct a sequence of d-restricted hopsets for exponentially increasing values of d, at each step using the hopsets constructed so far. In order to maintain a d-restricted hopset dynamically, we start with a decremental algorithm of Roditty and Zwick [85]. Their techniques allow us to maintain the clusters and bunches as defined in Section 4.2.1. However we need to modify their algorithm in several ways. The first modification is adjusting sampling probabilities to match the probabilities we gave in Section 4.2.1. Note that while the clusters we consider are slightly different than what was used in [85] (even if we ignore the difference in sampling probabilities), we use a *subset* of the clusters defined in [85] uses.

By extending their algorithm and analysis to our setting, we can maintain a d-restricted hopset decrementally in $\tilde{O}(dmn^{\rho})$ total time, where $0 < \rho < \frac{1}{2}$ is a parameter that balances the tradeoff between the hopbound and time as discussed in Section 4.2.1. This means we can efficiently maintain a d-restricted hopset, when d is small. However, for large d, such running time is prohibitive.

The main new technical component of our construction is providing a hierarchical algorithm that iteratively constructs restricted hopsets on a sequence of *scaled graphs*. Next, we explain this hierarchical construction.

Path doubling. Our algorithm maintains a sequence of graphs $H_0, \dots, H_{\log W}$ with the following property. For each $0 \leq j \leq \log W$, $\bigcup_{r=0}^{j} H_r$ is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. Note that for $0 \leq j \leq \log \beta$ we can set $H_j = \emptyset$, since G covers these scales (w.l.o.g the weights in G are at least 1, so if $d_G(u, v) \leq \beta$, there is a shortest path between u and v of at most β hops). To maintain the graphs H_i, we prove the following lemma.

Lemma 4.2.3. *Consider a graph $G = (V, E, w)$ subject to edge deletions. Assume that we have maintained $\bar{H}_j := H_1, ..., H_j$, which is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. Then, there is a data structure, that given the sequence of changes to G and $\bar{H}_j$, maintains a graph H_{j+1}, such that $\bar{H}_j \cup H_{j+1}$ is a $(2^{j+1}, \beta, (1+\epsilon)^{j+1})$-hopset of G.*

The data structure can be maintained in $\tilde{O}((m + \Delta)n^{\rho} \cdot \frac{\beta}{\epsilon})$ total time, where m is the initial size of G, Δ is the total number of edges inserted to $\bar{H}_j$ over all updates,

$\beta = (\frac{1}{\epsilon \cdot \rho})^{O(1/\rho)}$, and $0 < \epsilon < 1, \rho < \frac{1}{2}$ are parameters.

Note that the lemma does not hold for *any* restricted hopset, and we need to use special properties of our construction to prove this.

In our construction we use $G \cup_{r=0}^{j-1} H_r$ to construct H_j. Note that by our assumption it suffices if H_j is a hopset for paths of length in the range $[2^{j-1}, 2^j)$, since shorter paths are already taken care of by $H_1, \ldots, H_{j-1}$. The important observation is that each path π of length $\in [2^{j-1}, 2^j)$ in G can be approximated (within a $(1+\epsilon)^{j-1}$ factor) by a path of $2\beta + 1$ hops in $G \cup_{r=0}^{j-1} H_r$. This follows from the fact that any such π can be obtained by concatenating paths π_1, π_2 and π_3, where π_1 and π_3 have length at most 2^{j-1} (so we can apply the property of a 2^{j-1}-restricted hopset) and π_2 consists of a single edge. Hence, a subproblem that we need to solve for each distance scale $[2^{j-1}, 2^j)$ is computing a hopset for distances between $[2^{j-1}, 2^j)$, knowing that the length of each such shortest path in G can be approximated by a path in $G \cup_{r=0}^{j-1} H_r$ consisting of at most $2\beta + 1$ hops.

The path doubling idea has been used in hopset constructions in distributed/-parallel models (e.g. [25, 40, 41]), but to the best of our knowledge this is the first use of this approach in a dynamic setting. While implementing using this idea in parallel/distributed settings is relatively straight-forward, it is not immediately clear how to utilize this in dynamic settings. To do this we need to maintain the hopsets on a sequence of scaled graphs. We first review a scaling idea and then define this sequence.

Scaling. We review a scaling algorithm widely used in dynamic settings (e.g. [15, 16, 55] *repeatedly and iteratively* during the process of adding hopset edges.

This idea can summarized in the following scaling scheme due to Klein and Subramanian [64], which, roughly speaking, says that finding shortest paths of length $\in [2^{j-1}, 2^j)$ and at most ℓ hops, can be (approximately) reduced to finding paths of length at most $O(\ell)$ in a graph with in integral weights. This is done by a rounding procedure that adds a small additive factor $\frac{\epsilon_0 w(e)}{\ell}$ to each edge e. Then for a path of ℓ hops the overall stretch will be $(1+\epsilon_0)$.

Lemma 4.2.4 ([64]). *Let $G=(V,E,w)$ be a weighted undirected graph. Let $R \geq 0$ and $\ell \geq 1$ be integers and $\epsilon_0 > 0$. We define the* scaled graph *to be a graph $\textsc{Scale}(G,R,\epsilon_0,\ell) := (V,E,\hat{w})$, such that $\hat{w}(e) = \lceil \frac{w(e)}{\eta(R,\epsilon_0)} \rceil$, where $\eta(R,\ell) = \frac{\epsilon_0 R}{\ell}$.*

Then for each edge $e \in E$ we have $\hat{w}(e) \leq w(e) + \epsilon_0 R$, and for any path π in G such that π has at most ℓ hops and weight $R \leq w(\pi) \leq 2R$, we have

- $\hat{w}(\pi) \leq \lceil 2\ell/\epsilon_0 \rceil$,
- $w(\pi) \leq \eta(R,\epsilon_0) \cdot \hat{w}(\pi) \leq (1+\epsilon_0) w(\pi)$.

By using the above scaling in the construction of the data structure of Lemma 4.2.3, we can effectively reduce the problem that the data structure is solving to the problem of maintaining a $O(\beta)$-restricted hopset in a graph with integral weights in a dynamic setting. Note that we set $\beta = \text{poly}\log n$. Since there are edge insertion into the hopset, and hence each scaled graph, there are further challenges in how these ideas can be combined in decremental settings. We will explain later that for handling edge insertion we need to use another data structure called the monotone ES tree ([56]). We need to show that by combining the estimates from these different data structures we still get a hopset with the desired properties.

Handling insertions. While the algorithm of Roditty and Zwick [85] only works in the decremental setting, in our case we need to extend it to handle edge *insertions*. This is because we run it on a graph $G \cup \bigcup_{r=0}^{j-1} H_r$ (after applying scaling of Lemma 4.2.4). While edges of G can only be deleted, new edges may be added to some hopsets H_r.

We deal with this issue as follows. The algorithm of Roditty and Zwick [85] decrementally maintains a collection of single-source shortest path trees (up to a bounded depth) using the Even-Shiloach algorithm (ES-tree) [87]. We modify the algorithm by effectively replacing each ES-tree, by a *monotone* ES-tree proposed by [55, 56]. The monotone ES-tree, in addition to supporting edge deletions, also supports edge insertion operation in a limited way. Namely, whenever an edge (u, v) is inserted and the insertion of the edge causes a distance decrease in the tree, we do not update the currently maintained distance estimates. This change keeps the running time roughly the same as in the decremental setting.

The main challenge here lies in analyzing the hopset stretch. While [55] analyzed the stretch incurred by running monotone ES-trees on a hopset, the proof relied on the properties of the specific hopset used in their algorithm. Since the hopset we use is quite different, we need a different analysis, which combines the static hopset analysis, with the ideas used in [55], and also take into account the stretch incurred due to the fact that the restricted hopsets are maintained on the scaled graphs. In our final algorithm, we need to run this restricted hopset algorithm on the sequence of scaled graphs, so that we can utilize the smaller scale hopsets in a hierarchical way to get our improved update time.

Putting it together. We now go back to the setting of Lemma 4.2.3. Given a 2^j-restricted hopset $\bar{H}_j = H_1 \cup ... \cup H_j$ for distances up to 2^j, we can now construct a graph G^j by applying the scaling of Lemma 4.2.4 to $G \cup \bar{H}_j$ and setting $R = 2^j$, $\ell = 2\beta + 1$. Then we can efficiently maintain an ℓ-restricted hopset on G^j. Then by Lemma 4.2.3 we can use this to update H_{j+1}. Importantly, ℓ is independent of R, and thus we can eliminate the factor R to get $\tilde{O}(\beta m n^{\rho})$ total update time.

Our final algorithm is a hierarchical construction that maintains the restricted hopsets on scaled graphs and the original graph simultaneously. Since we are maintaining hopsets on scaled graphs, we will lose small factors in the stretch, but we can show that this has little impact on our overall hopbound/update time tradeoff. For obtaining our near-optimal time and hopbound tradeoff, we need to carefully combine the ideas described and show that the monotone ES tree ideas can be applied to these specific insertions.

We rely on a threefold inductive construction and analysis that combines the pieces we have described.

1. An induction on the i, the iterations of the base hopset, which controls the sampling rate and the resulting size and hopbound tradeoffs.

2. An induction on the scale j, which allows us to cover all ranges of distances $[2^j, 2^{j+1}]$ by maintaining distances in the appropriate scaled graphs.

3. An induction on time t that allows us to handle insertions by using the estimates from previous updates in order to keep the distances monotone.

The overall stretch argument needs to deal with several error factors in *addition to* the base hopset stretch. First, the error incurred by using hopsets for smaller scales, which we deal with by maintaining our hopsets by setting $\epsilon' = \frac{\epsilon}{\log n}$. This introduces polylogarithmic factors in the hopbound. The second type of error comes from the fact that the restricted hopsets are maintained for scaled graphs, which implies the clusters are only approximately maintained on the original graph. This can also be resolved by further adjusting ϵ'. Finally, since we use an idea similar to the monotone ES tree of [55, 56], we may set the level of nodes in each tree is to be larger than what it would be in a static hopset. But we argue that the specific types of insertions in our algorithm will still preserve the stretch. At a high-level this is because in case of a decrease we use an estimate from time $t-1$, which we can show inductively has the desired stretch.

We note that while the use of monotone ES tree and the structure of the clusters in our construction are similar to [55], our algorithm has several important differences.

Other than using a different and more general base (static) hopset, we use a different approach to maintain the hopset efficiently by using path doubling and maintaining restricted hopsets on the *scaled graphs*. Among other things, in [55] a different notion of approximate ball is used that is rather more lossy[4] with respect to the hopbound/stretch tradeoffs. By maintaining restricted hopsets on scaled graphs, we are also effectively preserving approximate balls in the original graph, but as explained above the error accumulation combines nicely with the path-doubling idea.

Finally, [55] uses an edge sampling idea to bound the update time, which we can avoid by utilizing the sampling probability adjustments in [41], and the ideas in [85].

4.2.3 Decremental Approximate Distances

Our algorithms for maintaining approximate distances under edge deletions are as follows. First, we maintain a $(\beta, 1+\epsilon)$-hopset. Then, we use the hopset and Lemma 4.2.4 to reduce the problem to the problem of approximately maintaining short distances from a single source. For our application in MSSP and APSP the best update time is obtained by setting the hopbound to be polylgarithmic whereas for SSSP the best choice for is $\beta = 2^{\tilde{O}(\sqrt{\log n})}$. Using this idea for SSSP and MSSP mainly involves using the monotone ES tree ideas described earlier. Maintaining the APSP distance oracle is slightly more involved but uses the same techniques as in our restricted hopset algorithm. This algorithm is based on maintaining Thorup-Zwick distance oracle [88] more efficiently. At a high-level, we maintain *both* a $(\beta, 1+\epsilon)$-hopset and Thorup-Zwick distance oracle simultaneously, and balance out the time required for these two algorithms. The hopset is used to improve the time required for maintaining the distance oracle from $O(mn)$ (as shown in [85]) to $O(\beta m n^{1/k})$, but with a slightly weaker stretch of $(2k-1)(1+\epsilon)$. Querying distances is then the same as in the static algorithm of [88], and takes $O(k)$ time.

[4]This is also on reason we get an improvement in amortized single-source shortest path update time.

4.3 Decremental Hopset

In this section we provide our decremental hopset algorithms. Our goal is to implement the hopset algorithm described in Section 4.2.1 dynamically. In Section 4.3.1, we explain how we can adapt ideas by Roditty-Zwick [85] to obtain an algorithm for computing a d-restricted hopset. The total running time of this algorithm is $O(dmn^{\rho})$ (where $\rho < 1$ is a constant), which is undesirable for large values of d. We will then improve the running time to $\tilde{O}(mn^{\rho})$ using scaling and path-doubling ideas. Recall that our algorithm maintains a sequence of graphs $H_0, \ldots, H_{\log W}$, where for each $1 \leq j \leq \log W$, $H_0 \cup \ldots \cup H_j$ is a 2^j-restricted hopset of G. Instead of computing each H_j separately, we use $G \cup \bigcup_{r=0}^{j-1} H_r$ to construct H_j. We observe that at the cost of some small approximation errors, any path of length $\in [2^{j-1}, 2^j)$ in G can be approximated by a path of at most $2\beta + 1$ hops in $G \cup \bigcup_{r=0}^{j-1} H_r$. To use this idea we will prove the following main lemma as a building block for our final hopset.

Lemma 4.3.1. *Consider a graph $G = (V, E, w)$ subject to edge deletions. Assume that we have maintained $\bar{H}_j := H_1, ..., H_j$, which is a $(2^j, \beta, (1+\epsilon)^j)$-hopset of G. Then, there is a data structure, that given the sequence of changes to G and $\bar{H}_j$, maintains a graph H_{j+1}, such that $\bar{H}_j \cup H_{j+1}$ is a $(2^{j+1}, \beta, (1+\epsilon)^{j+1})$-hopset of G.*

The data structure can be maintained in $\tilde{O}((m + \Delta)n^{\rho} \cdot \frac{\beta}{\epsilon})$ total time, where m is the initial size of G, Δ is the total number of edges inserted to $\bar{H}_j$ over all updates, $\beta = (\frac{1}{\epsilon \cdot \rho})^{O(1/\rho)}$, and $0 < \epsilon < 1, \rho < \frac{1}{2}$ are parameters.

There are two main challenges that we need to address for proving this lemma. First, we would like to make the running time independent of the scale bound 2^j, which is what we would get by directly using the algorithm of [85]. To that end, we are going to run our algorithm on a rescaled graph, which would allow us to only maintain distances up to depth $O(\beta/\epsilon)$. This relies on having the 2^j-restricted hopset $\bar{H}_j$, which allows us to maintain the hopset $\bar{H}_{j+1}$. Second, while G is undergoing deletions, H_j may be undergoing edge *insertions*. In Section 4.3.2 we explain how such insertions can be handled using the monotone ES tree algorithm (based on [55]). In Section 4.3.3 we use the properties of this algorithm to prove Lemma 4.2.3.

4.3.1 Maintaining a Restricted Hopset

In this section, our goal is to maintain a decremental *restricted* hopset. We start by adapting the decremental algorithm by [85] that maintains the Thorup-Zwick distance oracles [88] with stretch $(2k-1)$ for pairs of nodes within distance d in $\tilde{O}(dmn^{1/k})$ total time, but instead use their techniques for constructing hopsets. In the next sections, we give a variant of this algorithm that also handles insertions.

In order to turn their algorithm into a restricted hopset algorithm, we make two modifications. First, we change the sampling probabilities based on the hopset algorithm described in Section 4.2.1. Second, in addition to computing clusters we also construct the hopset by adding the corresponding edges. We argue that by choosing the parameter appropriately this extension leads to a $(d, \beta, 1+\epsilon)$-hopset, with update time $\tilde{O}(dmn^{\rho}\beta)$.

Since this is slow for large value of d, in Section 4.3.2, we give a new hopset algorithm. First, we combine the construction of [85] with the monotone ES tree ideas by [55] to generalize this construction to handle certain insertions. We will then run this algorithm on a sequence of scaled graphs, and show how our new hopset can be maintained efficiently using the previously added hopset edges.

First, we briefly review the algorithm of [85] here. A similar algorithm will also presented in Section 4.3.2, Algorithm 7, but with the additional property that certain types of insertions are also taken into account.

Recall the hopset algorithm discussed in Section 4.2.1. We sample sets $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{k+1/\rho+1} = \emptyset$ initially. The sets remain unchanged during the updates. Next, we need to maintain values $d(v, A_i), 1 \leq i \leq k + 1/\rho + 1$ for all nodes $v \in V$.

This can be performed by computing a shortest path tree rooted at a dummy node s_i connected to all nodes in A_i. We denote the estimate obtained by maintaining this distance by $L(v, A_{i+1})$, and let $\hat{d} = (1+\epsilon)d$. We use a well-known algorithm to maintain a single-source shortest path trees rooted at a source node up to depth d in $O(md)$ total time, called an Even-Shiloach [87, 62] tree. We will review a variant of this algorithm (that also handles insertions) in Section 4.3.2. We can maintain these shortest path trees from the dummy nodes up to depth $\hat{d}$ in $O(\hat{d}m)$ total update time. The pivots $p(v), \forall v \in V$ can also be maintained in this process.

Maintaining the clusters based on [85]. The more involved step is maintaining the clusters. At a high-level, the idea is to maintain Even-Shiloach [87] trees rooted at each node $z \in A_i \setminus A_{i+1}$ to compute $C(z)$. Recall that for $z \in A_i \setminus A_{i+1}$ we have $v \in C(z)$ if and only if $d(z, v) < d(v, A_{i+1})$. The algorithm of [85] can be summarized as follows (see Algorithm 7 for a similar algorithm that also handles insertions): After each deletion, for each node v and the cluster centers z we first check whether the distance $d(z, v)$ has increased. If $d(z, v) \geq d(v, A_{i+1})$, v will be removed from $C(z)$. The more subtle part is adding nodes to new clusters. For each $0 \leq i < k$, we define a set X_i consisted of all vertices whose distance to A_i is increased as a result of a deletion, but where this distance is still at most $\hat{d}$. The sets X_i can be computed while maintaining $d(v, A_i)$.

In order to bound the running time, we use the same modification as in the modified Dijkstra algorithm [85], which allows us to bound the number of shortest path trees that each node belongs to. However there are few challenges in using these ideas. First, while in the static algorithm each node overlaps with at most $\tilde{O}(n^{\rho})$ clusters, in dynamic settings this holds at any point in time but does not immediately hold for a sequence of updates, since nodes keep on changing clusters. This is how [85] handles this:

Note that a node v would join $C(w)$ only after an increase in $d(v, A_{i+1})$. Using this observation, after each deletion for every $v \in X_{i+1}, z \in B_i(u) \setminus B_i(v)$, and each edge $(u, v) \in E$ we check if $d(z, u) + w(u, v) < d(v, A_{i+1})$. If yes, then v joins $C(z)$. We push v to a priority queue $Q(z)$ with key $d(z, u) + w(u, v)$. If v was already in the queue the key will be updated if this distance is smaller than the existing estimate. In this case we *mark* v. The marked nodes join clusters z, but there may be other nodes that also need to join $C(z)$ as a result of this change.

Hence after this initial phase, for each $z \in A_i \setminus A_{i+1}$ where $Q(z) \neq \emptyset$, we run the modified Dijkstra's algorithm. Recall that in the modified Dijkstra's algorithm when we explore neighbors of a node x, we only relax an edge (x, y) if $d(x, y) + w(x, y) < d(x, A_i)$. Then [85] show that this process correctly maintains the clusters. We then repeat this process for all the $k+1/\rho+1$ iterations. We add a hopset edge between each $z \in A_i \setminus A_{i+1}$ and all nodes $v \in C(z)$ and set the weight of this edge to $w(v, z) = d_G(v, z)$.

Lemma 4.3.2. *For every $v \in V$ and $0 \leq i \leq k + 1/\rho + 1$, the expected total number of times the edges incident on v are scanned over all trees for each $w \in A_i$ (i.e. trees on $C(w)$) is $O(\hat{d}/q_i)$, where q_i is the sub-sampling probability.*

Proof. Let $w \in A_i \setminus A_{i+1}$. The edges of a node $v \in V$ is scanned when v joins $C(w)$, and any given time $d(v, w)$ is increased until v leaves $C(w)$. We start by analyzing the total cost of joining new clusters. Recall that $C(w) = \{v \in V : d(v, w) < d(w, A_{i+1})\}$. Since we are in a decremental setting, v can join $C(w)$ only when $d(w, A_{i+1})$ increases, and this can happen at most $\hat{d}$ times *per tree*. As in the static setting, at any time, v joins at most $\tilde{O}(1/q_i)$ trees, since the number of clusters v belongs to is dominated by a geometric random variable with parameter q_i. We will use a similar argument for analyzing the total number of clusters each node belongs to over time. Hence the total time for nodes joining new clusters is $\tilde{O}(\hat{d}m/q_i)$. Next, we consider the case when after the deletion the distance between v and the center increases. This will let us bound the number of times the edges incident on v are scanned for a tree rooted at some node in A_i. Let $d_t(w, v)$ denote the distance between v and w at time t (after t deletions), and let $C_t(w)$ denote the cluster rooted at w at time t. We bound the number of indices t for which $v \in C_t(w)$ and $d_t(w, v) < d_{t+1}(w, v)$. Let $w_{t,1}, w_{t,2}, ...$ be the sequence of nodes in A_i sorted based on their distance from v at time t. Ties will be broken by ordering based on pairs $(d_t(v, w), d_{t+1}(v, w))$, i.e. nodes with the same distance from v at time t will be sorted based on their distance at time $t + 1$. This ensures that if $d_t(v, w_{t,j}) < d_{t+1}(v, w_{t,j})$, then $d_t(v, w_{t,j}) < d_{t+1}(v, w_{t+1,j})$. Same as before $\Pr[v \in C_t(w_{t,j})] \leq (1 - q_i)^{j-1}$, since $v \in C_t(w_t, j)$ only if for all $j' < j$ we have $w_{t,j'} \in A_i \setminus A_{i+1}$. Let $I = \{(t, j) \mid d_t(v, w_{t,j}) < d_{t+1}(v, w_{t,j}) \leq \hat{d}\}$. Then since edges incident to v are scanned only if their distance increases, the expected number of times they are scanned over all trees rooted at centers in A_i is at most $\sum_{(t,j)} \Pr[v \in C_t(w_{t,j})]$. Also, by definition for a fixed j there can be at most $\hat{d}$ pairs of form (t, j). In other words, the distance to the j-th closest vertex can increase at most $\hat{d}$ times, and hence,

$$\sum_{(t,j)} \Pr[v \in C_t(w_{t,j})] \leq \hat{d} \sum_{j \geq 1} (1 - q_i)^{j-1} \leq \hat{d}/q_i.$$

□

By combining the analysis of the modified Dijkstra algorithms of [88], Theorem 4.2.2, and Lemma 4.3.2, we see that in iteration i, edges incident to each node are scanned at most $O(d/q_i)$ times in total. Using a standard argument as in [87], the total update time is $\tilde{O}(md/q_i)$ (see Lemma 4.3.4 for a more detailed description of this time of argument). Hence we show that a d-restricted hopset with the following guarantees can be constructed:

Theorem 4.3.3. *Fix $\epsilon > 0, k \geq 2$ and $\rho \leq 1$. Given a graph $G = (V, E, w)$ with integer and polynomial weights, subject to edge deletions we can maintain a $(d, \beta, 1+\epsilon)$-hopset, with $\beta = O\left((\frac{1}{\epsilon} \cdot (k + 1/\rho))^{k+1/\rho+1}\right)$ in $O(d(m + n^{1+\frac{1}{2^k-1}})n^\rho)$ total time. The algorithm works correctly with high probability.*

Next, we use this algorithm for obtaining an efficient algorithm for maintaining a sequence of restricted hopsets. Our final algorithms needs to handle edges insertion (from smaller scales). For handling these insertions we need to use a different approach that is similar to the monotone ES tree ideas in [55]. We first start describe the algorithm of [55] for a single-source. Then we explain how their algorithm, combined with [85] can be used to maintain restricted hopsets with certain insertions.

4.3.2 New Hopsets with Improved Running Time.

The algorithm in Theorem 4.3.3 is too slow. Therefore in the rest of this section we described how we can get improved running time using a hierarchical construction of restricted hopsets on a sequence of scaled graphs. As explained one idea is that we can add hopset edges for smaller scales and use the added edges in computing distances for larger scales. Before describing our main algorithm, we review the monotone ES tree technique proposed by [55]. Note that the correctness (stretch) argument depends on other components of our hopset analysis and will be covered later in Theorem 4.3.7. We first explain the algorithm and running time for maintaining a single-source shortest path based on montone ES tree, and later explain how it is combined with the restricted hopset algorithm in the previous section.

We are going to use the algorithm of [85], described earlier, in combination with the monotone ES tree, in our main algorithm in 7) and in presence of edge insertions. However, before explaining how these two algorithms are combined, we review the algorithm of [85] for maintaing clusters, and how it can be used to maintain the hopset described in Section 4.2.1. As discussed, we will see that the update time for maintaining this specific construction is larger than desired. We will later explain how a new hopset, consisted of a series of restricted hopsets on scaled graphs lead to improved update time.

Handling edge insertions. In this section, we explain the monotone ES tree idea and how it can be used for maintaining single-source shortest path up to a given depth D. In Section 4.3.1 we explain how this idea can be used in maintaining a restricted hopset. Using the monotone ES tree ideas may impact the stretch, and clearly do not apply to all types of insertions but only for insertion of certain structural properties. In Section 4.3.3, we will prove that specifically for the insertions in our restricted hopset algorithm the stretch guarantee holds. We show how to handle edge insertions by using a variant of the monotone ES-tree algorithm [55] (and further used in the hopset construction of [56]). This algorithm is given as Algorithm 6. The idea in a monotone ES tree is that if an insertion of an edge (u, v) causes the level of a node v to decrease, we will not decrease the level. In this case we say the edge (u, v) and the node v are *stretched*. More formally, a node v is stretched when $L(v) > \min_{(x,v)\in E} L(x) + w(x, v)$.

We observe multiple properties of the monotone ES tree algorithm as observed by [55, 56] that will be helpful in analyzing the stretch later:

- The level of a node never decreases.
- Only an inserted edge can be stretched.
- While an edge is stretched, its level remains the same. In other words, a stretched edge is not going to get stretched again unless it is deleted.

Also observe that we never underestimate the distances. This is clearly true for any edge weights obtained by the rounding in Lemma 4.2.4. It is also easy to see this is true for the stretched edges for the following reason: For any node v, the algorithm maintains the invariant that $L(s, v) \geq min_{(x,v)\in E} L(s, x) + w(x, v)$. In other words, $L(s, v)$ is either an estimate based on rounding that is at least $d_G(s, v)$ or it is larger than such an estimate.

Lemma 4.3.4. *Algorithm 6 processes any sequence of updates in $O((m+\Delta)D)$ overall update time on a graph with m edges, where Δ is the number of edge insertions.*

Algorithm 6: Maintaining a monotone ES tree up to depth D on G. Note that edge deletion can be achieved by setting the edge weight to ∞.

```
1  Function Init(G, s, D)
2      $E := E(G) \cup \{e_v = (s, v) : v \in V(G) \setminus \{s\}, w(e_v) = D + 1\}$ /* This
           ensures that distances are maintained up to level $D$ */
3      for $v \in V$ do
4          $L(s, v) := 0$
5      for $v \in V$ do
6          Update($T(s), v$)
7  Function InsertEdge($T(s), (a, b), c$)
       /* Insert an edge in the tree rooted at $s$ */
8      $E := E \cup \{(a, b)\}$
9      $w(a, b) := c$
10     Update($T(s), b$)
11 Function Update($T(s), v$)
12     $upd := \min_{(x,v) \in E} L(s, x) + w(x, v)$
13     if $v = s$ or $L(s, v) \geq upd$ then
           /* Node $v$ is stretched. */
14         return
15     $L(s, v) := upd$
16     for $(v, y) \in E(G)$ do
17         Update($T(s), y$)
```

Proof. The running time analysis of the algorithm follows based on an argument similar to the analysis of the classic ES tree algorithm [87, 62]. The total time for updating distances up to a depth D is $O((m + \Delta)D)$: the edges incident to each node v are scanned any time level of v changes in an update. This happens when a node on the shortest path tree from source s to v undergoes a distance increase. Since distances can only increase, this can occur at most D times for nodes with depth at most D to the source. Furthermore, Δ is the number of added edges that also need to be scanned in each update. By summing over all edges incident to all nodes, including the added edges, the claim follows. ☐

Restricted hopsets with insertions. Next, in Algorithm 7, we show how the algorithm of [85]- also described in Section 4.3.1- is modified to handle insertions by combining it with the monotone ES tree algorithm (Algorithm 6) for each tree inside a cluster. Using this we can bound the update time, however proving the stretch is more involved, and depends on the specific structure of insertions over a sequence of graphs, and it does not hold for any set of insertions. We prove the stretch later when we describe our overall hopset that is consisted of a sequence of hopsets for different scales.

Algorithm 7: Monotone d-restricted hopset. Adaptation of [85].

```
1  Sample sets $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_{k+1/\rho+1} = \emptyset$.
2  Function UpdateClusters($G, E^-, E^+, d$)
3      Add edges $(x,y) \in E^+$ to any tree $T(z)$ s.t. $(x,y) \in T(z)$
4      for $i = 0$ to $k + 1/\rho + 1$ do
5          $\mathcal{C} = \emptyset$.
6          Remove edges $E^-$ from the ES tree maintaining distances $L(\cdot, A_{i+1})$
7          Remove hopset edges $(z,v)$, and remove $v$ from $T(z)$ where
             $L(z,v) \geq L(v, A_{i+1})$
8          $X_{i+1}$ := set of nodes whose distances to $A_{i+1}$ have increased due to removal
             of $E^-$, yet remained at most $d$
9          for $\forall v \in X_{i+1}$ do
10             for $(u,v) \in E$ do
11                 for $\forall z \in B_i(u) \setminus B_i(v)$ do
12                     if $L(z,u) + w(u,v) < L(v, A_{i+1})$ then
13                         $\mathcal{C} = \mathcal{C} \cup \{z\}$
14                         Relax($(Q(z), u, v)$)/* Update the estimate from $z$ to
                               $v$                                          */
15         for $\forall z \in \mathcal{C}$ do
16             Dijkstra($z$)
17     return $(E^-, E^+)$
18 Function Dijkstra($z$)
19     while $Q(z) \neq \emptyset$ do
20         $u$ = ExtractMin($Q(z)$)
21         $B(u) = B(u) \cup \{z\}$
22         for $\forall (u,v) \in E : z \notin B(v)$ do
23             if $L(z,u) + w(u,v) < L(v, A_{i+1})$ then
24                 Relax($Q(z), u, v$)/* Update the estimate from $z$ to $v$      */
25 Function Relax($Q(z), u, v$)
       /* Distances $L(z,v)$ for each tree $T(z)$ are maintained in $Q(z)$    */
26     $d' := L(z,v) + w(z,v)$
27     if $d' \leq d$ then
28         if $v \in Q(z)$ then
29             decrease-key($Q(z), v, d'$)
30         else if $L(z,u) > d'$ then
31             Insert($Q(z), v, d'$)
32         Add node $v$ to $T(z)$
33         InsertEdge($T(z), (z,v), d'$)/* As defined in Algorithm 6          */
34         $E^+ = E^+ \cup \{(z,v)\}$
35         Add $(z,v)$ to $E^-$ if $L(z,v)$ has increased.
```

By combining these two algorithm we can keep the running time the same as in Theorem 4.3.3 despite the insertions:

Theorem 4.3.5. *Assume that we are given a set of Δ updates including a set E^- of deletions and a set E^+ of insertions, and parameters d, and ϵ, w.h.p. the total update time of Algorithm 7 is $O((m + \Delta + n^{1+\frac{1}{2^k-1}})dn^{\rho})$.*

Proof sketch. The proof of this theorem is almost exactly the same as the proof of Theorem 4.3.3. We rely on Lemma 4.3.2 again to show the over a sequence of updates, for each iteration i, each node v is only in $\tilde{O}(1/q_i)$ clusters. Since monotone ES tree ensures that distances are not decreasing the level of a node, the number of times edges incident to v are scanned is still $\tilde{O}(d/q_i) = \tilde{O}(dn^{\rho})$. We now have $m + \Delta$ edges, and the theorem follows by summing overall nodes. □

We showed that we can handle a set of insertions within the same the running time. But as discussed, directly using algorithm 7 still does not lead to our desired update time. Therefore in the rest of this section we described how we can get improved running time by using this algorithm to maintain a hierarchical construction of restricted hopsets on a sequence of scaled graphs. As explained one idea is that we can add hopset edges for smaller scales and use the added edges in computing distances for larger scales. Once we specify the set of insertion into each of the *scaled graphs* considered, we will show that such insertions will also preserve the hopset stretch (with small polylogarithmic overhead) in the *original graph.*

Path doubling and scaling. We first state the path doubling idea more formally for a *static hopset* in the following lemma. However for utilizing this idea dynamically we need to combine it with other structural properties of our hopsets.

Lemma 4.3.6. *Given a graph $G = (V, E)$, $0 < \epsilon_1 < 1$, the set of $(\beta, 1 + \epsilon_1)$-hopsets $H_r, 0 \leq r < j$ for each distance scale $(2^r, 2^{r+1}]$, provides a $(1+\epsilon_1)$-approximate distance* *for any pair $x, y \in V$, where $d(x, y) \leq 2^{j+1}$ using paths with at most $2\beta + 1$ hops.*

Proof. We can show this by an induction on j. Let π be the shortest path between x and y on . Then π can be divided into two segments, where for each segment there is a $(1 + \epsilon_1)$-stretch path using edges in $G \cup \bigcup_{r=0}^{j-1} H_r$. Let $[x, z]$ and $[z', y]$ be the segments on π each of which has length at most 2^{j-1}. In other words, z is the furthest point from x on π that has distance at most 2^{j-1}, and z' is the next point on π. Then we have,

$$\begin{aligned} d^{(2\beta+1)}_{G \cup \bigcup_{r=1}^{j-1} H_r}(x, y) &\leq [d^{(\beta)}_{G \cup \bigcup_{r=1}^{j-1} H_r}(x, z) + w(z, z') + d^{(\beta)}_{G \cup \bigcup_{r=0}^{j-1} H_r}(z', y)] \\ &\leq (1 + \epsilon_1) d_G(x, z) + w(z, z') + (1 + \epsilon_1) d_G(z', y) \\ &\leq (1 + \epsilon_1) d_G(x, y) \end{aligned}$$

□

This implies that it is enough to compute $(2\beta+1)$-hop limited distances in restricted hopsets for *each* scale. For using this idea in dynamic settings we have to deal with some technicalities. We should show that we can combine the rounding with the modification needed for handling insertions. In Algorithm 7, we presented a restricted hopset construction obtained by combinatining of [85] with Algorithm 6 (monotone ES tree).

We define a scaled graph using Lemma 4.2.4 as follows: $G^j := \textsc{Scale}(G \cup \bigcup_{r=0}^{j} H_r, 2^j, \epsilon_2, 2\beta + 1)$. Here we set $R = 2^j, \ell = 2\beta + 1$, and ϵ_2 is a parameter that we tune later. We first describe the operations performed on this scaled graph. We then explain how we can put things together for all scales to get the desired guarantees.

The key insight for scaling $G \cup \bigcup_{r=0}^{j} H_r, 2^j$ is that we can obtain H_{j+1} by computing an $O(\ell)$-restricted hopset of G^j (using the algorithm of Lemma 4.2.3) and scaling back the weights of the hopset edges.

In addition to the graph G undergoing deletions, our decremental algorithm maintains for each $1 \leq j \leq \log W$:

- The set $\bar{H}_j = \bigcup_{r=0}^{j} H_r$, union of all hopset edges for distance scales up to $[2^j, 2^{j+1}]$.
- The scaled graphs $G^1, ..., G^j$.
- Data structure obtained by constructing an $O(\beta/\epsilon_2)$-restricted hopset on G^j by running Algorithm 7 for the appropriate parameter $\epsilon_2 < 1$. We denote this data structure by D_j.

The data structure D_j is maintained by running Algorithm 7 on G^j, and maintaining the clusters and hence the bunches $B(v)$ for all $v \in V$. Given D_j, we can maintain H_{j+1}, where the edge weights in clusters are assigned by computing approximate distances based on Algorithm 6 on each cluster as follows: In a tree rooted at a cluster center z, we set the weight w_j on an edge (z, v) to be $\min_{r=1}^{j-1} \eta(2^r, \epsilon_2) L_r(z, v)$, where $L_r(z, v)$ is the level of v on G^r after running the monotone ES tree up to depth $D = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$. We the maintain a restricted hopset on the scaled graph G^j, and by *unscaling* its weights we get H_{j+1}.

Once each data structure D_j is initialized with a graph, it can execute a single operation Update(E^-, E^+), which updates the maintained graph by removing the edges of E^- and adding edges E^+ by running Algorithm 7. The set E^- is the set the edges corresponding to nodes leaving clusters. The operation returns a pair of edges (E^-, E^+) that are edges that should be removed or added from D_j. Additionally, by multiplying these distance by $\eta(2^j, \epsilon_2)$ for the appropriate ϵ_2, we can recover a pair (H^-, H^+) of edge sets, where H^- is the set of edges that are removed from the hopset and H^+ is the set of edges added to the hopset as a result of the update. Note that a change in the weight of a hopset edge is equivalent to removing the edge and adding it with a new weight.

In Algorithm 8 we update the data structures described as follows: we run Algorithm 7 for distances bounded by $d = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ starting on $j = 0, ..., \log W$ in increasing order of j to compute hopset edges H_j. After processing all the changes in scaled graph G^j, we add the inserted edges to G^{j+1}. Then we process the changes in G^{j+1} by running the algorithm of Section 4.3.1 and repeat until all distance scales of covered. As explained, when the distances increase a node may join a new cluster which will lead to a set of insertions in H and in turn insertions in a sequence of graphs G^j. We use an argument similar to Lemma 4.3.2 on each scaled graph to get the overall update time. In a way we can see the added edges passed to each scale as a set of batch distance increases, between the corresponding endpoints. This means we are not exactly in the setting of [85] where only one deletion occurs at each time, but the exact same analysis as in Lemma 4.3.2 still holds.

Algorithm 8: Updating the hopset after deleting an edge e.

```
1 Input: $0 < \epsilon, 0 < \epsilon_2 < 1$, set $d = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$.
2 $(E^-, E^+) := (\{e\}, \emptyset)$
3 for $j = 0, \ldots, \lfloor \log W \rfloor$ do
4     $(E^-, E^+) :=$ UpdateClusters$(G^j, E^-, d, \epsilon)$/* Run Algorithm 7 on
          $G^j$                                                          */
5     Update $H_{j+1}$ by unscaling weights of $E^+$ and removing $E^-$ (Lemma 4.2.4)
          /* add edges for the next scale                                */
6     Update $G^{j+1}$ based on Lemma 4.2.4 to reflect changes to $H_{j+1}$
```

We summarized the algorithm obtained by maintaining this data structure over all scales in Algorithm 8. Note that we need to update both the restricted hopsets D_j on the scaled graphs and the hopset H_j obtained by scaling back the distances using Lemma 4.2.4.

Running time (proof of Lemma 4.2.3). We can now put all the steps discussed to maintain the data structure of Lemma 4.2.3. In particular, for obtaining a 2^{j+1}-restricted hopset, we maintain the data structure of Lemma 4.2.3 on G^j for each cluster rooted at a node $z \in A_i \setminus A_{i+1}$ and by setting $\ell = 2\beta + 1$. By using Lemma 4.3.4 and Theorem 4.3.3 to compute d-restricted hopsets for $d = O(\beta/\epsilon)$. When weights are polynomial we get the running time of $\tilde{O}(\frac{\beta}{\epsilon}(m + \Delta)n^{\rho})$, where Δ is the overall number of hopset edges added over all updates.

4.3.3 Hopset stretch

In this section, we first prove the stretch incurred for a single-scale by combining properties of the monotone ES-tree algorithm of Section 4.3.2 with the static hopset argument and the rounding framework. We will then show that by setting the appropriate parameters we can prove the overall stretch and hopbound tradeoffs described in Lemma 4.2.3.

In the following, we extend the static hopset argument to dynamic settings. We use the path doubling observation in Lemma 4.3.6 and properties of monotone ES tree described in Section 4.3.2 to prove the stretch incurred in each scale. We denote the stretch of $\bar{H}_j$ to be $(1 + \epsilon_j)$. Then for getting the final stretch and hopbound we will set the parameters $\epsilon_2 = \epsilon'$ (error incurred by rounding), and $\delta = \frac{\epsilon}{8(k+1/\rho+1)}$.

The stretch argument is based on a threefold induction on i, j-th scale, and time t. By fixing i, j, t, and a source node s, we show that there is a $(1 + \epsilon_j)$-stretch path between s and any other node with β hops (or if we are using previous scale $2\beta + 1$-hops) such that based each segment of this path has the desired stretch based on the inductive claim on one of these three factors. At a high level induction on i and j follows from static properties of our hopset. To show that bounded depth monotone ES tree maintains the approximate distances, we note that any segment of the path undergoing an insertion consistent of a single shortcut and the weight on such an edge is a distance estimate between its endpoints.

Theorem 4.3.7. *Given a graph $G = (V, E)$, assume that we have maintained a $(2^j, \beta, (1 + \epsilon_j))$-restricted hopset $\bar{H}_j$, and let H_{j+1} be the hopset obtained by running*

Algorithm 8 for any given $0 \leq \epsilon_2 < 1$ on $G \cup \bar{H}_j$. Fix $0 < \delta \leq \frac{1}{8(k+1/\rho+1)}$, and consider a pair $x, y \in V$ where $d_{t,G}(x,y) \in [2^j, 2^{j+1}]$. Then for $0 \leq i \leq k + 1/\rho + 1$, either of the following conditions holds:

1. $d^{((3/\delta)^i)}_{G \cup \bar{H}_{j+1}}(x,y) \leq (1+8\delta i)(1+\epsilon_j)(1+\epsilon_2)d_{t,G}(x,y)$ *or,*

2. *There exists $z \in A_{i+1}$ such that,*

$$d^{((3/\delta)^i)}_{G \cup \bar{H}_{j+1}}(x,z) \leq 2(1+\epsilon_j)(1+\epsilon_2)d_{t,G}(x,y).$$

Moreover, by running Algorithm 6 on G^{j+1} up to depth $\lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$, and applying the rounding in Lemma 4.2.4, we can maintain $(1+\epsilon_{j+1})$-approximate single-source distances up to distance 2^{j+2} from a fixed source s on G, where $1+\epsilon_{j+1} = (1+\epsilon_j)(1+\epsilon_2)^2(1+\epsilon)$ and $\beta = (3/\delta)^{k+1/\rho+1}$.

Proof. We use a double induction on i and time t, and also rely on distance computed up to scaled graph G^j. First, using these distance estimates for smaller scales, we argue that when we add an edge to $\bar{H}_{j+1}$ it has the desired stretch. Let $L_{t,j}(u,v)$ denote the level of node v in the tree rooted at u after running Algorithm 7 up to depth $D = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ on graph G^j. This proof is based on a cyclic argument: assuming we have correctly maintained distances up to a given scale using our hopset, we show how we can compute the distances for the next scale. In particular, we first assume that based on the theorem conditions we are given $\bar{H}_j$ and have maintained all the clusters and the corresponding distances in $G^1, ..., G^j$ with stretch $1+\epsilon_j$. This lets us analyze H_{j+1}. Then to complete the argument, we show how given the hopsets of scale $[2^j, 2^{j+1}]$, we can compute approximate SSSP distances for the next scale based on the monotone ES tree on G^{j+1}.

First, in the following claim, we observe that the edge weights inserted in the latest scale have the desired stretch by using our assumption that all the shortest path trees on each cluster on $G^1, ..., G^j$ are approximately maintained. We use such distance to add edges in each cluster to construct H_{j+1}, and we observe the following about the weights on these edges:

Observation 4.3.1. *Let $v \in B(u)$ such that $d_{t,G}(u,v) \leq 2^{j+1}$. Consider an edge (u,v) added to H_{j+1} after running Algorithm 7 on $G^1, ..., G^j$ for $D = \lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ rooted at node v. Let $w_{j+1}(u,v) := \min_{r=1}^{j} \eta(2^r, \epsilon_2) L_r(u,v)$, that is the unscaled edge weight. Then we have $d_{t,G}(u,v) \leq w_{j+1}(u,v) \leq (1+\epsilon_j)(1+\epsilon_2) d_{t,G}(u,v)$.*

This claim implies that the weights of hopset edges assigned by the algorithm correspond to approximate distance of their endpoints. Let $d_{t,j}(x,y) := \min_{r=1}^{j} \eta(2^r, \epsilon_2) L_{t,j}(x,y)$ which would be the estimate we obtain by for distance between x and y after scaling back distances on G^j. In other words this is the hop-bounded distance after running monotone ES tree on G^j and scaling up the weights.

For any time t and the base case of $i = 0$, we have three cases. If $y \in B(x)$ then edge (x,y) is in the hopset H_{j+1}, and by Observation 4.3.1 the weight assigned to this edge is at most $(1+\epsilon_j)(1+\epsilon_2) d_{t,G}(x,y)$. In this case the first condition of the theorem holds. Otherwise if $x \in A_1$, then $z = x$ trivially satisfies the second condition. Otherwise we have $x \in A_0/A_1$, and by setting $z = p(x)$ we know that there is an edge $(x,z) \in \bar{H}_j$ such that $d_{t,j}(x,z) \leq (1+\epsilon_2) d_{G \cup \bar{H}_j}(x,y)$ (by definition of $p(x)$ and using the same argument as above). Hence the second condition holds.

By inductive hypothesis assume the claim holds for i. Consider the shortest path $\pi(x,y)$ between x and y. We divide this path into $1/\delta$ segments of length at most $\delta d_{t,G}(x,y)$ and denote the a-th segment by $[u_a, v_a]$, where u_a is the node closest to x (first node of distance at least $a\delta d_{t,G}(x,y)$) and v_a is the node furthest to x on this segment (of distance at most $(a+1)\delta d_{t,G}(x,y)$).

We then use the induction hypothesis on each segment. First consider the case where for all the segments the first condition holds for i, then there is a path of $(3/\delta)^i(1/\delta) \leq (3/\delta)^{i+1}$ hops consisted of the hopbounded path on each segment. We can show that this path satisfies the first condition for $i+1$. In other words,

$$d^{((3/\delta)^{i+1})}_{t,G\cup\bar{H}_{j+1}}(x,y) \leq \sum_{a=1}^{1/\delta} d^{((3/\delta)^{i})}_{t,G\cup\bar{H}_{j+1}}(u_a,v_a)+d^{(1)}_{t,G}(v_a,u_{a+1}) \leq (1+8\delta i)(1+\epsilon_j)(1+\epsilon_2)d_{t,G}(x,y)$$

Next, assume that there are at least two segments for which the first condition does not hold for i. Otherwise, if there is only one such segment a similar but simpler argument can be used. Let $[u_l, v_l]$ be the first such segment (i.e. the segment closest to x, where u_l is the first and v_l is the last node on the segment), and let $[u_r, v_r]$ be the last such segment.

First by inductive hypothesis and since we are in the case that the second condition holds for segments $[u_l, z_l]$ and $[u_r, v_r]$, we have,

- $d^{((3/\delta)^{i})}_{t,G\cup\bar{H}_{j+1}}(u_l,z_l) \leq 2(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(u_l,v_l)$, *and,*
- $d^{((3/\delta)^{i})}_{t,G\cup\bar{H}_{j+1}}(v_r,z_r) \leq 2(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(u_r,v_r)$

Again, we consider two cases. First, in case $z_r \in B(z_l)$ (or $z_l \in C(z_r)$), we have added a single hopset edge $(z_r, z_l) \in \bar{H}_{j+1}$. Note that $d_{t,G}(z_r, z_l) \leq 2^{j+1}$, since $d_{t,G}(z_r, z_l) \leq d_{t,G}(x, y) \leq 2^{j+1}$. Hence by Observation 4.3.1 the weight we assign to (z_r, z_l) is at most $(1+\epsilon_2)(1+\epsilon_j)d_{t,G}(z_r, z_l)$.

On the other hand, by triangle inequality, and the above inequalities (which are based on the induction hypothesis) we get,

$$d^{(1)}_{\bar{H}_{j+1}}(z_l,z_r) \leq (1+\epsilon_2)(1+\epsilon_j)d_G(z_l,z_r) \tag{4.1}$$

$$\leq (1+\epsilon_2)(1+\epsilon_j)[d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(u_l,z_l)+d_G(u_l,v_r)+d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(z_r,v_r)] \tag{4.2}$$

By applying the inductive hypothesis on segments before $[u_l, v_l]$, and after $[u_r, v_r]$, we have a path with at most $(3/\delta)^i$ for each of these segments, satisfying the first condition for the endpoints of the segment. Also, we have a $2(3/\delta)^i + 1$-hop path going through u_l, z_l, z_r, v_r that satisfies the first condition for u_l, v_r.

Putting all of these together, we argue that there is a path of hopbound $(3/\delta)^{i+1}$ satisfying the first condition. In particular, we have (the subscript t is dropped in the following),

$$\begin{aligned}
d^{(3/\delta)^{(i+1)}}_{G\cup\bar{H}_{j+1}}(x,y) &\leq \sum_{a=1}^{l-1}[d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(u_a,v_a)+d^{(1)}_{G}(v_a,u_{a+1})]+d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(u_l,z_l) && (4.3)\\
&+d^{(1)}_{\bar{H}_{j+1}}(z_l,z_r)+d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(z_r,v_r)+d^{(1)}_{G}(v_r,u_{r+1}) && (4.4)\\
&+\sum_{a=r+1}^{1/\delta}[d^{((3/\delta)^{i})}_{G\cup\bar{H}_{j+1}}(u_a,v_a)+d^{(1)}_{G}(v_a,u_{a+1})] && (4.5)\\
&\leq (1+8\delta i)(1+\epsilon_j)(1+\epsilon_2)[d_G(x,u_l)+d_G(v_r,y)]+d_G(u_l,v_r) && (4.6)\\
&+(1+\epsilon_2)(1+\epsilon_j)[2d_G(u_l,z_l)+2d_G(z_r,v_r)] && (4.7)\\
&\leq (1+\epsilon_2)(1+\epsilon_j)[8\delta d_G(x,y)+(1+8\delta i)d_G(x,y)] && (4.8)\\
&\leq (1+8\delta(i+1))(1+\epsilon_2)(1+\epsilon_j)d_G(x,y) && (4.9)
\end{aligned}$$

In the first inequality we used the induction on i for each segment, and triangle inequality.

In the second inequality we are using the fact that nodes u_j, v_j for all j are on the shortest path between x and y in G, and we are replacing $d^{(1)}_{\bar{H}_{j+1}}(z_l,z_r)$ with inequality 4.2. In line 4.8 we used the fact that the length of each segment is at most $\delta\cdot d_G(x,y)$. Hence we have shown that the first condition in the lemma statement holds.

Finally, consider the case where $z_r \notin B(z_l)$. If $z_l \notin A_{i+2}$, we consider $z = p(z_l)$, where $z_l \in A_{i+2}$. We now claim that this choice of z satisfies the second lemma condition.

We have added the edge (z_l, z) to the hopset. Since $z_r \notin B(z_l)$, we have $d_{t-1,G}(z_l,p(z_l)) \leq d_{t-1,G}(z_l,z_r) \leq d_{t,G}(x,y) \leq 2^{j+1}$. Therefore we can use Observation 4.3.1 on the edge $(z_l, p(z_l))$.

$$\begin{aligned}
d^{(3/\delta)^{(i+1)}}_{G\cup\bar{H}_{j+1}}(x,y) &\le \sum_{a=1}^{l-1}[d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_a,v_a) + d^{(1)}_{G}(v_a,u_{a+1})] \\
&+ d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) + (1+\epsilon_2)(1+\epsilon_j)d^{(1)}_{\bar{H}_{j+1}}(z_l,z) \\
&\le (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d_G(x,u_l) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) + (1+\epsilon_2)(1+\epsilon_j)d_{\bar{H}_{j+1}}(z_l,z_r) \\
&\le (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d_G(x,u_l) + d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(u_l,z_l) \\
&+ (1+\epsilon_2)(1+\epsilon_j)[2d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(z_l,u_l) + d_G(u_l,v_r) + d^{(3/\delta)^i}_{G\cup\bar{H}_{j+1}}(v_r,z_r)] \\
&\le (1+8\delta i)(1+\epsilon_2)(1+\epsilon_j)d^{((3/\delta)^i)}_{G\cup\bar{H}_{j+1}}(x,v_r) + 6\delta(1+\epsilon_j)d_G(x,y) \\
&\le 2(1+\epsilon_2)(1+\epsilon_j)d_G(x,y)
\end{aligned}$$

In the last inequality we used the fact that we set $\delta < \frac{1}{8(k+1/\rho+1)}$ and thus $8\delta i < 1$. The only remaining case is when $z_\ell \in A_{i+2}$, in which case a similar reasoning follows by setting $z = z_l$.

Finally, we prove that after adding hopset edges H_{j+1} we can maintain approximate single-source shortest path distances from a given source s. This enables us to show that Observation 4.3.1 can be used for the next scale, i.e. that we can set the weights for the next scale by maintaining the clusters and $(1+\epsilon_{j+1})$ approximate distance rooted at a source s when we have $d(s,v) \in [2^{j+1}, 2^{j+2}]$, and hence close the inductive cycle in the argument.

We run the monotone ES tree algorithm (Algorithm 6) up to depth $\lceil\frac{2(2\beta+1)}{\epsilon}\rceil$ on all of the scaled graphs $G^1, ..., G^{j+1}$. We let set the distance estimate $d_{t,j+1}(s,v)$ to be $\min_r \eta(2^r,\epsilon_2)L_{t,r}(s,v)$ where $L_{t,r}(s,v)$ is the level of v on G^r on the ES tree up to depth $\lceil\frac{2(2\beta+1)}{\epsilon_2}\rceil$ rooted at s. Note that by running Algorithm 7 we are also maintaining the same distances on each cluster while also maintaining the nodes that leave and join a cluster. We analyze the estimate for any $v \in V$ such that $d_G(s,v) \le 2^{j+2}$. W.l.o.g. assume that $d(s,v) \in [2^{j+1}, 2^{j+2}]$, since if $d(s,v) \in [2^r, 2^{r+1}], r \le j$, we can use the same argument for the ES tree on G^r. Let $L_{t,j+1}(s,v)$ be the level of v in the monotone ES tree of G^{j+1} maintained up to depth $\lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$. Our goal is to show,

$$d_{t,j+1}(s,v) := \eta(2^{j+1}, \epsilon_2) L_{t,j+1}(s,v) \leq (1+\epsilon_{j+1}) d_{t,G}(s,v)$$

As discussed in Lemma 4.3.6, we consider the shortest path between s and v in G, and first divide it into two segments π_1 and π_2 each with length at most 2^{j+1}. Then divide each one of π_1 and π_2 into segments and consider the case by case inductive analysis as we did before for showing the stretch in H_{j+1}. We argue why the levels in tree rooted at s corresponding to π_1 have the desired stretch, then a similar reasoning with a factor 2 in the number of hops follows for π_2.

We use a case-by-case analysis similar to what we used for showing properties of $\bar{H}_{j+1}$, and consider the paths that were inductively constructed for each segment $\bar{H}_{j+1}$. Using that structure, we argue that in the monotone ES tree on G^{j+1} we can maintain the levels such that for each $0 \leq i \leq 1/\rho + k + 1$ one of the following conditions holds:

1. We have $d_{t,j+1}(s,v) \leq \eta(2^j, \epsilon_2) L(s,v) \leq (1+\epsilon_{j+1}) d_{t,G}(s,v)$, where this estimate corresponds to a path with $\beta_i = (3/\delta)^i$ in H_{j+1} (and hence G^{j+1}).

2. There exists $z_1 \in A_{i+1}$, such that $d_{t,j+1}(s,z_1) \leq \eta(2^{j+1}, \epsilon_2) L(s,z_1) \leq 2(1+\epsilon_j)(1+\epsilon_2) d_{t,G}(s,v)$ that corresponds a path with $\beta_i = (3/\delta)^i$ hops.

Then we can use this to show that after all iterations there either exists a path of depth at most $\lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ on G^{j+1} between s and v with stretch $(1+\epsilon_{j+1})$, or the monotone ES tree returns and estimate with this stretch.

We briefly review the different cases, same as before. First assume that we have $s \in B(v)$ for some iteration $1 \leq i \leq 1/\rho + 1 + k$, which implies we have added a hopset edge with weight w_{j+1} to $\bar{H}_{j+1}$. In this case the edge (s,v) was directly added to H_{j+1}. If edge (s,v) is stretched then we set $L_{t,G}(s,v) = L_{t-1,G}(s,v)$, and by induction on time we have

$$\eta(2^{j+1}, \epsilon_2) L_{t,j+1}(s,v) = \eta(2^{j+1}, \epsilon_2) L_{t-1,j+1}(s,v) \leq (1+\epsilon_{j+1}) d_{t,G}(s,v).$$

If this edge is not stretched then by Lemma 4.2.4 after scaling we get distance at most $(1+\epsilon_j)(1+\epsilon_2)^2 d_{t,G}(s,v)$, where the additional factor of $(1+\epsilon_2)$ is due to scaling of $G \cup \bar{H}_{j+1}$.

Now consider the case $s \notin B(v)$. Recall that we inductively showed one of the two theorem conditions hold for each i for length in H_{j+1}, and we now argue that this corresponds to one of the two conditions above for the same i, but now on G^{j+1}. Let π_i be the path in H_{j+1} that satisfies one the theorem conditions for a fixed i.

First assume that no edge on this path is stretched. Then the stretch argument for $L(s,v)$ clearly holds based on the earlier arguments and Lemma 4.2.4. Now let us argue about the possible insertions on this path, i.e. when an edge added on π_i is strecthed with respect to s. Note that by our construction, and in all cases we considered in our hopset argument, an edge (x', y') was inserted into H_{j+1} only when $x' \in B(y')$ for some $0 \leq i \leq k + 1/\rho + 1$, and the weights were assigned based on Observation 4.3.1. Using these weights, we prove a claim that allows us to reason about possible insertions on π_i. At a high level, we show that the level of y' is either determined an estimate at time $t-1$ for $d(s,y')$ or by the level of a node x', and a single edge between (x', y') with weights satisfying Observation 4.3.1. In other words, in the second case using a case by case analysis same as before, we know that for any node y' there exists another node, in this case x', that shortcuts the path from s to y' using one edge.

Claim 4.3.8. *Let (x', y') be an edge added to H_{j+1} and hence G^{j+1} with weight $w_{G^{j+1}}(x', y')$ due to the fact that $x' \in B(y')$. Then either of the following holds for the level of node y' in the monotone ES tree rooted at s:*

- $L_{t,j+1}(s,y') = L_{t-1,j+1}(s,y')$ *and thus* $\eta(2^{j+1}, \epsilon_2) L_{t,j+1}(s,y') \leq \eta(2^{j+1}, \epsilon_2) L_{t-1,j+1}(s,y') \leq (1+\epsilon_{j+1}) d_{t,G}(s,y')$*; or,*
- *We have* $L_{t,j+1}(s,y') \leq L_{t,j+1}(s,x') + w_{G^{j+1}}(x', y')$.

Proof. The first case is when the edge (x', y') is stretched in the tree rooted at s on G^{j+1}. Note that this is different from the setting in Observation 4.3.1, where we were reasoning about the node y' being stretched in the tree rooted at x' on G^j. In this case we set $L_{t,j+1}(s, y') = L_{t-1,j+1}(s, y')$. Since we have maintained distances up to depth $\lceil \frac{2(2\beta+1)}{\epsilon_2} \rceil$ on G^{j+1} with stretch $(1 + \epsilon_j)$ at time $t - 1$, and since we are in the decremental setting this means that after scaling back we get the desired stretch.

The second case is when the edge (x', y') is not stretched in the tree rooted at s. The claim follows by definition of an edge that is not stretched. □

Note that if $d_{t,G}(x', y')$ belonged to a smaller scale, we have already added an edge that satisfied a similar condition for the corresponding scale.

Going back to the hopset argument, we note that every insertion into H_{j+1} on path π_i (and edge that is stretched with respect to s) satisfies the conditions in Claim 4.3.8. In other words, for any node on the path, say y', there exists a node x' that is directly connected to y', satisfying the stretch in Claim 4.3.8. This combined with what we proved inductively on the structure of segments of path π_1 in H_{j+1}, implies that for any node $v, d(s, v) \leq 2^{j+1}$ we have a path with the desired stretch that is consisted of all the edges added for different i. Finally, after the scaling we can obtain the desired stretch in which we lose another factor of $(1 + \epsilon_2)$.

We briefly review the cases that, at a high-level, shows that a such a node x', satisfying Claim 4.3.8 exists that appropriately *shortcuts* the distance from s to y' for any node y' on π_i stretched with respect to s.

Recall the hopset argument for i: an insertion into one of the segments (of length $\delta d(s, v)$) can only occur when condition two of the theorem is satisfied for some node in $z \in A_{i+1}$. Let $[z'_l, z'_r]$ be the segment for which the new edge was inserted. We argued that either $z'_r \in B(z'_l)$ or there is another node z' for which the second theorem condition holds and $z' \in B(z'_l)$.

In any case we inserted a single edge in H_{j+1} on this segment with weights satisfying Observation 4.3.1. Then using Claim 4.3.8, and similar calculation as we did for H_{j+1} we can show that the second condition also holds on G^{j+1}, but there is an additional error factor of $(1+\epsilon_2)$ from scaling.

At a high level, we have shown that the inserted edge on π_i has a length that appropriately shortcuts the last segment, otherwise no new edges were added in iteration i (when the first theorem condition holds for all segment).

We argued earlier that this path π_i has stretch $(1+\epsilon_j)(1+\epsilon)(1+\epsilon_2)$ in $G \cup \bar{H}_{j+1}$. Hence after scaling and running Algorithm 6 on G^{j+1}, we know that path π_i has depth $\lceil \frac{2(2\beta+1)}{\epsilon} \rceil$ and we have the following estimate for v:

$$d_{t,j+1}(s,v) \leq \min_{r=1}^{j+1} \eta(2^{j+1}, \epsilon_2) L_{t,r}(s,v) \leq (1+\epsilon_j)(1+\epsilon_2)^2(1+\epsilon) d_{t,G}(s,v)$$

Then after all the iterations $1 \leq i \leq 1/\rho + 1 + k$, the second condition cannot hold (since $A_{1/\rho+1+k} = \emptyset$), the first condition must hold, which states that there is a path with $\beta = (3/\delta)^{1/\rho+1+k}$-hops and stretch $(1+\epsilon_{j+1}) d_{t,G}(s,v)$ in $G \cup \bar{H}_{j+1}$ between s and v. Also by path doubling of Lemma 4.3.6 we argued that this also means that there is a path with $2\beta+1$ hops and $(1+\epsilon_j)(1+\epsilon_2)(1+\epsilon)$-stretch in $G \cup \bar{H}_{j+1}$ between s and v that is consisted of two paths satisfying the first theorem condition for H_{j+1}, and that this corresponds to a path with stretch $1+\epsilon_{j+1}$ in G^{j+1}. The concatenation of this same paths in G^{j+1} approximates π_i and after scaling and unscaling we will have an additional factor of $(1+\epsilon_2)$.

□

Theorem 4.3.7 allows us to hierarchically use the restricted hopsets for smaller scales to compute the distance for larger scales, that is in turn used to update the hopset edges in the larger scales.

In the following lemma, we will show that by setting $\delta = O(\frac{\epsilon}{(k+1/\rho+1)})$ we get the desired stretch for Lemma 4.2.3. Next, we use Lemma 4.2.3 for all scales and by setting the appropriate error parameters we can prove our overall stretch and hopbound tradeoffs. We also prove the overall update time using the running time of the monotone ES tree algorithm to run the restricted hopset algorithm on the hopsets obtained for each scale.

Single scale stretch. We will now use the stretch argument in Section 4.3.3 to get the hopbound and stretch for each scale by setting the appropriate parameters. As discussed, there are *two* error factors incurred in each scale. One is caused by the fact that we are using previously added hopset edges, which we denoted by $(1+\epsilon_j)$ for scale j, and another is caused due to the rounding error, which we denote by $(1+\epsilon_2)$. To get an overall stretch of $(1+\epsilon)$, we will set $\epsilon' = \frac{\epsilon}{6\log W}$ and $\epsilon_2 = \epsilon'$.

Corollary 4.3.9. *After each update t, and for all $j, 0 \le j \le \log W$ and any pair $x, y \in V$, where $2^j \le d_{t,G}(x,y) \le 2^{j+1}$, we have $d_{t,G}(x,y) \le d_{t,G\cup \bar{H}_j}(x,y) \le (1+3\epsilon')^j \cdot d^{(\beta)}_{t,G}(x,y)$.*

Proof. We use an induction on j. The base case $(j=0)$ is satisfied by the paths in G, since we can assume with out loss of generality that the edge weights are at least one. First, by induction hypothesis, we have a $(2^j, \beta, (1+3\epsilon')^j)$-hopset, and hence $1+\epsilon_j = (1+3\epsilon')^j$.

We then use Theorem 4.3.7, for $\epsilon_2 = \epsilon'$, and $\delta = \frac{\epsilon'}{8(k+1/\rho+1)}$. For the final iteration $i = \frac{1}{k+1/\rho+1}$ since $A_{i+1} = \emptyset$, the second item can not hold. Hence the first item should hold, and since $8\delta i < \epsilon'$ we have,

$$d^{(\beta)}_{t,G\cup\bar{H}_j}(x,y) \le (1+3\epsilon')^{j-1}(1+\epsilon')(1+\epsilon')d_{t,G}(x,y) \le (1+3\epsilon')^j d_{t,G}(x,y).$$

□

Proof of stretch and hopbound in Lemma 4.2.3. Now by simply setting $\epsilon' = \frac{\epsilon}{3}$ in Corollary 4.3.9 we get the desired stretch and hopbound.

Putting it together. We now use the stretch argument of Corollary 4.3.9 with the update time followed by Lemma 4.2.3 to get the following hopset guarantees.

Theorem 4.3.10. *The total update time in each scaled graph G^j, $1 \leq j \leq \log W$, over all deletions is $\tilde{O}((\ell/\epsilon')(n^{1+\nu}+m)n^{\rho})$, and hence the total update time*[5] *for maintaining a $(\beta, 1+\epsilon)$-hopset with hopbound $\beta = O(\frac{\log W}{\epsilon} \cdot (k+1/\rho))^{k+1/\rho+1}$ is $\tilde{O}(\frac{\beta}{\epsilon} \cdot mn^{\rho})$.*

Proof. First we use Corollary 4.3.9 to prove the stretch and hopbound, by setting $j = \log W$. For the final scale we have $d_{t,\log W}(u,v) = (1+3\epsilon')^{\log W} d_G(u,v) \leq (1+\epsilon)\log W$. The hopbound obtained is

$$O(\frac{1}{\epsilon'} \cdot (k+1/\rho))^{k+1/\rho+1} = O(\frac{\log W}{\epsilon} \cdot (k+1/\rho))^{k+1/\rho+1}.$$

The running time follows by Lemma 4.2.3 where $\Delta = O(n^{1+\nu})$, we get an overall running time of $\tilde{O}(mn^{\rho} \cdot \frac{\beta}{\epsilon})$ time. □

Hence the total update $\tilde{O}(mn^{\rho})$ and the hopbound β is polylogarithmic when ρ and k are constant.

4.4 Applications

In this section we explain two applications of our decremental hopsets to get improved bounds for $(1+\epsilon)$-approximate SSSP and MSSP and $(2k-1)(1+\epsilon)$-APSP. For both of these problems we first construct a hopset, where we choose the appropriate hopbound depending on the number of source. We then use the scaling scheme in Lemma 4.2.4 on the obtained graph.

Our algorithm for $(2k-1)(1+\epsilon)$-APSP involves maintaining two data structures simultaneously: A $(\beta, 1+\epsilon)$-hopset, and a Thorup-Zwick distance oracle [88]. At a high-level the hopset will let us maintain the distance oracle much faster, at the expense of a $(1+\epsilon)$-factor loss in the stretch.

[5]If weights are not polynomial the $\log n$ factor will be replace with $\log W$, and a factor of $\log^2 W$ will be added to the update time.

4.4.1 $(1+\epsilon)$-approximate SSSP and $(1+\epsilon)$-MSSP

Given a graph $G=(V,E)$ and a set S of size of sources, our goal is to maintain the distance from each source in $\tilde{O}(sm+mn^{\rho})$, total update time (where ρ is a constant), and constant query time.

Once a (β,ϵ)-hopset is constructed, we can run Algorithm 6 on all the scaled graphs $G^1, G^2, ..., G^{\log W}$ up to depth $O(\beta)$, scale back the distances, and return the smallest value to each source. Similar approaches for h-limited SSSP have also been used in previous work such as [15, 55, 59].

In the next theorems we argue that using the same techniques as we used for maintaining the hopset (that are similar to framework of [56]), namely by combining monotone ES tree and scaling, we get our SSSP and MSSP results. In particular after constructing the hopset we can use Theorem 4.3.7 and Theorem 4.3.10 to get,

Theorem 4.4.1. *Given an undirected and weighted graph $G=(V,E)$, there is a decremental algorithm for maintaining $(1+\epsilon)$-approximate distances from a set S of sources in total update time of $\tilde{O}(\beta(|S|(m+n^{1+\frac{1}{2^k-1}})+mn^{\rho}))$, where $\beta = O(\frac{\log W}{\epsilon}\cdot(k+1/\rho)^{k+1/\rho+1})$, and with $O(1)$ query time.*

Proof. We maintain a $(\beta,\frac{\epsilon}{3})$-hopset H based on Theorem 4.3.10. Then we run Algorithm 6 on $G\cup H$ from all the s for all scaled graphs. The the claim follows by the argument in Theorem 4.3.7. In particular, after adding all the hopset edges at time t for all scales, we will run the monotone ES tree algorithm rooted at each source again on the union of all scaled graphs $G^1\cup ...\cup G^{\log W}$ (by setting $\epsilon_0=\epsilon/3$) and let the level $L(s,v)$ of a node be $\min_j \eta(2^j,\frac{\epsilon}{3})L_j(s,v)$ where $L_j(s,v)$ is the level of v on G^j after running the monotone ES tree that is run up to depth β. By item 3 of Theorem 4.3.10, we get an overall stretch of $(1+\epsilon/3)^2\leq(1+\epsilon)$.

The time required for maintaining the hopset is $\tilde{O}((m+n^{1+\nu})n^{\rho})$ and by setting $n^{\rho}=s$ the time required for maintaining h-SSSP from all sources is $O(sm\cdot\beta)=\tilde{O}(sm)$, when $s = n^{\Omega(1)}$. $\square$

We next state two specific consequences. First implication is that when the number of sources is a polynomial, and the graph is not very sparse, we can get a near-optimal (up to polylogarithmic factors) algorithm for $(1+\epsilon)$-MSSP.

Corollary 4.4.2. *Given an undirected and weighted graph $G = (V, E)$, where $|E| = n^{1+\Omega(1)}$, there is a decremental algorithm for maintaining $(1+\epsilon)$-approximate distances from s sources, where $s = n^{\Omega(1)}$ in total update time of $\tilde{O}(sm)$, and with $O(1)$ query time.*

When the number of sources $s = n^{o(1)}$ (e.g. in case of SSSP), the best tradeoff can be obtain by setting $\rho = \frac{\log\log n}{\sqrt{\log n}}$. We will then have $\beta = 2^{\tilde{O}(\sqrt{\log n})}$ and also $n^{\rho} = 2^{\tilde{O}(\sqrt{\log n})}$. In this case we get improved bounds over the result of [55], which has a total update time of is $mn^{\tilde{O}(\log^{3/4} n)}$.

Corollary 4.4.3. *Given an undirected and weighted graph $G = (V, E)$, there is a decremental algorithm for maintaining $(1+\epsilon)$-approximate distances from s sources, when $0 < \epsilon < 1$ is a constant and $|E| = n \cdot 2^{\tilde{\Omega}(\sqrt{\log n})}$, with total update time of $\tilde{O}(sm \cdot 2^{\tilde{O}(\sqrt{\log n})})$, and with $O(1)$ query time. Hence, we can maintain $(1+\epsilon)$-approximate SSSP in $2^{\tilde{O}(\sqrt{\log n})}$ amortized time.*

4.4.2 APSP distance oracles

It is known that in static settings for any weighted graph $G = (V, E)$, we can construct a Thorup-Zwick [88] distance oracle of size (w.h.p.) $\tilde{O}(n^{1+1/k})$, such that after the preprocessing time of $\tilde{O}(mn^{1/k})$, we can query $(2k-1)$-approximate distances for any pair of nodes in $O(k)$ time. In this section we show that in decremental settings we can maintain these distance oracles in total update time of $\tilde{O}(mn^{1/k})$ (for graphs that are not too sparse), and we can query $(2k-1)(1+\epsilon)$-approximate distances in $O(k)$ time. This can be done by maintaining a $(\beta, 1+\epsilon)$-hopset and a distance oracles for

G at the same time, where β is polylogarithmic in n. Intuitively, the hopset will allow us to update distances faster on the distance oracles.

Distance oracle construction via hopsets. Assume that we are given a $(\beta, 1+\epsilon)$-hopset for G. The algorithm for constructing the Thorup-Zwick distance oracle is as follows: Similar to the algorithm in Section 4.2.1, we define sets $V = A_0 \supseteq A_1 \supseteq ... \supseteq A_k = \emptyset$[6]. But here each set A_{i+1} is obtained by sampling each element from A_i with probability $p_i = n^{-1/k}$. Same as before, for every node $u \in A_i \setminus A_{i+1}$, let $p_i(u) \in A_{i+1}$ be the closest node to set A_{i+1}. We the bunch of a node u is the set $B(u) = \cup_{i=1}^{k} B_i(u) = \{v \in A_i : d(u,v) < d(u, A_{i+1})\} \cup \{p(u)\}$, $C(v)$ called the cluster of v such that if $v \in B(u)$ then $u \in C(v)$. The distance oracle is consisted of bunches $B(v)$ for all $v \in V$, and the distances associated with them. Note that the information stored here are also different from the hopset algorithm described in Section 4.2.1, since there we only added edges for nodes $v \in A_i$ and their bunches. Thorup and Zwick [88] show that this distance oracle has the following properties (in static settings):

Theorem 4.4.4 ([88]). *There is a distance oracle of expected size $O(kn^{1+1/k})$, that can answer queries $(2k-1)$-approximate distance queries for a given weighted and undirected graph $G = (V, E)$ in $O(k)$ time for any $k \geq 2$. The preprocessing time in static settings is w.h.p. $\tilde{O}(mn^{1/k})$.*

As discussed in Section 4.3.1, Roditty and Zwick [85] showed how to maintain this data structure in $O(mn)$ update time for *unweighted graphs*, but where the size is increased to $\tilde{O}(m + n^{1+1/k})$. For weighted graphs their updates time can be as large as $O(mn^{1+1/k})$. We will argue that by maintaining a $(\beta, 1+\epsilon)$-hopset along with the distance oracle we can improve the total update time to $\tilde{O}(\beta mn^{1/k})$. This combined with our hopset construction in Section 4.3.1 will lead to the desired bounds. More

[6] This k should not be confused with the size parameter in the hopset algorithm of Section 4.2.1. Here we only use the fact that the hopset size can be bounded based on the graph density.

formally,

Theorem 4.4.5. *Given a weighted and undirected graph $G = (V, E)$ and a $(\beta, 1+\epsilon)$-hopset H for G, and a parameter $k \geq 2$, we can maintain a distance oracle of size $\tilde{O}(m + n^{1+1/k})$ and with stretch $(1+\epsilon)(2k-1)$ in $\tilde{O}(\frac{\beta}{\epsilon} \cdot mn^{1/k})$ total update time.*

Proof. We will again use the scaling idea described in Section 4.3.1. Similar to Theorem 4.4.1, we consider the sequence $G^1, ..., G^j$, where $G^r, r \leq j$ is scaling of the graph $G \cup \bar{H}_r$ as defined in Section 4.3.1, where $\epsilon_0 = \frac{\epsilon}{3}$ and $\bar{H}_j$ is a $(2^j, \beta, \frac{\epsilon}{3})$ hopset. We then run the algorithm of Roditty-Zwick [85] on G up to depth $\lceil 3\beta/\epsilon \rceil$ for maintaining the clusters and the bunches. The algorithm and the running time analysis is similar to the restricted hopset algorithm described in Section 4.3.1. The main differences in the algorithm are the sampling probabilities and the information stored. Therefore using the argument in Lemma 4.3.2 we can show that by running this algorithm on G with depth $\lceil 3\beta/\epsilon \rceil$ we can maintain a bunch $B_i(u)$ for all nodes $u \in V, 1 \leq i \leq k-1$ in $\tilde{O}(\frac{\ell m}{\epsilon q_i}) = \tilde{O}(\frac{\beta}{\epsilon} mn^{1/k})$ total update time. This algorithm lets us maintain clusters. We also maintain the distances in clusters and hence bunches as follows: For each $v \in V, u \in B(v)$, we run single-source shortest path distance between from v on scaled graphs $G^1, ..., G^{\log W}$ (by setting $\epsilon_0 = \epsilon/3$). We then set the distance $d(u, v)$ to be $\min_j \eta(2^j, \frac{\epsilon}{3}) L_j(s, v)$ where $L_j(s, v)$ is the level of v on G^j after running the monotone ES tree that is run up to depth $\lceil 6(\beta+1)/\epsilon) \rceil$.

Again, when we combine the hopset stretch with the stretch with the rounding algorithm caused by rounding, we get an overall stretch of $(1+\frac{\epsilon}{3})^2 \leq 1+\epsilon$. The overall stretch is thus $(2k-1)(1+\epsilon)$. □

Theorem 4.4.6. *Given a weighted graph $G = (V, E)$ with polynomial weights, and constant*[7] *$k \geq 2$ and $0 < \epsilon < 1$, we can maintain a data structure with expected size $\tilde{O}(m+n^{1+1/k})$ and total update time of $\tilde{O}(mn^{1/k} \cdot (1/\epsilon)^{O(1)})$, that returns $(2k-1)(1+\epsilon)$-*

[7]If $k = \omega(1)$, then a factor of $n^{o(1)}$ will be added to the running time.

stretch queries for any pair $u, v \in V$ with $O(k)$ query time.

Proof. We construct and maintain a $(\beta, 1+\frac{\epsilon}{3})$-hopset using Theorem 4.3.10. If $m = n^{1+\Omega(1)}$ we can set $\rho = \frac{1}{k}$, and we set the hopset size parameter ν to a small constant[8] such that $O(n^{1+\nu}) = O(m)$. If $m = n^{1+o(1)}$, we set $\rho = \frac{1}{2k}$). In both cases time required for maintaining a hopset is $\tilde{O}(mn^{1/k} \cdot (1/\epsilon)^{O(1)})$. We get hopbound $\beta = O(\log n/\epsilon)^{\log(1/\nu)+1/k+1} = \textit{polylog}\ (n)$. Hence we can also maintain the distance oracle in $\tilde{O}(mn^{1/k})$ total update time. The stretch will be $(2k-1)(1+\epsilon)$, and the query time remains the same as the static query time, which is $O(k)$. □

4.5 Conclusion and Future Direction

While we do not prove this formally, we expect that many of our techniques would also extend to the incremental (insert-only) settings. However we heavily rely on properties of partially dynamic settings, and we would need new techniques for constructing hopsets efficiently in the fully-dynamic settings, which is a natural next step. Currently known bounds for shortest path computation is far from optimal in this model, and not much is known about utilizing hopsets in these settings. Instead recent results in fully-dynamic model heavily rely on algebraic techniques such as matrix multiplication. It would be interesting to see if those techniques can be combined with hopset constructions to achieve much better bounds.

[8]The choice of size parameter impact the polylogarithmic factors. Hence one option is to choose the smallest constant such that the graph size is not smaller than the hopset size.

Chapter 5

Distributed Distance Bounded Network Design

5.1 Introduction

In this chapter, we focus on an optimization perspective on spanners and several related network design problems. The results presented here are published in [30].

Distributed network design is a classical type of distributed algorithmic problem, going back at least to the seminal work on distributed MST by Gallager, Humblet, and Spira [48]. By "network design", we mean the class of problems which can be phrased as "given input graph G, find a subgraph H which has some property P, and minimize the cost of H". Clearly different properties P, and different notions of cost, lead to very different problems. One important class of problems are *distance-bounded* network design problems, where the property P is that certain pairs of vertices are within some distance of each other in H (where distance refers to the shortest-path distance). The most well-known type of distance-bounded network design problems are problems involving *graph spanners*, in which the distance requirement is that the distance in H for all (or certain) pairs is within a certain factor (known as the stretch) of their original distance in H. But there are many other important versions of distance-bounded network design, such as the bounded diameter problem [32] and the shallow-light Steiner tree/network problems [61].

Many of these problems are NP-hard, so they cannot be solved optimally in polynomial time even in the centralized setting unless P=NP. Thus they have been studied extensively from an approximation algorithms point of view, where we design algorithms which approximate the optimal solution but which run in polynomial time. For many of these problems, a key step in the best-known centralized approximation algorithm is solving a linear programming relaxation of the problem, and then rounding the optimal fractional solution into a feasible integral solution. Interestingly, it is relatively common for the rounding to be "local": if we are in a distributed setting and happen to know the optimal fractional LP solution, then the algorithm used to round this to an integral solution can be accomplished with a tiny amount of extra time (either 0 or a small constant number of rounds). So the bottleneck when trying to make these algorithms distributed is solving the LP, not rounding it.

Solving LPs in distributed settings has received only a small amount of attention, since it unfortunately turns out to be extremely challenging in general. Most notably, Kuhn, Moscibroda, and Wattenhofer [66] gave an efficient distributed algorithm (in the LOCAL model of distributed computation) for packing/covering LPs. Unfortunately, the LPs used for distance-bounded network design are not packing/covering LPs[1], and hence we are not able to use their techniques. Floréen et al. [45] also studied a special class of linear programs, namely min-max LPs, in distributed settings, which also cannot be used for our problems of interest. In this chapter we show how to solve these LPs (and convex generalizations of them) in the LOCAL model of distributed computation, which almost immediately gives the best-known results for a variety of distance-bounded network design problems.

In particular, for many network design problems (Directed k-Spanner, Basic

[1]They can be turned into packing/covering LPs through a projection operation, but unfortunately this technique results in an exponential number of constraints, making [66] inapplicable. However, this technique has been used in the centralized setting for the fault-tolerant directed k-spanner problem [27].

3-Spanner, Basic 4-Spanner, Lowest-Degree k-Spanner, Directed Steiner Network with Distance Constraints with spanning demands, and Shallow-Light Steiner Network with spanning demands) we give approximation algorithms which run in $O(D \log n)$ rounds (where D is the maximum distance bound) and have the same approximation ratios as in the centralized setting. Previous distributed algorithms for these problems with similar round complexity have either used "heavy" computations (non-polynomial time algorithms) at the nodes (in which case they can often do *better* than the best computationally-bounded centralized algorithm), or give approximation bounds which are asymptotically worse than the best centralized bounds. See Section 5.1.2 for more discussion of previous work.

5.1.1 Our Results

We give two main types of results. First, we give a distributed algorithm that (approximately) solves distance-bounded network design convex programs with small round complexity. We then use this result to (almost immediately) get improved distributed approximation algorithms for a variety of network design problems.

5.1.1.1 Solving convex programs

Stating our main technical result (distributed approximations of distance-bounded network design convex programs) in full generality requires significant technical setup, so we provide an informal description here. See Section 5.4 for the full definitions and theorem statements (Theorem 5.4.4 in particular). But informally, a distance-bounded network design convex program is the following. We are given a graph $G = (V, E)$, a set $\mathcal{S} \subseteq V \times V$, and for each $(u, v) \in \mathcal{S}$ there is a set of "allowed" $u - v$ paths $\mathcal{P}_{u,v}$. Roughly speaking, we typically assume that the allowed paths are short, and define D to be maximum length of such paths. Informally, the discrete problem is to find a subgraph H of G so that every $(u, v) \in \mathcal{S}$ is connected by at least one path from $\mathcal{P}_{u,v}$ in H, and the goal is to minimize some notion of "cost". If our notion of "cost" is captured by an objective function $g : \mathbb{R}_{\geq 0}^{|E|} \to \mathbb{R}$ (which is typically linear, but which can be more general convex functions as long as they satisfy a "partitionability" constraint – see Section 5.4 for the details), then the natural relaxation of this problem is the following convex program, which has a variable x_e for every edge and a variable f_P for every allowed path.

$$\begin{aligned}
\min \quad & g(x) & \\
\text{s.t.} \quad & \sum_{P \in \mathcal{P}_{u,v} : e \in P} f_P \leq x_e & \forall (u,v) \in \mathcal{S}, \forall e \in E \\
& \sum_{P \in \mathcal{P}_{u,v}} f_P \geq 1 & \forall (u,v) \in \mathcal{S} \\
& x_e \geq 0 & \forall e \in E \\
& f_P \geq 0 & \forall (u,v) \in \mathcal{S}, \forall P \in \mathcal{P}_{u,v}
\end{aligned}$$

Informally, the first type of constraint says that an allowed path is included only if all edges in it are included, and the second type of constraint required us to include at least one allowed path for each $(u,v) \in \mathcal{S}$. We call this type of convex program a *distance-bounded network design convex program.* It is clearly not a packing/covering LP due to the first type of constraint, and hence there is no known distributed algorithm to solve this kind of program. However, note that if the maximum length of any allowed path is constant, then there are only a polynomial number of such paths, and hence the size of the convex program is polynomial and so it can be solved in polynomial time in the centralized setting under reasonable assumptions on g (see [51] for details on solving convex programs in polynomial time).

Our main technical result is that we can approximately solve these optimization problems even in a distributed setting. For any path P let $\ell(P)$ denote the length of the path (the number of edges in it).

Theorem 5.1.1. *For any constant $\epsilon > 0$, any distance-bounded network design convex program can be solved up to a $(1+\epsilon)$-approximation in $O(D \log n)$ rounds in the LOCAL model, where $D = \max_{(u,v)\in\mathcal{S}} \max_{P\in P_{u,v}} \ell(P)$. Moreover, if the convex program can be solved in polynomial time in the centralized sequential setting, then the distributed algorithm uses only polynomial-time computations at every node*

The dependence on ϵ in the above theorem is hidden in the $O(\cdot)$ notation – see Theorem 5.4.4 for the full statement.

Padded Decompositions. Our main technique is to use a distributed construction of *padded decompositions*, a specific type of network decomposition which we explain in detail in Section 5.3. Padded decompositions have been very useful for metric embeddings and approximation algorithms (e.g., [53, 65]), but to the best of our knowledge have not been used before in distributed algorithms (with the exception of [28], which used a special case of them to give a distributed algorithm for the fault-tolerant 2-spanner problem). Very similar decompositions, such as the famous Linial-Saks decomposition [71], have been used extensively in distributed settings, but the guarantees for padded decompositions are somewhat different (and we believe that these decompositions may prove useful in the future when designing distributed approximation algorithms). In Section 5.3 we give a distributed algorithm in the LOCAL model to construct padded decompositions. These padded decompositions allow us to solve a collection of "local" convex programs with the guarantees that a) most of the demands in $\mathcal{S}$ are satisfied in one of the local programs, and b) the solutions of the local convex programs combine into a (possibly infeasible) global solution with cost at most the cost of the global optimum. Then by averaging over $O(\log n)$ of these decompositions we get a feasible global solution which is almost optimal. Interestingly, the neighborhood covers that we constructed in Chapter 3 were also based on repeated construction of padded decompositions with different radius parameters. The algorithm used in this section is based on a different construction of these objects due to Bartal [7]. We use this construction as it is simpler to show the properties that we need for this section, even though this construction does not directly apply to weighted graphs, which is what we needed in Chapter 3.

5.1.1.2 Distributed approximation algorithms for network design

Solving convex programming problems in distributed environments is interesting in its own right, and Theorem 5.1.1 is our main technical contribution, but the particular class of convex programs that we can solve are mostly interesting as convex relaxations of interesting combinatorial optimization problems. Many of the problems are NP-hard, but there has been significant work (some quite recent) on designing approximation algorithms for them (see, e.g., [27, 22, 31, 14]). Almost all of these approximations depend on convex relaxations which fall into our class of "distance-bounded network design convex programs". This means that as long as the rounding scheme can be computed locally, we can design distributed versions of these approximation algorithms by using Theorem 5.1.1 to solve the appropriate convex relaxation and then using the local rounding scheme.

We are able to use this framework to give distributed approximation algorithms for several problems. Most of them are variations of *graph spanners*, which were introduced by Peleg and Ullman [84] and Peleg and Schäffer [83], and are defined as follows.

Definition 5.1.1. Let $G = (V, E)$ be a graph (possibly directed), and let $k \in \mathbb{N}$. A subgraph H of G is a *k-spanner* of G if $d_H(u, v) \leq k \cdot d_G(u, v)$ for all $u, v \in V$. The value k is called the *stretch* of the spanner.

Before stating our results, we first define the problems. In the Basic k-Spanner problem we are given an undirected graph G and a value $k \in \mathbb{N}$. A subgraph H of G is a feasible solution if it is a k-spanner of G, and the objective is to minimize the number of edges in H. For $k = 3, 4$, the best-known approximation algorithm for this problem is $\tilde{O}(n^{1/3})$ [14, 31]. If the input graph G (and the solution H) are directed, then this is the Directed k-Spanner problem, for which the best-known approximation is $\tilde{O}(\sqrt{n})$ [14]. If the objective is instead to minimize the maximum degree in H then this is the Lowest-Degree k-Spanner problem, for which the best-known approximation is $\tilde{O}(n^{(1-1/k)^2})$ [22].

The following theorem contains our results on distributed approximations of graph spanners. Informally, it states that we can give the same approximations in the LOCAL model as are possible in the centralized model.

Theorem 5.1.2. *There are algorithms in the LOCAL model that w.h.p.*[2] *provide the following guarantees. For* Directed *k-*Spanner*, the algorithm runs in $O(k \log n)$ rounds and gives an $\tilde{O}(\sqrt{n})$-approximation. For* Basic *3-*Spanner *and* Basic *4-*Spanner*, the algorithms run in $O(\log n)$ rounds and gives an $\tilde{O}(n^{1/3})$-approximation. For* Lowest-Degree *k-*Spanner*, the algorithm runs in $O(k \log n)$ rounds and gives an $\tilde{O}(n^{(1-1/k)^2})$-approximation. All of these algorithms use only polynomial-time computations at each node.*

We emphasize that our algorithms for these spanner problems both match the best-known centralized approximations and only use polynomial-time computations at each node. There is significant previous work (see Section 5.1.2) on designing distributed approximation algorithms for these and related problems that has only one of these two properties, but all previous approaches which use only polynomial-time computations necessarily do worse than the best centralized bound (or have much worse round complexity). At a high level, this is because previous approaches (most notably [6]) do not actually use the structure of the centralized algorithm: they only use the efficient centralized approximation as a black box. By going inside

[2]By "with high probability" (or w.h.p.), we mean with probability at least $1 - 1/n^c$ for some $c \geq 1$.

the black box and noticing that they all use a similar type of convex relaxation, we can simultaneously get low round complexity, best-known approximation ratios, and efficient local computation.

It turns out that we can use our techniques for an even broader question: Directed Steiner Network with Distance Constraints with a spanning demand graph. In this problem there is a set $\mathcal{S} \subseteq V \times V$ of demands, and for every demand $(u, v) \in \mathcal{S}$ there is a length bound $L(u, v)$. The goal is to find a subgraph H so that $d_H(u, v) \leq L(u, v)$ for all $(u, v) \in \mathcal{S}$, and the objective is to minimize the number of edges in H. The state of the art centralized bound for this problem is a $O(n^{3/5+\epsilon})$-approximation [23], but if we further assume that every vertex $u \in V$ is the endpoint of at least one demand in $\mathcal{S}$ (which we will refer to as a *spanning demand graph*) then it is straightforward to see that the centralized algorithm of [14] for Directed k-Spanner can be generalized to give a $\tilde{O}(\sqrt{n})$-approximation. Our distributed version of this algorithm also generalizes, w.h.p. giving the following result.

Theorem 5.1.3. *There is an approximation algorithm in the LOCAL model for Directed Steiner Network with Distance Constraints with a spanning demand graph with approximation ratio $\tilde{O}(\sqrt{n})$ which runs in $O((\max_{(u,v)\in\mathcal{S}} L(u, v)) \log n)$ rounds and uses only polynomial-time computations.*

Note that Directed k-Spanner and Basic k-Spanner are special cases of this problem, where there is a demand for every edge and the length bound is just k times the original distance. Interestingly, other network design problems which have proved important for distributed systems are also special cases, including the Distance Preserver problem (when $L(u, v) = d_G(u, v)$ for all $(u, v) \in \mathcal{S}$), the Pairwise k-Spanner problem (where $L(u, v) = k \cdot d_G(u, v)$ for all $(u, v) \in \mathcal{S}$), and the Shallow-Light Steiner Network problem (where $L(u, v) = D$ for all $(u, v) \in \mathcal{S}$, for some global parameter D). Shallow-Light Steiner Network in particular is a key component in state of the art systems for reliable Internet transport [5], although in that particular application the demand graph is not spanning. Extending our techniques to handle totally general demands by giving a distributed version of [23] is an extremely interesting open question.

5.1.2 Related Work

While distributed solving of convex programs is a natural question, there is little previous work in the LOCAL model. Possibly most related to our results is a line of work on solving *positive* linear programs (packing and covering LPs). This was introduced by [78], improved by [8], and then essentially optimal upper and lower bounds were given by [66]. Unfortunately, the convex programs we consider are not positive linear programs due to the "capacity" constraints in which some variables appear with positive coefficients while others have negative coefficients.

A special case of our result was proved earlier in [28], who showed how to solve the LP relaxation of Basic 2-Spanner in the LOCAL model in $O(\log^2 n)$ rounds (they actually show more than this, by giving a distributed algorithm for the *fault-tolerant* version of Basic 2-Spanner, but that is not germane to our results). Our techniques are heavily based on [28], which is itself based on the ideas from [66]. In particular, [66] uses a Linial-Saks decomposition [71] to solve "local" versions of the linear program in different parts of the graph, and then combines these appropriately. To make this work for the Basic 2-Spanner LP relaxation, [28] had to use *padded decompositions*, which can be thought of as a variant of Linial-Saks with slightly different guarantees which, for technical reasons, are more useful for network design LPs. In this chapter we extend these techniques further by giving a more general definition of padded decomposition which works for larger distance requirements, showing how to construct them in the LOCAL model, and then showing that the basic "combining" idea from [28] can be extended to handle these more general decompositions and far more general constraints and objective functions.

The major type of combinatorial optimization problem which our techniques allow us to approximate are various versions of graph spanners. There are an enormous number of papers on spanners in both centralized and distributed models, but fewer papers which attempt to find the "best" spanner for the particular given input graphs (most papers on spanners give existential results and algorithms to achieve them, rather than optimization results). These optimization questions (e.g., BASIC k-SPANNER, DIRECTED k-SPANNER, and LOWEST-DEGREE k-SPANNER) have been considered quite a bit in the context of centralized approximation algorithms and hardness of approximation [27, 14, 31, 22, 26], but almost all of the known centralized results use linear programming relaxations, making them difficult to adapt to distributed settings. Hence there have been only two results on optimization bounds in distributed models: [28] and [6].

Barenboim et al. [6] provided a distributed algorithm using Linial-Saks decompositions that for any integer parameters k, α, gives an $O(n^{1/\alpha})$-approximation for DIRECTED k-SPANNER in $\exp(O(\alpha)) + O(k)$ time. This is an extremely strong approximation bound, and in fact is better than even the best centralized bound. This is possible due to their use of very heavy (exponential time) local computation. Our algorithms, on the other hand, take polynomial time for local computations. Barenboim et al. [6] show that heavy local computations can be removed from their algorithm by using a centralized approximation algorithm for a variant of spanners known as *client-server k-spanners*, and in particular that an $f(k)$-approximation for client-server k-spanner can be turned into an $O(n^{1/\alpha} f(k))$-approximation algorithm running in $\exp(O(\alpha)) + O(k)$ rounds in the $LOCAL$ model for minimum k-spanner with only polynomial local computation. So in order to achieve the same asymptotic approximation ratio as the best-known centralized algorithm, the parameter α must be $\Omega(\log n)$ and hence the running time is polynomial in n, even though k might be a constant. It is essentially known (though not written anywhere) that a variety of other results with slightly different tradeoffs can be achieved through similar uses of Linial-Saks [34] or refinements of Linial-Saks such as [37]. However, since all of these approaches treat the centralized approximation algorithm as a black box, none of them can achieve the same approximation ratio as the centralized algorithm without suffering a much worse (usually polynomial) round complexity that the $O(k \log n)$ that we achieve.

Subsequent work. After this work, [18] showed that in the CONGEST model any algorithm for constructing an α-approximation to the directed spanner problem requires $\tilde{\Omega}(\frac{\sqrt{n}}{\sqrt{\alpha}})$ number of rounds. This implies that there is a strict separation between LOCAL and CONGEST algorithms for approximating spanners.

5.2 Preliminaries and Notation

The distributed setting we will be considering is the LOCAL model [82], in which time passes in synchronous rounds and in each round every node can send an arbitrary message of unbounded size to each of its neighbors in the underlying graph $G = (V, E)$ (as always, we will let $n = |V|$ and $m = |E|$). This is in contrast to the CONGEST model, where nodes can only send a message of size $O(\log n)$ to each of their neighbors in each round. We are not focusing on the CONGEST model in this chapter. We will assume that all nodes know n (or at least know a constant approximation of n). Usually in this model the communication graph is the same as the graph of computational interest; e.g., we will be trying to compute a spanner of the communication graph itself. But for some applications we will want the graph to be directed, in which case we make the standard assumption that communication is bidirectional: the graph for which we are trying to compute a convex relaxation / network design problem is directed, but messages can be sent in both directions across a link. In other words, the communication graph is just the undirected version of the given directed graph.

For any pair of nodes $u, v \in V$ we define $d(u, v)$ to be the distance between u and v in the communication graph (i.e. the length of a shortest path between u and v regardless of edge directions). We define $B(u, k)$ to be an undirected ball of radius k from u in the communication graph. More precisely, $B(u, k) = \{w \in V \mid d(u, w) \leq k\}$. If x is a vector then we use x_i to denote the i'th component of x. Most of the time our vectors will be indexed by edges in a graph, in which case we will also use the notation $(x_e)_{e \in E}$.

Given a partition of the vertices V of a graph, we will refer to each part of the partition as a "cluster". For any graph $G = (V, E)$ and set $S \subseteq V$, we let $E(S)$ denote the set of edges in the subgraph induced by S, i.e., $E(S) = \{(u, v) \in E \mid u, v \in S\}$. We will frequently need "restrictions" of vectors to induced subgraphs, so for any vector $x \in \mathbb{R}^m$, we define $x^S = (x^S_e)_{e \in E}$ to be the vector in $\mathbb{R}^m$ where $x^S_e = 0$ if $e \notin E(S)$ and $x^S_e = x_e$ if $e \in E(S)$.

5.3 Padded decompositions

We will now define and give an algorithm to construct *padded decompositions*, which are one of the key technical tools that we will use when designing algorithm to solve distance-bounded network design convex programs. In this section all graphs are undirected and all distances are with respect to this undirected graphs (in fact, the definition and our algorithm work more generally for any metric space). Recall that $B(u, k)$ denotes the undirected ball of radius k from node u (in the communication graph), and $diam(C) = \max_{u,v \in C} d_G(u, v)$, which is often called the weak diameter. Intuitively, a (k, ϵ)-padded decomposition partitions a graph into clusters, where nodes in each cluster are not too far in the original graph, and balls of a radius k are preserved with probability at least $1 - \epsilon$.

Definition 5.3.1. Given an *undirected* graph G, a (k, ϵ)-padded decomposition, where

$0 < \epsilon \leq 1$, is a probability measure μ over the set of graph partitions (clusterings) that has the following properties:

1) For every $P \in \text{supp}(\mu)$,[3] and every cluster $C \in P$, we have: $diam(C) \leq O((k/\epsilon)\log n)$.

2) For every $u \in V$, it holds that $\Pr(\exists C \in P \mid B(u,k) \subseteq C) \geq 1-\epsilon$, i.e. the probability that all nodes in $B(u,k)$ are in the same cluster is at least $1-\epsilon$.

This notion of padded decompositions is standard in metric embeddings and approximation algorithms [53, 65], but to the best of our knowledge has not yet been used in distributed algorithms. We first use a centralized algorithm (Algorithm 9) to sample from a (k,ϵ)-padded decomposition, and then describe how it can be implemented in the *LOCAL* model. Algorithm 9 and its analysis are similar to a partitioning algorithm proposed in [7], which was shown to have a low probability of separating nodes in a close neighborhood.

Algorithm 9: Sampling from a (k,ϵ)-padded decomposition of $G=(V,E)$.

```
1 Let $\pi : V \to [n]$ be an arbitrary bijection from $V$ to $[n]$, and let $r = (\frac{2}{\epsilon})k$.
2 for $v \in V$ do
3     Sample $z_v$ independently from a distribution with probability density
        function $p(z_v) = \left(\frac{n}{n-1}\right)\frac{e^{-z_v/r}}{r}$.
4     Set the radius $r_v = \min(z_v, r\ln n + k)$.
5 for $u \in V$ do
6     Node $u$ joins cluster $C(v)$, such that
          $d(v,u) \leq r_v \wedge (\pi(v) < \pi(w)\ \forall w \neq v$ s.t. $d(w,u) \leq r_w)$. // Node $u$
          joins the cluster $C(v)$, with cluster center $v$, which is
          the first node in the permutation where $d(v,u) \leq r_v$.
```

For any partition P constructed by Algorithm 9, each cluster is clearly $C(v)$ for some $v \in V$. We call this special node v the *center* of cluster $C(v)$. Later, we will use the center of each cluster for solving locally defined convex programs.

[3]By $\text{supp}(\mu)$ we mean the set of partitions that have non-zero probability.

Lemma 5.3.1. *Algorithm 9 partitions a given undirected graph $G = (V, E)$ into a partition P such that P is sampled from a (k, ϵ)-padded decomposition.*

Proof. The first property in Definition 5.3.1 is directly implied by the definition of r_v for all nodes $v \in V$. For the second property we consider an arbitrary node $u \in V$, and compute the probability that the ball $B(u, k)$ is not in any of the clusters in P. Consider an arbitrary value $1 \leq t \leq n$, let $v \in V$ be the node such that $t = \pi(v)$, and let $z = z_v$ be the real number sampled by v. Also, for any $x, y \in V$, let $\tilde{d}(x, y) = \min(d(x, y), r \ln n + k)$. Let us also order the clusters based on their center's position in the permutation, so that C_t is the cluster corresponding to $t = \pi(v)$ (i.e. v is the cluster center of C_t). We define X_t to be the event that if $B(u, k)$ is not in the first $t - 1$ clusters, then it is also not in any of the remaining clusters. We provide a recursive bound on X_t based on X_{t+1}. Then we will get the second property once we show $\Pr(X_0) \leq \epsilon$. We need to define the following events:

- A_t : $B(u, k)$ does not intersect with any of the clusters $C_1, .., C_{t-1}$.
- $M_t^{cut} : (\tilde{d}(v, u) - k \leq z < \tilde{d}(v, u) + k \mid A_t)$.
- $M_t^{ex} : (z < \tilde{d}(v, u) - k \mid A_t)$.
- $X_t : (\nexists j \geq t : B(u, k) \subseteq C_j \mid A_t)$.

In other words, conditional on the event that $B(u, k)$ is not in any of the first $t - 1$ clusters, either $B(u, k) \subseteq C_t$, or else one the following two events will occur: M_t^{cut} is the event that $B(u, k)$ partially intersects C_t, and M^{ex} is the event that $B(u, k)$ does not intersect C_t.

Now the event X_t occurs only when either M_t^{cut} occurs or both M_t^{ex} and X_{t+1} occur (i.e. when $B(u, k)$ is not in C_t or any of the next clusters). Hence we can write $\Pr(X_t) \leq \Pr(M_t^{cut}) + \Pr(M_t^{ex}) \Pr(X_{t+1})$. Recall that z is independently sampled from the density function $p(z_v) = \left(\frac{n}{n-1}\right)\frac{e^{-z_v/r}}{r}$, and thus M^{cut} can be written as follows:

$$\Pr(M_t^{cut}) = \int_{\tilde{d}(v,u)-k}^{\tilde{d}(v,u)+k} p(z)d_z = \left(\frac{n}{n-1}\right)\left(1 - e^{-2k/r}\right)e^{-(\tilde{d}(v,u)-k)/r}$$
$$\leq \left(\frac{n}{n-1}\right)\frac{2k}{r}e^{-(\tilde{d}(v,u)-k)/r}.$$

Similarly, we can write,

$$\Pr(M_t^{ex}) = \int_0^{\tilde{d}(v,u)-k} p(z)d_z = \left(\frac{n}{n-1}\right)\left(1 - e^{-(\tilde{d}(v,u)-k)/r}\right).$$

We now inductively prove that $\Pr(X_t) \leq (2 - \frac{t}{n-1})(\frac{2k}{r})$. If $t < n$ is the last step, then $\Pr(X_t) = 0$, and thus this bound clearly holds. Assume that the bound is true for X_{t+1}, we show that then it also holds for X_t. We have,

$$\Pr(X_t) \leq \Pr(M_t^{cut}) + \Pr(M_t^{ex})\Pr(X_{t+1})$$
$$\leq \left(\frac{n}{n-1}\right)\left(\frac{2k}{r}\right)\left(1 + \frac{n-t-2}{n-1}\left(1 - e^{-(\tilde{d}(v,u)-k)/r}\right)\right).$$

Since $e^{-(\tilde{d}(v,u)-k)/r} \geq e^{-(\ln n)} \geq 1/n$, we get that $\Pr(X_t) \leq \left(2 - \frac{t}{n-1}\right)\left(\frac{2k}{r}\right)$. The second property is then implied by the fact that $\Pr(X_0) \leq \frac{2k}{r} = \frac{2k}{2k(1/\epsilon)} = \epsilon$. □

We will now use an idea similar to the one used in [28] to make Algorithm 9 distributed. In [28] they only considered the special case of $k = 1$ and $\epsilon = 1/2$, which is why we cannot simply use their result as a black box.

Lemma 5.3.2. *There is an algorithm in the LOCAL model that runs in $O(\frac{k}{\epsilon}\ln n)$ rounds and samples from a (k, ϵ)-padded decomposition (so every node knows the cluster that it is in).*

Proof. Without loss of generality, we assume that all nodes have unique IDs[4]. The sequence of IDs in ascending order will determine the permutation π used in Algorithm 9, i.e. if $\text{ID}_u < \text{ID}_v$ then $\pi(u) < \pi(v)$. The algorithm proceeds as follows until all nodes

[4]We assume this since nodes can each draw an ID from a suitably large space, so the probability of a collision is small enough that it does not affect the guarantees required by a padded decomposition.

have been assigned to a cluster: each node $u \in V$ chooses a radius r_u based on the distribution defined in Algorithm 9. Then every $u \in V$ simultaneously sends a message containing ID_u to all nodes in $B(u, r_u)$. After receiving all the messages, each node chooses the node with the smallest ID as the cluster center. Then Lemma 5.3.1 implies that the clusters satisfy the properties of a (k, ϵ)-padded decomposition. Since the radius that each node chooses is $O((k/\epsilon) \log n)$, and each node only communicates with nodes within its radius, the running time in the $LOCAL$ mode is $O((k/\epsilon) \log n)$. $\square$

5.4 Distributed distance bounded network design convex programming

In this section we prove Theorem 5.1.1, giving an algorithm similar to [28] which can almost optimally solve distance-bounded network design convex programs. We first make all definitions formal in Section 5.4.1, and in particular define formally the class of objective functions where our results hold. Then in Section 5.4.2 we give a distributed algorithm which solves these programs up to arbitrarily small error.

5.4.1 Distance bounded network design convex programs

We will first describe a general class of objective functions that our algorithm applies to. For a graph $G = (V, E)$ and a set $S \subseteq V$, we let $E(S)$ denote the set of edges in the subgraph of G induced by S. Recall that for a vector $x \in \mathbb{R}^m$ (where $m = |E|$), we define $x^S = (x^S_e)_{e \in E} \in \mathbb{R}^m$ to be the vector where $x^S_e = 0$ if $e \notin E(S)$ and $x^S_e = x_e$ if $e \in E(S)$.

Definition 5.4.1. Given a graph $G = (V, E)$, a function $g : \mathbb{R}^m \mapsto \mathbb{R}$ is *convex partitionable* with respect to G if g is a non-decreasing[5] and convex function with the following property: for all partitions $\sigma = \{\sigma_1, ..., \sigma_\ell\}$ of nodes in V, there exists a

[5] Let $f(x_1, ..., x_k)$ be a multivariate function. We will say f is nondecreasing if the following holds: if $x_i \leq x'_i$ for all $1 \leq i \leq k$, then $f(x_1, ..., x_k) \leq f(x'_1, ..., x'_k)$.

non-decreasing function $h_\sigma : \mathbb{R}^\ell \mapsto \mathbb{R}$, s.t. $g(x) = h_\sigma(g(x^{\sigma_1}), g(x^{\sigma_2}), ..., g(x^{\sigma_\ell}))$ for all $x = (x_e)_{e \in E}$ where $x_e = 0$ for any edge e with endpoints in different clusters of σ (equality does not need to hold for vectors x with nonzero values on edges between clusters).

Convex partitionable functions for graphs are a natural class of functions for distributed computing purposes. Moreover, this class includes many types of objective functions that are of interest in network design problems, including p-norms and linear functions. For example, if the function g is the p-norm with $p \in \mathbb{Z}_{\geq 0}$, then it is easy to verify that by setting the function h_σ to also be the p-norm for any partition σ of V, the conditions of Definition 5.4.1 are satisfied. Note, since we consider non-negative values, an unweighted sum is just the 1-norm, and the max function is the infinity norm, and hence they will also satisfy the conditions of Definition 5.4.1. Similarly, in case of linear functions, it is easy to see that conditions of Definition 5.4.1 are satisfied by setting h_σ to be the *unweighted* sum.

There are also other, less trivial examples. For example, it is not hard to show the p-norm of the *degree vector* (rather than just the edge vector) is also convex partitionable with respect to G. An important special case of this is the ∞-norm of the degree vector, i.e., the maximum degree. For an integral vector $x \in \mathbb{R}^m$ we can write $g(x) = \max_{v \in V} \deg(v)$. By generalizing this notation to all $x \in \mathbb{R}^m$, we can define fractional node degrees as $\deg(v) = \sum_{u:(v,u) \in E} x_{(v,u)}$[6].

Lemma 5.4.1. *Given a graph $G = (V, E)$, the function $g(x) = \max_{v \in V}(\sum_{u:(v,u)\in E} x_{(v,u)})$ is convex partitionable w.r.t. G.*

Proof. Let $\sigma = \{\sigma_1, ..., \sigma_\ell\}$ be a partition of nodes in V. For all $1 \leq i \leq \ell$, we have $g(x^{\sigma_i}) = \max_{v \in \sigma_i}(\sum_{u:(v,u)\in E} x^{\sigma_i}_{(v,u)})$. Then we can set $h_\sigma(y) = \max_{i \in [\ell]}(y_i), y \in \mathbb{R}^\ell$,

[6]Here we are considering out-degree of nodes in a directed graph. It is easy to see that Lemma 5.4.1 also holds in cases of in-degree only or sum of out-degree and in-degree. The latter is what are interested in for Section 5.5.

where y_i is the i-th coordinate of y. Let $\sigma(v) \in \sigma$ be the cluster that node v belongs to. For all $x = (x_{(u,v)})_{(u,v)\in E}$, where $x_{(u,v)} = 0$ for any $(u,v) \in E$ s.t. $\sigma(u) \neq \sigma(v)$, we have,

$$\begin{aligned} g(x) = \max_{v\in V}\left(\sum_{u:(v,u)\in E} x_{(v,u)}\right) &= \max_{\sigma_i\in\sigma}\left(\max_{v\in\sigma_i}\left(\big(\sum_{u:(v,u)\in E} x^{\sigma_i}_{(v,u)}\right)\right) \\ &= \max_{\sigma_i\in\sigma}\left(g\left(x^{\sigma_i}\right)\right) = h_\sigma(g(x^{\sigma_1}), g(x^{\sigma_2}), ..., g(x^{\sigma_\ell})). \end{aligned}$$

It is also easy to see that the function h_σ is convex and non-decreasing. Hence h_σ satisfies the conditions in Definition 5.4.1. $\square$

Now that this class of functions has been defined, we can formally define the class of distance-bounded network design convex programs.

Definition 5.4.2. Let $\mathcal{S} \subseteq V \times V$ be a set of pairs in the graph $G = (V, E)$, and for any pair $(u, v) \in \mathcal{S}$ let $\mathcal{P}_{u,v}$ be a set of paths from u to v, which we sometimes call the set of "allowed" paths. Let g be a non-decreasing convex-partitionable function of $x = (x_e)_{e\in E}$ with $g(\vec{0}) = 0$. Then we call a convex program of the following form a *distance bounded network design CP*:

$$\begin{aligned} \min \quad & g(x) & \\ s.t \quad & \sum_{P\in\mathcal{P}_{u,v}:e\in P} f_P \leq x_e & \forall(u,v)\in\mathcal{S}, \forall e\in E \\ & \sum_{P\in\mathcal{P}_{u,v}} f_P \geq 1 & \forall(u,v)\in\mathcal{S} \\ & x_e \geq 0 & \forall e\in E \\ & f_P \geq 0 & \forall(u,v)\in\mathcal{S}, \forall P\in\mathcal{P}_{u,v} \end{aligned}$$

As we will see in Section 5.5, many network design problems use linear (or convex) programming relaxations that satisfy the conditions of Definition 5.4.2. A key parameter of such a program is the length of the longest allowed path $D = \max_{(u,v)\in\mathcal{S}} \max_{p\in\mathcal{P}_{u,v}} \ell(p)$ (where $\ell(p)$ is the length of path p).

5.4.2 Distributed Algorithm

In order to solve these convex programs in a distributed manner, we will first use padded decompositions to form a local problem using a simple distributed algorithm. Let P be a partition sampled from a (k, ϵ)-padded decomposition (in particular, obtained by Lemma 5.3.2), where $0 < \epsilon \leq 1$. Recall that for each cluster $C \in P$, $E(C) = \{(u, v) \in E \mid u, v \in C\}$. We define $G(C)$ to be the subgraph induced by C.

Lemma 5.4.2. *For each cluster C sampled from a (k, ϵ)-padded decomposition, there is a distributed algorithm running in $O(\frac{k}{\epsilon} \log n)$ rounds so that every cluster center knows $G(C)$.*

Proof. The first property of (k, ϵ)-padded decompositions implies that for all nodes $u \in C$, we have $d(u, v) = O((k/\epsilon) \log n)$, where v is the center of cluster C. Each node $u \in C$ that determines v as the center of the cluster it belongs to, will send the information of its incident edges to v. Since there is no bound on the size of the messages being forwarded, this can be done in $O((k/\epsilon) \log n)$ time. $\square$

Let $\mathrm{CP}(G)$ be a distance bounded network design CP defined on graph $G = (V, E)$. We will define local convex programs based on a partition P of G that is sampled from a (D, λ)-padded decomposition. The value of $0 < \lambda \leq 1$ will be set later based on the parameters of our distributed algorithm. For each $C \in P$, let $\mathrm{CP}(C)$ be $\mathrm{CP}(G)$ defined on $G(C)$, but where only demands corresponding to any pair $(u, v) \in \mathcal{S}$ in which $B(u, D)$ is fully contained in C are included. We denote the set of these demands by $N(C)$, more precisely, $N(C) = \{(u, v) \in \mathcal{S} \mid B(u, D) \subseteq C\}$. The objective will then be to minimize $g(x) = g\left((x_e)_{e\in E(C)}\right)$. In other words $CP(C)$ is defined as follows:

$$\begin{aligned}
\min \quad & g(x) \\
s.t \quad & \sum_{P\in\mathcal{P}_{u,v}: e\in P} f_P \leq x_e & \forall (u,v)\in N(C), \forall e\in E(C) \\
& \sum_{P\in\mathcal{P}_{u,v}} f_P \geq 1 & \forall (u,v)\in N(C) \\
& x_e \geq 0 & \forall e\in E(C) \\
& f_P \geq 0 & \forall (u,v)\in N(C), \forall P\in\mathcal{P}_{u,v}
\end{aligned}$$

There is a technical subtlety about computing the function g on each cluster, which is the fact that a solution $\langle x^C, f^C\rangle$ of CP(C) is only defined on $G(C)$. While in practice x^C is a vector defined only on edges in $E(C)$, in our analysis, we will assume that x^C is a vector in $\mathbb{R}^{|E|}$ and $x^C_e = 0$ for all $e \notin E(C)$. This assumption does not impact the correctness of algorithm. The following lemma is similar to Lemma 3.8 in [28], and we show that it holds for our modified definition of local convex programs and for generalized objective functions that satisfy Definition 5.4.1.

Lemma 5.4.3. *Let $\langle x^*, f^*\rangle$ be an optimal solution of $CP(G)$ and let $x^{*C} = (x^*_e)_{e\in E(C)}$. For each cluster $C \in P$, let $\langle \tilde{x}^C, \tilde{f}^C\rangle$ be an optimal solution of CP(C). Then $g(\tilde{x}^C) \leq g(x^{*C})$.*

Proof. We argue that the vector $\langle x^{*C}, f^{*C}\rangle$, where $x^{*C}_e = x^*_e$ for all $e \in E(C)$ and $f^{*C}_p = f^*_p$ for all $p \in \mathcal{P}_{u,v}$, is a feasible solution to CP(C). By definition of $N(C)$ we have that for any $(u,v) \in N(C)$ all paths in $\mathcal{P}_{u,v}$ also appear in $G(C)$, and therefore $\langle x^{*C}, f^{*C}\rangle$ satisfies both capacity and flow constraints of CP(C) for pairs $(u,v) \in E(C)$ since they were satisfied in CP(G). Since we assumed that $\langle \tilde{x}^C, \tilde{f}^C\rangle$ is an optimal solution of CP(C), we get $g(\tilde{x}^C) \leq g(x^{*C})$. □

We now provide in Algorithm 10 a distributed algorithm for solving CP(G). The high level idea is the following: we partition the graph t times, have cluster centers solve CP(C) of their cluster using a sequential algorithm in each iteration, and then take an average over the solutions for each edge. Intuitively, for each edge, by averaging over local solutions for iterations in which the ball around that edge is in the same cluster, with high probability we get a feasible global solution. In the proof of Theorem 5.4.4, we will show that this solution gives a $(1+\epsilon)$-approximation solution to the LP, for an arbitrary $0 < \epsilon \leq 1$.

We assume that all nodes know the values of D and ϵ. Let $C_{u,i}$ denote the cluster that node u belongs to in the i-th iteration, and let $\langle x^{C_{u,i},i}, f^{C_{u,i},i} \rangle$ be the fractional CP solution of $C_{u,i}$, where $\langle x_e^{C_{u,i},i}, f_p^{C_{u,i},i} \rangle$ is the fractional CP value for $e = (u,v)$, and $p \in \mathcal{P}_{u,v}$. Since the objective is a function of edge vectors, what we mean by having a distributed solution to a distance bounded network design CP is that each node u will know the value x_e for all the edges e incident to u. It is not hard to see that the algorithm could be modified so that every node u can also know the flow value f_p for each path p.

Algorithm 10: Distributed algorithm for approximating distance bounded network design CPs.

```
1 Set $\lambda = \frac{\epsilon(1-\epsilon)}{(2-\epsilon)(1+\epsilon)}$ and $t = \left\lceil \frac{16(1-\frac{\epsilon}{2})(1+\epsilon)\ln n}{\epsilon^2} \right\rceil$.
2 Sample from $(D, \lambda)$-padded decompositions $t$ times by Lemma 5.3.2, and let $P_i$
    be the partition obtained in the $i$'th run.
3 For each cluster $C \in P_i$, the center of cluster $C$ computes $G(C)$ (see Lemma
    5.4.2).
4 The center of each cluster $C \in P_i$ solves $CP(C)$ and sends the solution
    $\langle x^{C,i}, f^{C,i} \rangle$ to all nodes $u \in C$.
5 for $e = (u,v) \in E$ do
6     Let $I_{u,v} = \{i \mid \exists C \in P_i : u, v \in C\}$.
      // these are the iterations in which both endpoints are in
         same cluster
7     $\tilde{x}_e \leftarrow \min(1, \frac{1+\epsilon}{t} \sum_{i \in I_{u,v}} x_e^{C_{u,i},i})$.
```

Theorem 5.4.4. *Algorithm 10 takes $O((D/\epsilon)\log n)$ rounds to terminate, and it will compute a solution of cost at most $(1+\epsilon)CP^*$ to a bounded distance network design*

CP (Definition 5.4.2) with high probability, where CP^* is the optimal solution and $0 < \epsilon \leq 1$. Moreover, if the convex program can be solved sequentially in polynomial time, then all of the node computations are also polynomial time.

Proof. **Correctness:** We first show that with high probability the values $\tilde{x}_e, e \in E$ form a feasible solution. Here we only need to show that a feasible solution for the flow values exist, and do not require nodes to compute these values. Let $I_u = \{i : \exists C \in P_i, B(u, D) \subseteq C\}$, i.e. I_u is the set of iterations in which $B(u, D)$ is contained in a cluster, and let $I_{u,v}$ be the set of iterations in which both u and v are in the same cluster. Since we need to implement Algorithm 10 in a distributed manner, we use $I_{u,v}$ in our implementation, while the analysis is based on I_u. We can do so since by definition we have $I_u \subseteq I_{u,v}$, for any $(u, v) \in E$.

For any $p \in \mathcal{P}_{u,v}$, we set the flow values to be $\tilde{f}_p = \frac{1}{|I_u|} \sum_{i \in I_u} f_p^{C_{u,i},i}$,i.e. $\tilde{f}_p$ is the average over local flows in iterations in which $B(u, k)$ is fully contained in a cluster. We will show that this gives a feasible flow. First we argue that enough flow is being sent. For all $(u, v) \in \mathcal{S}$, we have,

$$\sum_{p \in \mathcal{P}_{u,v}} \tilde{f}_p = \sum_{p \in \mathcal{P}_{u,v}} \frac{1}{|I_u|} \sum_{i \in I_u} f_p^{C_{u,i},i} = \frac{1}{|I_u|} \sum_{i \in I_u} \sum_{p \in \mathcal{P}_{u,v}} f_p^{C_{u,i},i} \geq \frac{1}{|I_u|} \sum_{i \in I_u} 1 \geq 1.$$

We have used the fact that for each $i \in I_u$ the solution corresponding to the CP of the cluster containing u satisfies the constraint that $\sum_{p \in \mathcal{P}_{u,v}: e \in p} f_p^{C_{u,i},i} \geq 1$, because for each such i we know that $(u, v) \in N(C)$.

Next, we will argue that the capacity constraints are also satisfied. The second property of (D, λ)-padded decompositions implies that for each iteration $1 \leq i \leq t$, we have $\Pr(i \in I_u) \geq 1 - \lambda = 1 - \frac{\epsilon(1-\epsilon)}{(2-\epsilon)(1+\epsilon)} = \frac{1}{(1-\frac{\epsilon}{2})(1+\epsilon)}$. By linearity of expectations we have $E[|I_u|] \geq t(1 - \lambda)$. Since each sampling is performed independently, by Chernoff bound for $\delta = \epsilon/2$, we get,

$$\begin{aligned}\Pr(|I_u| \le t(1-\lambda)(1-\delta)) &= \Pr\left(|I_u| \le \frac{t(1-\frac{\epsilon}{2})}{(1-\frac{\epsilon}{2})(1+\epsilon)}\right)\\ &= \Pr\left(|I_u| \le \frac{t}{(1+\epsilon)}\right) \le e^{-\frac{(\epsilon/2)^2(1-\lambda)t}{2}} \le e^{-2\ln n} = \frac{1}{n^2}.\end{aligned}$$

Hence by a union bound on all nodes we have that with high probability $|I_u| > t/(1+\epsilon)$. Therefore, for all $(u,v) \in \mathcal{S}, e \in E$, we have (w.h.p.),

$$\begin{aligned}\sum_{p \in \mathcal{P}_{u,v}: e \in p} \tilde{f}_p &= \sum_{p \in \mathcal{P}_{u,v}: e \in p} \frac{1}{|I_u|} \sum_{i \in I_u} f_p^{C_{u,i},i} = \frac{1}{|I_u|} \sum_{i \in I_u} \sum_{p \in \mathcal{P}_{u,v}: e \in p} f_p^{C_{u,i},i} \le \frac{1}{|I_u|} \sum_{i \in I_u} x_e^{C_{u,i},i}\\ &\le \min\left(1, \frac{1}{|I_u|} \sum_{i \in I_{u,v}} x_e^{C_{u,i},i}\right) \le \min\left(1, \frac{1+\epsilon}{t} \sum_{i \in I_{u,v}} x_e^{C_{u,i},i}\right) = \tilde{x}_e.\end{aligned}$$

Upper bound: We will now show that the upper bound holds. Let $\langle x^*, f^* \rangle$ be an optimal solution to CP(G). We have $\tilde{x}_e = \min(1, \frac{1+\epsilon}{t} \sum_{i \in I_e} x_e^{C_{u,i},i})$, and for each $e = (u,v)$ and $1 \le i \le t$, we set $\tilde{x}_e^i = x_e^{C_{u,i},i}$ if $i \in I_e$, and $\tilde{x}_e^i = 0$ otherwise. Note that $0 < (1+\epsilon)/t < 1$, and since g is a convex function and $g(\vec{0}) = 0$, by Jensen's inequality we have $g\left(\frac{1+\epsilon}{t}x\right) \le \frac{1+\epsilon}{t} g(x)$. Then for $\tilde{x} = (\tilde{x}_e)_{e \in E}$ we can write:

$$\begin{aligned}g(\tilde{x}) = g\left((\tilde{x}_e)_{e \in E}\right) &\le g\left(\frac{1+\epsilon}{t}\left(\sum_{i \in I_e} x_e^{C_{u,i},i}\right)_{e \in E}\right) \le \frac{1+\epsilon}{t} g\left(\left(\sum_{i \in I_e} x_e^{C_{u,i},i}\right)_{e \in E}\right)\\ &\le \frac{1+\epsilon}{t} g\left(\left(\sum_{i=1}^{t} \tilde{x}_e^i\right)_{e \in E}\right) \le \frac{1+\epsilon}{t} g\left(\sum_{i=1}^{t} \left(\tilde{x}_e^i\right)_{e \in E}\right) \le \frac{1+\epsilon}{t} \sum_{i=1}^{t} g\left(\left(\tilde{x}_e^i\right)_{e \in E}\right).\end{aligned}$$

In the final inequality, since g is convex, we used Jensen's inequality to take the sum out of the function. It is now enough to show that in each iteration i, it holds $g((\tilde{x}_e^i)_{e \in E}) \le g(x^*)$. Let $\tilde{x}^i = ((\tilde{x}_e^i)_{e \in E})$, and let $P_i = \{C_1, C_2, ..., C_\ell\}$ be the partition of V. Since g is a convex partitionable function w.r.t. G, there exists a nondecreasing and convex function $h : \mathbb{R}^m \mapsto \mathbb{R}$ for which we can write $g(\tilde{x}^i) = h_{P_i}(g(\tilde{x}^{i,C_1}), g(\tilde{x}^{i,C_2}), ..., g(\tilde{x}^{i,C_\ell}))$, since $\tilde{x}_e^i = 0$ by definition for edges which go between clusters (for simplicity we are denoting $\tilde{x}^{i^{C_j}}$ by $\tilde{x}^{i,C_j}$).

Recall that x^{*C} is the vector in which $x^{*C}_e = x^*_e$ for all $e \in E(C)$ and $x^{*C}_e = 0$ otherwise. By Lemma 5.4.3 we get that for all $C \in P_i$, $g(\tilde{x}^{i,C}) \le g(x^{*C})$. Now we consider a vector $\hat{x}$, defined by setting $\hat{x}_e = x^*_e$ for all edge e with both endpoints in the same cluster, and $\hat{x}_e = 0$ otherwise. Since we assumed h_{P_i} to be nondecreasing, we get,

$$\begin{aligned} g(\tilde{x}^i) &= h_{P_i}(g(\tilde{x}^{i,C_1}), g(\tilde{x}^{i,C_2}), ..., g(\tilde{x}^{i,C\ell})) \leq h_{P_i}(g(x^{*C_1}), g(x^{*C_2}), ..., g(x^{*C_\ell})) \\ &= h_{P_i}(g(\hat{x}^{C_1}, \hat{x}^{C_2}, ..., g(\hat{x}^{C_\ell})) = g(\hat{x}) \leq g(x^*). \end{aligned}$$

For the last inequality we have used the fact that g is non-decreasing, and that for all $e \in E$, $\hat{x}_e \leq x^*_e$ (since either $\hat{x}_e = x^*_e$ or $\hat{x}_e = 0$). By plugging this into the above inequalities, we will get $g(\tilde{x}) \leq \frac{1+\epsilon}{t} \sum_{i=1}^{t} g(\tilde{x}^i) \leq (1+\epsilon) g(x^*)$, which implies the claim that Algorithm 10 gives a $(1+\epsilon)$-approximation to the optimal solution.

Time Complexity: The decomposition step and sending the information within a cluster takes $O((D/\epsilon)\log n)$ rounds since the diameter of each cluster is $O((D/\lambda)\log n) = O((D/\epsilon)\log n)$. Since each decomposition is independent, we can do all of them in parallel, so steps 1-4 of the algorithm only take $O((D/\epsilon)\log n)$ rounds in total. Clearly the rest of the algorithm can be done in a constant number of rounds. Hence in total w.h.p. the algorithm will take $O((D/\epsilon)\log n)$ rounds. $\square$

5.5 Distributed Approximation Algorithms for Network Design

In this section, we will focus on several network design problems which can be approximated by first solving a convex relaxation using Algorithm 10 and then locally rounding the solution. For that purpose, we will describe how each problem has a distance bounded network design CP relaxation (Definition 5.4.2), and will then show that existing rounding schemes are local.

5.5.1 Directed k-Spanner

Dinitz and Krauthgamer [27] introduced a linear programming relaxation for Directed k-Spanner which is just a distance-bounded network design CP with demands pairs $\mathcal{S} = E$, allowed paths $\mathcal{P}_{u,v}$ which are the directed paths from u to v of length at most k, and objective function $g(x) = \sum_{e \in E} x_e$. They showed that this LP can be solved in polynomial time (approximately if k is non-constant). We will denote this LP by $LP(G)$. Clearly, LP(G) is a distance bounded network design CP with $D = k$. Hence, Theorem 5.4.4 implies that we can use Algorithm 10 to approximately solve this LP in $O(k \log n)$ rounds in the LOCAL model.

We now provide in Algorithm 11 a distributed rounding scheme that gives an $O(n^{1/2} \log n)$-approximation for Directed k-Spanner. This algorithm matches the best centralized approximation ratio known [14], and is just the obvious distributed version of the algorithm proposed in [14]. The difference is that here we truncate the shortest-path trees at depth k (as opposed to full shortest-path trees), and nodes choose whether to become a tree root independently (rather than chosen without replacement as in [14].

Algorithm 11: Distributed rounding algorithm for k-spanner.

```
   Input: Graph $G = (V, E)$, fractional solution $\langle x, f \rangle$ to LP($G$).
1  $E' = \emptyset, \forall v \in V : T_v^{in} = \emptyset, T_v^{out} = \emptyset$.
2  for $e \in E$ do
3      Add $e$ to $E'$ with probability $\min(n^{1/2} \cdot \ln n \cdot x_e, 1)$.
4  for $v \in V$ do
       // Random tree sampling
5      Choose $p$ uniformly at random from $[0, 1]$.
6      if $p < \frac{3 \ln n}{\sqrt{n}}$ then
7          $T_v^{in} \leftarrow$ shortest path in-arborescence rooted at $v$ truncated at depth $k$.
8          $T_v^{out} \leftarrow$ shortest path out-arborescence rooted at $v$ truncated at depth
             $k$.
9  Output $E' \cup (\cup_{v \in V} (T_v^{in} \cup T_v^{out}))$.// A node knows its portion of the
       output.
```

The following lemma is essentially from [14], with the proof requiring only slight technical changes due to the slightly different algorithms.

Lemma 5.5.1. *Given a directed graph G, $LP(G)$ as defined, and a fractional solution LP^* to $LP(G)$, the output of Algorithm 11 has size $O(n^{1/2} \cdot (n + LP^*) \log n)$.*

Proof. Let $N_{s,t}$ be the subgraph of G induced by the nodes on paths in $\mathcal{P}_{s,t}$. Edge $e \in E$ is called a *thick* edge if $|N_{s,t}| \geq n^{1/2}$, and otherwise it is called a *thin* edge. The set E' in Algorithm 11 satisfies the spanner property for all thin edges (as argued in [14]), and the random tree sampling phase satisfies the spanner property for the thick edges. Each thick edge (s,t) is spanned if at least one node in $N_{s,t}$ performs the random tree sampling. This probability is at least $1 - (1 - \frac{3 \ln n}{n^{1/2}})^{n^{1/2}} \geq 1 - 1/n^3$. Then a union bound on all the edges (of size at most $O(n^2)$) implies that w.h.p. all thick edges are spanned. We now argue that the output is an $O(n^{1/2} \log n)$-approximation algorithm: at most $O(n^{1/2} \log n)$ arborescences are chosen with high probability (each arborscence has $O(n)$ edges), and we argued that $|E'| = O(n^{1/2} \log n \cdot LP^*)$. Hence, the overall size of the output is $O(n^{1/2} \log n \cdot (n + LP^*))$. □

It is easy to see that this algorithm can be implemented in the *LOCAL* model.

Lemma 5.5.2. *Algorithm 11 runs in $O(k)$ time in the LOCAL model.*

Proof. Each node v in G has received the fractional solutions x_e corresponding to all edges $e \in E$ incident to v. The randomized rounding step can be performed locally: the node with the smaller ID flips a coin, and exchanges the coin flip result with its corresponding neighbors. In order to form T_i^{in} and T_i^{out}, v performs a distributed BFS algorithms by forming a shortest path tree while keeping track of the distance from v. When the distance counter reaches k, the tree construction terminates. □

We now immediately get our main result for Directed k-Spanner.

Corollary 5.5.3. *Algorithm 10 with $D = k$ along with the rounding scheme in Algorithm 11 yields an $O(n^{1/2} \ln n)$-approximation w.h.p. to* Directed k-Spanner *that runs in $O(k \log n)$ time in the LOCAL model and uses only polynomial-time computations at each node.*

Proof. We first run Algorithm 10 to solve LP(G) up to a constant factor (by setting $\epsilon = 1/2$), which takes time $O(k \log n)$ with high probability (Theorem 5.4.4). Since each cluster center can solve the local LP in polynomial time, all computations are polynomial time. We then use Algorithm 11 to round the fractional solutions of LP(G), which takes $O(k)$ time. Since the size of a k-spanner is at least $\Omega(n)$, Algorithm 11 then outputs an $O(n^{1/2} \ln n)$-approximation to the minimum (Lemma 5.5.1). $\square$

5.5.2 Basic 3-Spanner and Basic 4-Spanner

If the input graph is undirected then stronger approximations are possible. In particular, for stretch 3 and 4, there are $\tilde{O}(n^{1/3})$-approximations due to [14] (for stretch 3) and [31] (for stretch 4). Without going into details, both of these algorithms use the same LP relaxation as in Directed k-Spanner, but round the LP differently. So in order to give distributed versions of these algorithms, we only need to modify Algorithm 11 to use the appropriate rounding algorithm (and change some of the other parameters in the shortest-path arborescence sampling). Fortunately, both of these algorithms use rounding schemes which are highly local. Informally, rather than sample each edge independently with probability proportional to the (inflated) fractional value as in Algorithm 11, these algorithms sample a value independently at each *vertex* and then include an edge if a particular function of the values of the two endpoints (different in each of the algorithms) passes some threshold. Clearly this is a very local rounding algorithm: once we have solved the LP relaxation using Theorem 5.4.4, each node can draw its random value and then spend one more round to exchange a message with each of its neighbors to find out their values, and thus determine which of the edges have been included by the rounding. Thus the total running time is dominated by the time needed to solve the LP, which in these cases is $O(\log n)$ using Theorem 5.4.4.

5.5.3 Lowest-Degree k-Spanner

We now turn our attention to Lowest-Degree k-Spanner: Given a graph $G = (V, E)$ and a value k, we want to find a k-spanner that minimizes the maximum degree. We will use the relaxation and rounding scheme proposed by Chlamtáč and Dinitz [22]. The linear programming relaxation used in [22] is very similar to the Directed and Basic k-spanner LP relaxation described earlier, with the difference being that a new variable λ is added to represent the maximum degree, and so the objective is to minimize λ and constraints are added to force λ to upper bound the maximum fractional degree.

Theorem 5.5.4. *Given a graph $G = (V, E)$ (directed or undirected), and any integer $k \geq 1$ there is a distributed algorithm that w.h.p. computes an $\tilde{O}(\Delta^{(1-1/k)^2})$-approximation to the Lowest-Degree k-Spanner problem, taking $O(k \log n)$ rounds of the LOCAL model and using only polynomial-time computations at each node.*

Proof. It is easy to see that the LP relaxation proposed in [22] can be written as a distance bounded network design CP where the objective is $\max_{v \in V}(\deg(v)) = \max_{v \in V}\left(\sum_{u:\{v,u\} \in E} x_{\{v,u\}}\right)$ (we do not need to use their extra variable λ, since we can instead directly write the objective). Lemma 5.4.1 implies that this function is convex partitionable w.r.t. G, and hence the Lowest-Degree k-Spanner problem can be approximately solved (to within a constant factor) by using Algorithm 10 with $\epsilon = 1/2$. Next, we use the following rounding scheme proposed in [22]: each edge $e \in E$ is included in the spanner with probability $x_e^{1/k}$. It is clear that this can be done in a constant number of rounds, and hence the overall algorithm takes $O(k \log n)$

rounds (by Theorem 5.4.4) in the *LOCAL* model. In [22], it was shown that this leads to a $\tilde{O}(\Delta^{(1-1/k)^2})$-approximation solution of the problem. $\square$

5.5.4 Directed Steiner Network with Distance Constraints

It is well-known that the centralized rounding of [14] for Directed k-Spanner is more general than is actually stated in their paper. In particular, the randomized rounding for "thin" edges gives the same guarantee even when each demand has a possibly different distance constraint. This fact was used, e.g., in [23] in their algorithms for Distance Preserver, Pairwise k-Spanner, and Directed Steiner Network with Distance Constraints. The difficulty in extending the algorithm of [14] is not in the LP rounding, but rather because the arborescence sampling technique used to handle thick edges in [14] (and in our Algorithm 11) assumes that n is a lower bound on the optimal cost. This assumption is true for Directed k-Spanner, but false for variants where there might be a tiny number of demands. However, it is easy to see that if we assume the demand graph is spanning (i.e., assume that every node is an endpoint of at least one demand) then the optimal solution must have at least $n/2$ edges, and hence we can again just use [14] to get a $\tilde{O}(\sqrt{n})$-approximation for Directed Steiner Network with Distance Constraints as long as the demand graph is spanning.

While this is in the centralized setting, since our algorithm for Directed k-Spanner is just a lightly modified distributed version of [14] (the only difficulty in the distributed setting is solving the LP, which is why that is our main technical contribution), we can easily modify it to give the same approximation for Directed Steiner Network with Distance Constraints with spanning demand graphs. The only change is that we use $D = \max_{(u,v)\in\mathcal{S}} L(u,v)$ instead of k when solving the linear programming relaxation (using Theorem 5.4.4) and when truncating the shortest-path arborescences that we sample (note that we have to assume that D

is global knowledge, which is reasonable for spanner problems and for Shallow-Light Steiner Network but may be less reasonable for other special cases of Directed Steiner Network with Distance Constraints). This implies Theorem 5.1.3, and all of the interesting special cases (Shallow-Light Steiner Network, Distance Preserver, Pairwise k-Spanner, etc.) which it includes.

5.6 Conclusion and Future Work

In this chapter, we heavily relied on the power of LOCAL model to send messages of unbounded size to solve certain network design LPs. One clear open problem is whether we can extend our results to others models, such as CONGEST, Congested Clique, MPC, or the PRAM model.

One component of our algorithm is a distributed padded decomposition algorithm. While for the LOCAL model we chose a simple algorithm, we could instead use a network decomposition algorithm based on [75], as was used in our neighborhood cover construction in Chapter 3. Such an algorithm can be implemented in all the models described efficiently. However, the second component of our algorithm, collecting the neighborhoods to form the LOCAL LPs, is more difficult to extend to other models, specially in CONGEST. However, in stronger models such as Congested Clique certain neighborhood collection techniques may be useful, however it is often only possible to explore *sparse* or limited size neighborhoods. Thus new techniques will be needed to overcome this challenge.

Chapter 6

Conclusion and Future Direction

In this chapter, we review the problems discussed, and describe several open problems. In summary, in this thesis we studied construction of (β, ϵ)-hopsets mainly in three models: the MPC model, the Congested Clique model, and the decremental (partially dynamic) model. We also used our hopset algorithms to construct distance oracles more efficiently in these models, and in some cases obtained improved algorithms for shortest path computation. We also use several techniques and showed their significance by utilizing them for various problems in different models. These techniques include low diameter decompositions, subsampling and clustering ideas (variations of Thorup-Zwick [88] type of clustering), and use of scaling techniques for obtaining improved running time in various models.

Larger Stretch Constructions. While hopsets were originally introduced for computing $(1+\epsilon)$-approximate shortest paths ([25]), they have since found many other applications, and have been generalized to construction with larger stretch but smaller hopbound/size [13, 36]. One natural open direction is to explore such hopsets with larger stretch, and hope to obtain a better tradeoff between construction time and hopbound. This problem is for the most part open in all of the models we discussed, and may have implications in obtaining much faster shortest path algorithms in these models, at the cost of a weaker accuracy.

One related direction is our recent work [17] in which we obtain an $\tilde{O}(\log\log k)$ round algorithm for constructing $O(k)$-spanners in low and linear memory MPC. This leads to a $(\log n^{1+o(1)})$-approximate all-pairs-shortest-path algorithm in $\tilde{O}(\log\log n)$ rounds of *linear memory MPC*. While this is exponentially faster than algorithms discussed in this thesis and other related algorithms translated from PRAM literature (e.g. [3, 70]), the approximation ratio is larger than desired. Hence a natural direction is to combine the hopset ideas used in this thesis, with contraction-based ideas used in [17] to obtain faster $O(k)$-approximation algorithms for shortest path computation in MPC, Congested Clique, or streaming.

Fully Dynamic Hopsets. In the context of dynamic algorithms, an intriguing direction is constructing and utilizing hopsets in the fully-dynamic model. There are many open problems in the fully dynamic settings and current bounds for shortest path computation is far from optimal. Moreover, to our knowledge hopsets are not yet explored in this model. Instead, recent results in fully-dynamic model heavily rely on algebraic techniques such as matrix multiplication. It would be interesting to see if such techniques can be combined with combinatorial ideas for constructing hopsets and emulators to obtain better bounds, as was used in a few results in parallel and distributed models ([47, 19, 42]).

Linear Programming in Other Models. As discussed in Chapter 5, the problem solving (non-positive) linear programs efficiently is open in almost all distributed models. We used networked decomposition to solve a very specific class of linear programs. An interesting open direction is to see if these techniques can be combined with other LP solving approaches such as the *multiplicative weight updates* method to extend our results to other models or solve a wider class of LPs.

Unifying the Models. Finally, we conclude this thesis by posing an open problem regarding unification of various models for graph problems. We noted that many techniques used for distance-based graph problems led to better results in various models. This has also been observed for various other local and global graph problems. A natural question is if there is a way to characterize such problems and present primitives that allow reductions between these different models. Any such characterizations would be helpful in further abstracting the models, and bringing different theory communities together.

Bibliography

[1] Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. *SIAM Journal on Computing*, 2018.

[2] Alexandr Andoni, Clifford Stein, Zhao Song, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. *arXiv preprint arXiv:1805.03055*, 2018.

[3] Alexandr Andoni, Clifford Stein, and Peilin Zhong. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 322–335, 2020.

[4] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proceedings of the Symposium on Foundations of Computer Science*. IEEE, 1990.

[5] Amy Babay, Emily Wagner, Michael Dinitz, and Yair Amir. Timely, reliable, and cost-effective internet transport service using dissemination graphs. In *37th IEEE International Conference on Distributed Computing Systems, (ICDCS)*, pages 1–12, 2017.

[6] Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 2016.

[7] Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *FOCS'96*, pages 184–193, 1996.

[8] Yair Bartal, John W. Byers, and Danny Raz. Global optimization using local information with applications to flow control. In *FOCS*, pages 303–312, 1997.

[9] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007.

[10] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

[11] Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[12] Soheil Behnezhad, Mahsa Derakhshan, and MohammadTaghi Hajiaghayi. Brief announcement: Semi-mapreduce meets congested clique. *arXiv preprint arXiv:1802.10297*, 2018.

[13] Uri Ben-Levy and Merav Parter. New (α, β) spanners and hopsets. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1695–1714. SIAM, 2020.

[14] Piotr Berman, Arnab Bhattacharyya, Konstantin Makarychev, Sofya Raskhodnikova, and Grigory Yaroslavtsev. Improved approximation for the directed spanner problem. In *ICALP Part I*, pages 1–12, 2011.

[15] Aaron Bernstein. Fully dynamic $(2+\varepsilon)$ approximate all-pairs shortest paths with fast query and close to linear update time. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 693–702. IEEE, 2009.

[16] Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 1355–1365. SIAM, 2011.

[17] Amartya Shankha Biswas, Michal Dory, Mohsen Ghaffari, Slobodan Mitrović, and Yasamin Nazari. Massively parallel algorithms for distance approximation and spanners. 2021.

[18] Keren Censor-Hillel and Michal Dory. Distributed spanner approximation. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 139–148, 2018.

[19] Keren Censor-Hillel, Michal Dory, Janne H Korhonen, and Dean Leitersdorf. Fast approximate shortest paths in the congested clique. In *Proceedings of Symposium on Principles of Distributed Computing*. ACM, 2019.

[20] Shiri Chechik. Approximate distance oracles with constant query time. In *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663. ACM, 2014.

[21] Shiri Chechik. Near-optimal approximate decremental all pairs shortest paths. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 170–181. IEEE, 2018.

[22] Eden Chlamtác and Michael Dinitz. Lowest-degree k-spanner: Approximation and hardness. *Theory of Computing*, 12(1):1–29, 2016.

[23] Eden Chlamtác, Michael Dinitz, Guy Kortsarz, and Bundit Laekhanukit. Approximating spanners and directed steiner forest: Upper and lower bounds. In *SODA*, 2017.

[24] Edith Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 1998.

[25] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *Journal of the ACM (JACM)*, 2000.

[26] Michael Dinitz, Guy Kortsarz, and Ran Raz. Label cover instances with large girth and the hardness of approximating basic k-spanner. *ACM Trans. Algorithms*, 12(2):1–16, 2016.

[27] Michael Dinitz and Robert Krauthgamer. Directed spanners via flow-based linear programs. In *STOC'11*, pages 323–332, 2011.

[28] Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *PODC'11*, pages 169–178, 2011.

[29] Michael Dinitz and Yasamin Nazari. Distributed distance-bounded network design through distributed convex programming. In *Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[30] Michael Dinitz and Yasamin Nazari. Massively parallel approximate distance sketches. *OPODIS*, 2019.

[31] Michael Dinitz and Zeyu Zhang. Approximating low-stretch spanners. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, 2016.

[32] Yevgeniy Dodis and Sanjeev Khanna. Design networks with bounded pairwise distance. In *STOC '99*, pages 750–759, 1999.

[33] Michal Dory and Merav Parter. Exponentially faster shortest paths in the congested clique. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 59–68, 2020.

[34] Michael Elkin. Personal Communication, 2017.

[35] Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Almost shortest paths and pram distance oracles in weighted graphs. *arXiv preprint arXiv:1907.11422*, 2019.

[36] Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Almost shortest paths and pram distance oracles in weighted graphs. *arXiv preprint arXiv:1907.11422*, 2019.

[37] Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition: Extended abstract. In *PODC '16*, pages 211–216, 2016.

[38] Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 128–137. IEEE, 2016.

[39] Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 235–244. ACM, 2016.

[40] Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 2019.

[41] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in rnc. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 333–341, 2019.

[42] Michael Elkin and Ofer Neiman. Centralized and parallel multi-source shortest paths via hopsets and fast matrix multiplication. *arXiv preprint arXiv:2004.07572*, 2020.

[43] Michael Elkin and Ofer Neiman. Near-additive spanners and near-exact hopsets, a unified view. *arXiv preprint arXiv:2001.07477*, 2020.

[44] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 531–543. Springer, 2004.

[45] Patrik Floréen, Marja Hassinen, Joel Kaasinen, Petteri Kaski, Topi Musto, and Jukka Suomela. Local approximability of max-min and min-max linear programs. *Theory of Computing Systems*, 2011.

[46] Greg N Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[47] Stephan Friedrichs and Christoph Lenzen. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)*, 65(6):43, 2018.

[48] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, January 1983.

[49] Alan Gibbons and Paul Spirakis. *Lectures in parallel computation*, volume 4. Cambridge University Press, 1993.

[50] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *International Symposium on Algorithms and Computation*, pages 374–383. Springer, 2011.

[51] Martin Grötschel, Lászlo Lovász, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer, 1988.

[52] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 471–475, 2001.

[53] Anupam Gupta, Mohammad T. Hajiaghayi, and Harald Räcke. Oblivious network design. In *SODA '06*, pages 970–979, 2006.

[54] James W Hegeman and Sriram V Pemmaraju. Lessons from the congested clique applied to mapreduce. *Theoretical Computer Science*, 2015.

[55] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 146–155, 2014.

[56] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 489–498. ACM, 2016.

[57] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 21–30. ACM, 2015.

[58] Shang-En Huang and Seth Pettie. Thorup–zwick emulators are universally optimal hopsets. *Information Processing Letters*, 142:9–13, 2019.

[59] Adam Karczmarz and Jakub Lacki. Simple label-correcting algorithms for partially dynamic approximate shortest paths in directed graphs. In *3rd Symposium on Simplicity in Algorithms, SOSA@SODA 2020*. SIAM, 2020.

[60] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[61] M. Reza Khani and Mohammad R. Salavatipour. Improved approximations for buy-at-bulk and shallow-light k-steiner trees and (k,2)-subgraph. *Journal of Combinatorial Optimization*, 31(2):669–685, Feb 2016.

[62] Valerie King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 81–89. IEEE, 1999.

[63] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 391–410. Society for Industrial and Applied Mathematics, 2015.

[64] Philip N Klein and Sairam Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 1997.

[65] Robert Krauthgamer, James R. Lee, Manor Mendel, and Assaf Naor. Measured descent: A new embedding method for finite metrics. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, pages 434–443, 2004.

[66] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *SODA '06*, pages 980–989, 2006.

[67] Jakub Łącki and Yasamin Nazari. Near-optimal decremental approximate multi-source shortest paths. *arXiv preprint arXiv:2009.08416*, 2020.

[68] Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proceedings of the ACM Symposium on Principles of Distributed computing*. ACM, 2013.

[69] Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 381–390. ACM, 2013.

[70] Jason Li. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 308–321, 2020.

[71] Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, Dec 1993.

[72] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in o (log log n) communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005.

[73] Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006)*, pages 109–118. IEEE, 2006.

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

[75] Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the ACM Symposium on Parallelism in algorithms and architectures*. ACM, 2013.

[76] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the ACM Symposium on Theory of Computing*. ACM, 2014.

[77] Yasamin Nazari. Sparse hopsets in congested clique. In *Conference on Principles of Distributed Systems (OPODIS)*, 2019.

[78] Christos H. Papadimitriou and Mihalis Yannakakis. Linear programming without the matrix. In *STOC '93*, pages 121–129, 1993.

[79] Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2019.

[80] Mihai Patrascu and Liam Roditty. Distance oracles beyond the thorup–zwick bound. *SIAM Journal on Computing*, 43(1):300–311, 2014.

[81] Mihai Patrascu, Liam Roditty, and Mikkel Thorup. A new infinity of distance oracles for sparse graphs. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 738–747. IEEE, 2012.

[82] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, 2000.

[83] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[84] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *PODC'87*, pages 77–85, 1987.

[85] Liam Roditty and Uri Zwick. Dynamic approximate all-pairs shortest paths in undirected graphs. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 499–508. IEEE, 2004.

[86] Atish Das Sarma, Michael Dinitz, and Gopal Pandurangan. Efficient distributed computation of distance sketches in networks. *Distributed Computing*, 28(5):309–320, 2015.

[87] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981.

[88] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.

[89] Mikkel Thorup and Uri Zwick. Spanners and emulators with sublinear distance errors. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 802–809, 2006.

[90] Christian Wulff-Nilsen. Approximate distance oracles with improved query time. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 539–549. SIAM, 2013.