Processing Massive Graphs under Limited Visibility

Chhaya Trehan

A thesis submitted for the degree of
Doctor of Philosophy

Department of Mathematics
London School of Economics
and Political Science

February 2023

Declaration

I certify that the thesis I have presented for examination for the MPhil/PhD degree of the London School of Economics and Political Science is solely my own work, apart from Chapters 2 to 7 which are joint works with collaborators from England and Israel. Chapters 2 to 5 are joint work with Michael Elkin from Ben-Gurion University of the Negev, Beer-Sheba, Israel. Chapter 6 is joint work with Oded Lachish and Felix Reidl from Birkbek College, University of London and Chapter 7 is joint work with Tugkan Batu from the London School of Economics & Political Science.

Abstract

Graphs are one of the most important and widely used combinatorial structures in mathematics. Their ability to model many real world scenarios which involve a large network of related entities make them useful across disciplines. They are useful as an abstraction in the analysis of networked structures such as the Internet, social networks, road networks, biological networks and many more. The graphs arising out of many real world phenomenon can be very large and they keep evolving over time. For example, Facebook reported over 2.9 billion monthly active users in 2022. Another very large and dynamic network is the human brain consisting of around 10^{11} nodes and many more edges. These large and evolving graphs present new challenges for algorithm designers. Traditional graph algorithms designed to work with centralised and sequential computing models are rendered useless due to their prohibitively high resource usage. In fact one needs huge amounts of resources just to read the entire graph. A number of new theoretical models have been devised over the years to keep up with the trends in the modern computing systems capable of handing massive input datasets. Some of these models such as streaming model and the query model allow the algorithm to view the graph piecemeal. In some cases, the model allows the graph to be processed by a set of interconnected computing elements such as in distributed computing. In this thesis we address some graph problems in these non-centralised, non-sequential models of computing with a limited access to the input graph.

Specifically, we address three different graph problems, each in a different computing model. The first problem we look at is the computation of approximate shortest paths in dynamic streams. The second problem deals with finding kings in tournament graphs, given query access to the arcs of the tournament. The third and the final problem we investigate is a local test criteria for testing the expansion of a graph in the distributed CONGEST model.

Acknowledgements

First and foremost, I would like to thank my supervisor, Tugkan Batu, for his support and enthusiasm. Working with him has been an insightful and rewarding experience. I would also like to thank my co-authors Michael Elkin, Oded Lachish and Felix Reidl. My special thanks to Michael Elkin for offering me a summer internship with him. I wrote the first paper of my PhD with him and I enjoyed every bit of it. I thank him for answering my numerous emails even in the busiest of times and reading and improving my drafts, time and again. I cannot thank Oded enough for taking time for endless discussions on Zoom and answering all my questions patiently.

Thanks should also go to my family and friends for their constant support and encouragement. I thank my late father every single day for instilling a love of reading and learning in me. Many thanks to my mother for always being my source of solace. I am immensely grateful to my husband for his moral support and confidence in me. Thanks to him for all the wonderful meals he cooked for me while I was busy chasing deadlines. Finally, I would be remiss in not mentioning my little dog who always stayed put by my side and spent as many sleepless nights as I did while writing this thesis.

Contents

1

Introduction

1.1 Graph Processing with Limited Visibility

Ever growing (large), ever changing (dynamic) - these are some of the defining characteristics of the data that modern day computing systems have to process. The data produced by many real world applications contains valuable information, but its sheer size, dynamicity and susceptibility to errors makes it hard to analyze. Often, one is required to compute or test certain properties of the input without the ability to read it in its entirety due to resource constraints. Furthermore, the data to be processed may be distributed among loosely interacting computing units with limited resources and possibly faulty components. Various theoretical frameworks (such as query model of property testing) and models of computing (such as distributed, parallel, streaming etc.) have been devised to address the challenges involved in modern day computing. Each of these frameworks and models define their own cost metrics for analyzing the algorithms designed to work with them.

This thesis is concerned with the design and analysis of efficient algorithms for processing large (or large and dynamic) graphs in non-sequential, non-centralized frameworks and models of computing. One can think of a graph as a representation of relationships among a set of objects. The set of objects is represented by the vertices of the graph and the relationships between these objects are represented by the edges of the graph. Note that one needs to invest resources (storage capacity, computation time or communication; depending on the model) linear in the size of the underlying input graph to even read it fully. For massive graphs with billions

of vertices and trillions of edges, this cost may be prohibitively large. In all the frameworks and models we deal with in this thesis, at any given time, any single computing unit involved has only a *limited view* of the underlying input graph, due to resource constraints, i.e., it can only see a part of the input graph but not the entire graph. We use the umbrella term *limited visibility models* to refer to such frameworks and models. It is natural to expect algorithms designed to work in these models to solve the problem at hand with cost sub-linear in the size of the input graph. Often one can only hope to solve the problem approximately given the resource constraints.

Specifically, we address three graph processing problems, each in a different limited visibility model. In the subsequent sections, we describe each of these problems along with the description of the corresponding limited visibility model and its associated cost metrics. To start with, we first give an outline of the thesis in the following section and a short overview of specific problems follows after that.

1.2 Thesis outline

The contribution of this thesis consists of three parts: As a first result (**Part I**), we present algorithms for computing approximate shortest paths in graphs in dynamic streaming model. In the second part (**Part II**), we give a new query strategy for finding a king (see Section 6.1 for definition) in a tournament. In the last part (**Part III**), we present a distributed algorithm for testing the expansion of a graph.

1.3 Approximate Shortest Paths in Dynamic Streams (Part I)

In the streaming model of computation, the vertices of the graph are known to the algorithm and the edges are revealed one at a time. The model comes in two variations. In the *insertion-only* streaming model, the edges can only arrive and never get deleted. In the more general *dynamic* (also known as *turnstile*) streaming model, the edges can either arrive or get deleted as the stream progresses. The main challenge in these settings is that the storage capacity of the algorithm is limited. Typically it should be close to linear in the number of *vertices*, n (as op-

posed to being linear in the number of edges $m = |E|$). In particular, one usually allows space of $\tilde{O}(n)$, though it is often relaxed to $n^{1+o(1)}$, sometimes to $O(n^{1+\rho})$, for an arbitrarily small constant parameter $\rho > 0$, or even to $O(n^{1+\eta_0})$, for some fixed constant η_0, $0 < \eta_0 < 1$. Generally, the model allows several passes over the stream, and the objective is to keep both the number of passes and the space complexity of the algorithm in check. In **Chapters 2 to 4**, we present algorithms for finding $(1+\varepsilon)$-approximate shortest paths in an input graph in the dynamic streaming model. Here $(1+\varepsilon)$ is a multiplicative approximation factor. We sometimes refer to the approximation factor of an algorithm for computing approximate shortest paths as its *stretch*.

A lot of work has been done on the problem of computing approximate shortest paths in the dynamic streaming model (7; 31; 32; 40). However, all the existing solutions are doomed to incur a stretch-space tradeoff (due to an existing lower bound) of $2\kappa - 1$ versus $n^{1+1/\kappa}$, for an integer parameter κ. (In fact, existing solutions also incur an extra factor of $1+\varepsilon$ in the stretch for weighted graphs, and an additional factor of $\log^{O(1)} n$ in the space.) In **Chapters 2 to 4**, we show that $(1+\varepsilon)$-approximate single-source shortest paths can be computed in this setting with $\tilde{O}(n^{1+1/\kappa})$ space using just *constantly* many passes in unweighted graphs, and polylogarithmically many passes in weighted graphs (assuming ε and κ are constant). Moreover, in fact, the same result applies for multi-source shortest paths, as long as the number of sources is $O(n^{1/\kappa})$.

We achieve these results by devising efficient dynamic streaming constructions of combinatorial structures called $(1+\varepsilon,\beta)$-spanners and hopsets. These structures will be described in detail in the relevant chapters. For now it suffices to note that these structures are useful in computing approximate shortest distances and paths in graphs. We believe that the dynamic streaming constructions of these structures are of independent interest.

Chapter 2 provides an overview of the problem and presents necessary technical concepts and tools required to understand the rest of the chapters in **Part I**. **Chapter 3** presents an algorithm for constructing near-additive spanners, and shows how we use this algorithm to compute $(1+\varepsilon)$-approximate shortest paths in unweighted graphs. **Chapter 4** presents an algorithm for constructing

hopsets with constant hopbound and shows how we use this algorithm to compute $(1+\varepsilon)$-approximate shortest paths in weighted graphs.

We also devise a space efficient sampling technique as a sub-routine of our algorithms for $(1+\varepsilon,\beta)$-spanners and hopsets. This technique is described in detail in **Chapter 5**.

A short version of Chapters 2 to 5 has been published as a brief announcement in PODC 2022 (1) and a more complete version has been published as a full paper in Approx 2022 (2).

1.4 Query Strategies for Tournaments (Part II)

When a graph is prohibitively large, one can assume *query access* to the edges of the graph. Each query reveals an edge between a pair of objects and incurs a fixed cost. A natural question to ask in this setting is the number of queries required to identify certain structures in the input graph. This is the perspective we took in **Chapter 6** on finding a king in a tournament graph.

A *tournament* is an orientation of a complete graph on n vertices. A query in this setting reveals the direction of the arc between a given pair of vertices. We say that a vertex x in a tournament *controls* another vertex y if there exists a directed path of length at most two from x to y. A vertex is called a *king* if it controls every vertex of the tournament. Shen Sehng and Wu (67) showed that $\Omega(n^{4/3})$ queries are always necessary and provided an algorithm which reveals a king using $O(n^{3/2})$ queries. Despite the large gap between the upper and lower bound, not much progress has been made on the original question since 2003. The difficulty of this problem lies in its game-theoretic nature: we can alternatively think of it as an adversarial game where one player wants to identify a king by querying the arcs of the tournament while an adversary, who tries to delay the first player for as long as possible, can choose the orientation of queried arcs. The state of the art algorithm by Biswas et al. (62) proved that with $O(n^{4/3})$ queries, one can identify a *semi*-king, meaning a vertex which controls at least half of all vertices. We present a novel strategy which improves upon the number of controlled vertices: using $\tilde{O}(n^{4/3})$ queries we can identify a $(\frac{1}{2}+\frac{2}{17})$-king. We do so by showing

the existence of a sparse yet well-connected (with good expansion) query graph, which forces the adversary to give us a $(\frac{1}{2}+\frac{2}{17})$-king. The sparsity of the query graph keeps the query complexity in check and connectivity ensures that we do not miss the information required to identify the desired $(\frac{1}{2}+\frac{2}{17})$-king.

A version of Chapter 6 has been published in FSTTCS 2022 (3).

1.5 Graph Expansion Testing in Distributed Computing Model (Part III)

In the distributed model of computing, each vertex of the input graph is a processing unit itself and the edges of the graph represent communication links between the vertices. The processing proceeds in synchronous rounds. In each round, each vertex receives messages from its neighbours in the graph, performs some computation and sends messages to its neighbours. The messages sent by a vertex in a given round are received by its neighbours in the subsequent round. In the distributed *CONGEST* model, the size of each message is limited to $O(\log n)$ bits. The model allows processing to continue for several rounds and the objective is to keep the number of rounds in check without violating the constraint on the message size. In **Chapter 7** we present a distributed property testing algorithm for testing the expansion of a graph.

Given a graph G with vertex set V and edge set E, we say that G has expansion α, if every $U \subseteq V$ such that size of $|U| \leq |V|/2$ has at least $\alpha|U|$ neighbours. Here a vertex $v \in V \setminus U$ is a neighbour of U if it has at least one edge incident to some $u \in U$. For a given graph G and an expansion parameter α, the problem of testing whether G has expansion at least α or is far from having expansion at least $\Omega(\alpha^2)$ has been extensively studied (see (84; 74; 85; 88)) in the classic property testing model. The only existing distributed algorithm for this problem (78) first tries to build a rooted BFS spanning tree of the input graph. Having built a spanning tree, the algorithm performs a certain number of short random walks from a randomly chosen source vertex. When the walks end, each vertex collects some information about the walks and sends it to the root of the spanning tree. This whole process ultimately collects enough information at the root of the spanning tree for it to

decide whether the input graph has the desired expansion or not. If the input graph indeed has the desired expansion, the root vertex outputs *Accept* with sufficiently high probability. If the graph is far from having the desired expansion, it outputs *Reject* with sufficiently high probability.

This process of building a spanning tree and collecting information at the root to decide if the property holds or not takes a global and centralized view of the testing process. This motivates us to enquire if one can devise a *local* test criteria for expansion testing. The question of local verification of global properties has been well studied in the area of distributed verification and self-stabilization. Afek, Kutten and Young (68) and Awerbuch and Varghesh (71) suggested that one can check certain global predicates on a distributed network by using local means, i.e., by having each vertex check the state of all its neighbours. Sometimes one allows each vertex to check the local state of all the vertices within a constant distance from itself. This is the perspective we take in our work on distributed expansion testing. We run short random walks from a randomly selected vertex. After that every vertex in the graph acts as our designated tester (in contrast to having just the root of a spanning tree as our designated tester). Every designated distributed tester examines the information it collected from the random walks. If the graph has the desired expansion, all the designated testers output *Accept* with sufficiently high probability. If the graph is far from having the desired expansion, at least one of the designated testers outputs *Reject*. Avoiding the construction of a rooted spanning tree and information accumulation at the root has its advantages. It makes our algorithm time and communication efficient and avoids a single point of failure.

A version of Chapter 7 is currently under submission to a peer-reviewed conference.

In the remainder of **this chapter**, we summarize relevant prerequisites and notational conventions which we use throughout this thesis.

1.6 Preliminaries

The terminology and notational conventions we describe in this section apply to all the remaining chapters. In addition, each chapter has its own preliminary section with concepts and definitions specific to the problem at hand.

We denote by $G = (V,E)$ (or $G(V,E)$) an undirected, unweighted graph G with vertex set V and edge set E. An orientation of a graph G is a directed graph $\vec{G}$ obtained from G by replacing every one of its edges by a directed arc.

We denote by $G = (V,E,\omega)$ (or $G(V,E,\omega)$) an undirected, weighted graph with vertex set V, edge set E and weight function ω which maps every $v \in V$ to a real number.

We assume that the vertices of an n-vertex graph G have unique identifiers from the set $\{1,2,\ldots,n\}$.

All vectors $\vec{x} \in \mathbb{R}^n$ are column vectors (although for reasons of space, in examples we often write vectors as row vectors). For a vector $\vec{x} \in \mathbb{R}^n$, we denote by $\vec{x}^{\mathsf{T}}$ the transpose of $\vec{x}$. For two vectors $\vec{x}$ and $\vec{y}$ in $\mathbb{R}^n$, $\langle \vec{x}, \vec{y} \rangle$ denotes their inner product.

Inequalities between vectors hold component-wise. As usual, e_i denotes the ith unit vector, and $\mathbf{0}$ and $\mathbf{1}$ the all-zero- and all-one-vector, respectively, with dimension understood from context. For a matrix A, we denote by A^{T} its transpose. We write I for the identity matrix.

For a set X, we denote by $|X|$ its cardinality.

We denote by $\mathbb{Z}$ the set of all integer numbers. For integers q, r, $q \leq r$, the notation $[q,r]$ stands for $\{q, q+1, \cdots, r\}$. For an integer $k \geq 1$, $[k]$ denotes the set $\{1,\ldots,k\} = [1,k]$.

Note that all logarithms are to the base 2 unless explicitly stated otherwise. We use $\tilde{O}(f(n))$ as a shorthand for $O(f(n) \cdot polylogn)$.

We have now collected all the prerequisites that we require for the following chapters. Any additional concepts and definitions we need will be introduced as we go along.

Part I

A Graph Processing Problem in Dynamic Streaming Model

2

Approximate Shortest Paths-Nuts and Bolts

Computing approximate shortest paths in the dynamic streaming setting is a fundamental challenge that has been intensively studied during the last decade. Currently existing solutions for this problem either build a sparse multiplicative spanner of the input graph and compute shortest paths in the spanner offline, or compute an exact single source BFS tree.

Solutions of the first type are doomed to incur a stretch-space tradeoff of $2\kappa-1$ versus $n^{1+1/\kappa}$, for an integer parameter κ. (In fact, existing solutions also incur an extra factor of $1+\varepsilon$ in the stretch for weighted graphs, and an additional factor of $\log^{O(1)} n$ in the space.) The only existing solution of the second type uses $n^{1/2-O(1/\kappa)}$ passes over the stream (for space $O(n^{1+1/\kappa})$), and applies only to unweighted graphs.

In chapters 2 to 4 we show that $(1+\varepsilon)$-approximate single-source shortest paths can be computed in this setting with $\tilde{O}(n^{1+1/\kappa})$ space using just *constantly* many passes in unweighted graphs, and polylogarithmically many passes in weighted graphs (assuming ε and κ are constant). Moreover, in fact, the same result applies for multi-source shortest paths, as long as the number of sources is $O(n^{1/\kappa})$.

We achieve these results by devising efficient dynamic streaming constructions of $(1+\varepsilon,\beta)$-spanners and hopsets. We believe that these constructions are of independent interest.

On our way to these results, we also devise a new dynamic streaming algorithm for the 1-sparse recovery problem which can be used to implement an ℓ_0-sampler. Even though our algorithm for this task is slightly inferior to the existing algorithms of (33; 17), we believe that it is of independent interest.

2.1 Introduction

2.1.1 Graph Streaming Algorithms

One of the most common theoretical models for addressing the challenge of processing massive graphs is the *semi-streaming* model of computation (29; 6; 44). In this model, edges of an input n-vertex graph $G = (V, E)$ arrive one after another, while the storage capacity of the algorithm is limited. Typically it should be close to linear in the number of *vertices*, n (as opposed to being linear in the number of edges $m = |E|$). In particular, one usually allows space of $\tilde{O}(n)$, though it is often relaxed to $n^{1+o(1)}$, sometimes to $O(n^{1+\rho})$, for an arbitrarily small constant parameter $\rho > 0$, or even to $O(n^{1+\eta_0})$, for some fixed constant η_0, $0 < \eta_0 < 1$. Generally, the model allows several passes over the stream, and the objective is to keep both the number of passes and the space complexity of the algorithm in check.

The model comes in two main variations. In the first one, called *static* or *insertion-only* model (29), the edges can only arrive, and never get deleted. If the algorithm employs multiple passes, then the streams of edges observed on these passes may be permutations of one another, but are otherwise identical. In the more general *dynamic* (also known as *turnstile*) streaming setting (6), edges may either arrive or get deleted.

The study of graph problems in the dynamic streaming model has been blossoming in the last decade. A lot of research is devoted to building spectral and cut sparsifiers (59; 7; 51; 52; 53; 40). Numerous other graph problems such as connectivity and k-connectivity, MST, maximum matching, set cover, and counting small subgraphs were studied in (7; 6; 54; 55; 56; 57; 58).

2.1.2 Distances in the Streaming Model

An important thread of the literature on dynamic streaming algorithms for graph problems is concerned with computing *distances* and constructing *spanners* and *hopsets*. This is also the topic of the current part (part I) of this thesis. For a pair of parameters $\alpha \geq 1$, $\beta \geq 0$, given an undirected graph $G = (V,E)$, a subgraph $G' = (V,H)$ of G is said to be an *(α,β)-spanner* of G, if for every pair $u,v \in V$ of vertices, it holds that $d_{G'}(u,v) \leq \alpha \cdot d_G(u.v) + \beta$, where d_G and $d_{G'}$ are the distance functions of G and G', respectively. A spanner with $\beta = 0$ is called a *multiplicative* spanner and one with $\alpha = 1$ is called an *additive* spanner. There is another important variety of spanners called *near-additive* spanners for which $\beta \geq 0$ and $\alpha = 1+\varepsilon$, for an arbitrarily small $\varepsilon > 0$. The near-additive spanners are mostly applicable to *unweighted* graphs, even though there are some recent results about weighted near-additive spanners (20).

Spanners are very well-studied from both combinatorial and algorithmic viewpoints. It is well-known that for any parameter $\kappa = 1,2,\ldots,$ and for any n-vertex graph $G = (V,E)$, there exists a $(2\kappa-1)$-spanner with $O(n^{1+1/\kappa})$ edges, and this bound is nearly-tight unconditionally, and completely tight under Erdos-Simonovits girth conjecture (47; 8). The parameter $2\kappa-1$ is called the *stretch* parameter of the spanner. Also, for any pair of parameters, $\varepsilon > 0$ and $\kappa = 1,2,\ldots,$ there exists $\beta = \beta_{EP} = \beta(\kappa,\varepsilon)$, so that for every n-vertex undirected graph $G = (V,E)$, there exists a $(1+\varepsilon,\beta)$-spanner with $O_{\kappa,\varepsilon}(n^{1+1/\kappa})$ edges (26). The additive term $\beta = \beta_{EP}$ in (26) behaves as $\beta(\kappa,\varepsilon) \approx \left(\frac{\log\kappa}{\varepsilon}\right)^{\log\kappa}$, and this bound is the state-of-the-art. A lower bound of $\Omega(\frac{1}{\varepsilon\cdot\log\kappa})^{\log\kappa}$ for it was shown by Abboud et al. (5).

Given an n-vertex weighted undirected graph $G = (V,E,\omega)$ and two parameters $\varepsilon > 0$ and $\beta = 1,2,\ldots$, a graph $G' = (V,H,\omega')$ is called a $(1+\varepsilon,\beta)$-*hopset* of G, if for every pair of vertices $u,v \in V$, we have

$$d_G(u,v) \leq d^{(\beta)}_{G\cup G'}(u,v) \leq (1+\varepsilon)\cdot d_G(u,v) \tag{2.1}$$

Here $d^{(\beta)}_{G\cup G'}(u,v)$ stands for β-bounded distance (See Definition 2.5) between u and v in $G\cup G'$. (Note that for a weighted graph $G = (V,E,\omega)$, the weight of a non-edge $(u,v) \notin E$ is defined as $\omega((u,v)) = \infty$, and the weight of an edge (x,y) in the edge set of $G\cup G'$ is given by $\min\{\omega(x,y),\ \omega'(x,y)\}$.) The parameter β is called the *hopbound* of the hopset G'. We often refer to the edge set H of G' as the

hopset. Just like spanners, hopsets are a fundamental graph-algorithmic construct. They are extremely useful for computing approximate shortest distances and paths in various computational settings, in which computing shortest paths with a limited number of hops is significantly easier than computing them with no limitation on the number of hops. A partial list of these settings includes streaming, distributed, parallel and centralized dynamic models. Recently, hopsets were also shown to be useful for computing approximate shortest paths in the standard centralized model of computation as well (24).

Cohen (15) showed that for any undirected weighted n-vertex graph G, and parameters $\varepsilon > 0$, $\rho > 0$, and $\kappa = 1, 2, \ldots$, there exists a $(1+\varepsilon, \beta_C)$-hopset with $\tilde{O}(n^{1+1/\kappa})$ edges, where $\beta_c = \left(\frac{\log n}{\varepsilon}\right)^{O\left(\frac{\log\kappa}{\rho}\right)}$. Elkin and Neiman (23) improved Cohen's result, and constructed hopsets with *constant* hopbound. Specifically, they showed that for any $\varepsilon > 0$, $\kappa = 1, 2, \ldots$, and any n-vertex weighted undirected graph, there exists a $(1+\varepsilon, \beta_{EN})$-hopset with $\tilde{O}(n^{1+1/\kappa})$ edges, and $\beta_{EN} = \beta_{EP} \approx (\frac{\log\kappa}{\varepsilon})^{\log\kappa}$. The lower bound of Abbound et al. (5), $\beta = \Omega(\frac{1}{\varepsilon \cdot \log\kappa})^{\log\kappa}$ is applicable to hopsets as well. Generally, hopsets (see (15; 36; 23)) are closely related to near-additive spanners. See a recent survey (25) for an extensive discussion on this relationship.

Most of the algorithms for computing (approximate) distances and shortest paths in the streaming setting compute a sparse spanner, and then employ it for computing exact shortest paths and distances in it offline, i.e., in the post-processing, after the stream is over (30; 18; 11; 28; 22; 7; 40; 31; 32). Feigenbaum et al. (30) devised the first efficient *static* streaming algorithm for building multiplicative spanners. Their algorithm produces a $(2\kappa+1)$-spanner with $O(n^{1+1/\kappa}\kappa^2 \log^2 n)$ edges (and this is also the space complexity of the algorithm) in a single pass, and its processing time per edge is $\tilde{O}(n^{1/\kappa})$, for a parameter $\kappa = 1, 2, \ldots$. More efficient static streaming algorithms for this problem, that also provide spanners with a better stretch-size tradeoff, were devised in (18; 11). Specifically, these static streaming algorithms construct $(2\kappa-1)$-spanners of size $\tilde{O}(n^{1+1/\kappa})$ (and using this space), and as a result produce $(2\kappa-1)$-approximate all pairs shortest paths (henceforth, $(2\kappa-1)$-APASP) using space $\tilde{O}(n^{1+1/\kappa})$ in a single pass over the stream.

The algorithms of (30; 18; 11) apply to unweighted graphs, but they can be extended to weighted graphs by running many copies of them in parallel, one for each weight scale. Let $\Lambda = \Lambda(G)$ denote the *aspect ratio* of the graph, i.e., the ratio

between the maximum distance between some pair of vertices in G and the minimum distance between a pair of distinct vertices in G. Also, let $\varepsilon \geq 0$ be a slack parameter. Then by running $O(\frac{\log \Lambda}{\varepsilon})$ copies of the algorithm for unweighted graphs and taking the union of their outputs as the ultimate spanner, one obtains a one-pass static streaming algorithm for $2(1+\varepsilon)\kappa$-spanner with $\tilde{O}(n^{1+\frac{1}{\kappa}} \cdot (\log \Lambda)/\varepsilon)$ edges. See, for example, (27) for more details.

Elkin and Zhang (28) devised a static streaming algorithm for building $(1+\varepsilon, \beta_{EZ})$-spanners with $\tilde{O}(n^{1+1/\kappa})$ edges using β_{EZ} passes over the stream and space $\tilde{O}(n^{1+\rho})$, where $\beta_{EZ} = \beta_{EZ}(\varepsilon, \rho, \kappa) = \left(\frac{\log \kappa}{\varepsilon \cdot \rho}\right)^{O(\frac{\log \kappa}{\rho})}$, for any parameters $\varepsilon, \rho > 0$ and $\kappa = 1, 2, \ldots$. This result was improved in (22), where a static streaming algorithm with similar properties, but with $\beta = \beta_{EN} = \left(\frac{\log \kappa\rho + 1/\rho}{\varepsilon}\right)^{\log \kappa\rho + 1/\rho}$ was devised. The algorithms of (28; 22) directly give rise to β-pass static streaming algorithms with space $\tilde{O}(n^{1+\rho})$ for $(1+\varepsilon, \beta)$-APASP in unweighted graphs where $\beta(\rho) \approx (1/\rho)^{(1/\rho)(1+o(1))}$. They can also be used for producing purely multiplicative $(1+\varepsilon)$-approximate shortest paths and distances in $O(\beta/\varepsilon)$ passes and $\tilde{O}(n^{1+\rho})$ space from up to $n^{\rho(1-o(1))}$ designated sources to all other vertices.

There are also a number of additional *not* spanner-based static streaming algorithms for approximate shortest paths. Henzinger, Krinninger and Nanongkai (37) and Elkin and Neiman (23) devised $(1+\varepsilon)$-approximate *single*-source shortest paths (henceforth, SSSP) algorithms for weighted graphs, that are based on *hopsets*. The $(1+\varepsilon)$-SSSP algorithm of (36) employs $2^{O(\sqrt{\log n \log\log n})} = n^{o(1)}$ passes and space $n \cdot 2^{O(\sqrt{\log n \cdot \log\log n})} \cdot O(\frac{\log \Lambda}{\varepsilon}) = n^{1+o(1)} \cdot O(\frac{\log \Lambda}{\varepsilon})$. Elkin and Neiman (23) generalized and improved this result. For any parameters $\varepsilon, \rho > 0$, their static streaming algorithm computes $(1+\varepsilon)$-approximate SSSP using $\tilde{O}(n^{1+\rho})$ space and $\left(\frac{\log n}{\varepsilon \cdot \rho}\right)^{\frac{1}{\rho}(1+o(1))}$ passes. Moreover, in fact the same bound for number of passes and space applies in the algorithm of (23) for computing $S \times V$ $(1+\varepsilon)$-approximately shortest paths, for any subset $S \subseteq V$ of up to n^{ρ} designated sources. Yet more efficient static streaming algorithm for $(1+\varepsilon)$-approximate SSSP was devised by Becker et al. (12) using techniques from the field of continuous optimization. Their static streaming algorithm uses polylogarithmically many passes over the stream and space $O(n \cdot polylog(n))$. Finally, an *exact* static streaming SSSP algorithm was devised in (19). For any parameter $1 \leq p \leq n$, it requires $O(n/p)$ passes and $O(n \cdot p)$ space, and applies to weighted undirected graphs. The algorithm of (19)

also applies to the problem of computing $S \times V$ approximately shortest paths for $|S| \leq p$, and requires the same pass and space complexities as in the single-source case.

Recently Chang et al. (14) devised a *dynamic streaming* algorithm for this problem in *unweighted* graphs. Their algorithm uses $\tilde{O}(n/p)$ passes (for parameter $1 \leq p \leq n$ as above) and space $\tilde{O}(n+p^2)$ for the SSSP problem, and space $\tilde{O}(|S|n+p^2)$ for the $S \times V$ approximate shortest path computation. Ahn, Guha and McGregor (7) devised the first *dynamic streaming* algorithm for computing approximate distances. Their algorithm computes a $(2\kappa-1)$-spanner (for any $\kappa = 1,2,\ldots$) with $\tilde{O}(n^{1+1/\kappa})$ edges (and the same space complexity) in κ passes over the stream. This bound was recently improved by Fernandez, Woodruff and Yasuda (31). Their algorithm computes a spanner with the same properties using $\lfloor \kappa/2 \rfloor + 1$ passes. Ahn et al. (7) also devised an $O(\log \kappa)$-pass algorithm for building $O(\kappa^{\log_2 5})$-spanner with size and space complexity $\tilde{O}(n^{1+1/\kappa})$. This bound was recently improved by Filtser, Kapralov and Nouri (32), whose algorithm produces $O(\kappa^{\log_2 3})$-spanner with the same pass and space complexities, and the same size. Another dynamic streaming algorithm was devised by Kapralov and Woodruff (40). It produces a $(2^\kappa - 1)$-spanner with $\tilde{O}(n^{1+1/\kappa})$ edges (and space usage) in two passes. Filtser et al. (32) improved the stretch parameter of the spanner to $2^{\frac{\kappa+3}{2}} - 3$, with all other parameters the same as in the results of (40). Filtser et al. (32) also devised a general tradeoff in which the number of passes can be between 2 and κ, and the stretch of the spanner decreases gradually from exponential in κ (where the number of passes is 2) to $2\kappa-1$ (when the number of passes is κ). They have also devised a single pass algorithm with stretch $\tilde{O}(n^{\frac{2}{3}(1-1/\kappa)})$. As was mentioned above, all these spanner-based algorithms provide a solution for $(2\kappa-1)$-approximate all pairs almost shortest paths (henceforth, $(2\kappa-1)$-APASP) for unweighted graphs with space $\tilde{O}(n^{1+1/\kappa})$ and the number of passes equal to that of the spanner-construction algorithm. Like their static streaming counterparts (30; 18; 11), they can be extended to weighted graphs, at the price of increasing their stretch by a factor of $1+\varepsilon$ (for an arbitrarily small parameter $\varepsilon > 0$), and their space usage by a factor of $O\left(\frac{\log \Lambda}{\varepsilon}\right)$.

Table 2.1 provides a detailed overview of the prior works and their comparison to our results. To summarize, all known dynamic streaming algorithms for computing approximately shortest paths (with space $\tilde{O}(n^{1+1/\kappa})$, for a parameter $\kappa = 1,2,\ldots$),

Table 2.1: Prior Work on the Problem

Citation	Model	Stretch	Space	No. of Passes	Technique
(30)	Static	$(2\kappa+1)$	$\tilde{O}(n^{1+1/\kappa})$	1	Multiplicative Spanner
(18; 11)	Static	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	1	Multiplicative Spanner
(28; 22)	Static	$(1+\varepsilon,\beta_{EN})$	$\tilde{O}(n^{1+1/\kappa})$	β_{EN}	Near-Additive Spanner
(37)	Static	$(1+\varepsilon)$	$n^{1+o(1)}\cdot O(\frac{\log\Lambda}{\varepsilon})$	$2^{O(\sqrt{\log n\log\log n})}=n^{o(1)}$	Hopsets
(23)	Static	$(1+\varepsilon)$	$\tilde{O}(n^{1+\rho})$, for $\rho>0$	$\left(\frac{\log n}{\varepsilon\cdot\rho}\right)^{\frac{1}{\rho}(1+o(1))}$	Hopsets
(19)	Static	Exact	$O(n\cdot p)$, for $1\le p\le n$	$O(n/p)$	Hopsets
(7)	Dynamic	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	κ	Multiplicative Spanner
(31)	Dynamic	$(2\kappa-1)$	$\tilde{O}(n^{1+1/\kappa})$	$\lfloor\kappa/2\rfloor+1$	Multiplicative Spanner
(7)	Dynamic	$O(\kappa^{\log_2 5})$	$\tilde{O}(n^{1+1/\kappa})$	$O(\log\kappa)$	Multiplicative Spanner
(32)	Dynamic	$O(\kappa^{\log_2 3})$	$\tilde{O}(n^{1+1/\kappa})$	$O(\log\kappa)$	Multiplicative Spanner
(14)	Dynamic	Exact	$\tilde{O}(n+p^2)$, for $1\le p\le n$	$\tilde{O}(n/p)$	Exact BFS
(This Paper)	Dynamic	$(1+\varepsilon)$	$\tilde{O}(n^{1+1/\kappa})$	Constant (unweighted Graphs) $polylog(n)$ (weighted graphs)	Near-Additive Spanners and Hopsets

can be divided into two categories. The algorithms in the first category build a sparse multiplicative $(2\kappa-1)$-spanner, and they provide a *multiplicative* stretch of at least $2\kappa-1$ (7; 40; 31; 32). Moreover, due to existential lower bounds for spanners, this approach is doomed to provide stretch of at least $\frac{4}{3}\kappa$ (43). The algorithms in the second category compute *exact single source* shortest paths in *unweighted* graphs, but they employ $n^{1/2-O(1/\kappa)}$ passes (14; 19).

2.1.3 Our Results

We present the first dynamic streaming algorithm for SSSP with stretch $1+\varepsilon$, space $\tilde{O}(n^{1+1/\kappa})$, and *constant* (as long as ε and κ are constant) number of passes for unweighted graphs. For weighted graphs, our number of passes is *polylogarithmic* in n. Specifically, the number of passes of our SSSP algorithm is $(\frac{\kappa}{\varepsilon})^{\kappa(1+o(1))}$ for unweighted graphs, and $\left(\frac{(\log n)\kappa}{\varepsilon}\right)^{\kappa(1+o(1))}$ for weighted ones. Moreover, within the same complexity bounds, our algorithm can compute $(1+\varepsilon)$-approximate $S\times V$ shortest paths from $|S|=n^{1/\kappa}$ designated sources. Moreover, in *unweighted* graphs, *all* pairs almost shortest paths with stretch $(1+\varepsilon,\left(\frac{\kappa}{\varepsilon}\right)^{\kappa})$ can also be computed within the same space and number of passes. (That is, paths and distances with multiplicative stretch $1+\varepsilon$ and additive stretch $\left(\frac{\kappa}{\varepsilon}\right)^{\kappa}$.) Note that our multiplicative stretch $(1+\varepsilon)$ is dramatically better than $(2\kappa-1)$, exhibited by algorithms based on multiplicative spanners (7; 40; 31; 32), but this comes at a price of at least exponential increase in the number of passes. Nevertheless, our number of

passes is *independent of n*, for unweighted graphs, and depends only polylogarithmically on n for weighted ones.

2.1.4 Technical Overview

We devise two algorithms which build structures that help us compute approximate shortest paths. One of them builds a near-additive spanner (Chapter 3) and the other builds a near-exact hopset (Chapter 4). The following two theorems summarize the results of our spanner and hopset constructions.

Theorem 2.1. *(Theorem 3.12 in Section 3.1.3) For any unweighted graph $G(V,E)$ on n vertices, parameters $0<\varepsilon<1$, $\kappa\geq 2$, and $\rho>0$, our dynamic streaming algorithm computes a $(1+\varepsilon,\beta)$-spanner with $O_{\varepsilon,\kappa,\rho}(n^{1+1/\kappa})$ edges, in $O(\beta)$ passes using $O(n^{1+\rho}\log^4 n)$ space with high probability, where β is given by:*

$$\beta=\left(\frac{\log\kappa\rho+1/\rho}{\varepsilon}\right)^{\log\kappa\rho+1/\rho}.$$

Theorem 2.2. *(Theorem 4.20 in Section 4.2) For any n-vertex graph $G(V,E,\omega)$ with aspect ratio Λ, $2\leq\kappa\leq(\log n)/4$, $1/\kappa\leq\rho\leq 1/2$ and $0<\varepsilon'<1$, our dynamic streaming algorithm computes whp, a $(1+\varepsilon',\beta')$ hopset H with expected size $O(n^{1+1/\kappa}\cdot\log n)$ and the hopbound β' given by*

$$\beta'=O\left(\frac{(\log\kappa\rho+1/\rho)\log n}{\varepsilon'}\right)^{\log\kappa\rho+1/\rho}$$

It does so by making $O(\beta'\cdot(\log\kappa\rho+1/\rho))$ passes through the stream and using $O(n\cdot\log^3 n\cdot\log\Lambda)$ bits of space in the first pass and $O(\frac{\beta'^2}{\varepsilon'}\cdot\log^2 1/\varepsilon'\cdot n^{1+\rho}\cdot\log^5 n)$ bits of space in each of the subsequent passes.

The spanner is then used (in Section 3.2 of Chapter 3) to compute approximate shortest paths in unweighted graphs and the hopset is used (in Section 4.3 of Chapter 4) to compute approximate shortest paths in weighted graphs.

Our algorithms for spanner and hopset construction extend the results of (22; 23) from the static streaming setting to dynamic streaming one. The algorithms of (22; 23), like their predecessor, the algorithm of (26), are based on the superclustering-and-interconnection (henceforth, SAI) approach. Our algorithms in the current work also fall into this framework. Algorithms that follow the SAI approach proceed in phases, and in each phase they maintain a partial partition of the vertex

set V of the graph. Some of the clusters of G are selected to create superclusters around them. This is the superclustering step. Clusters that are not superclustered into these superclusters are then *interconnected* with their nearby clusters. The main challenge in implementing this scheme in the dynamic streaming setting is in the interconnection step. Indeed, the superclustering step requires a single and rather shallow BFS exploration (or a Bellman-Ford Exploration for weighted graphs), and implementing depth-d BFS in unweighted graphs in d passes over the dynamic stream can be done in near-linear space (See, e.g., (7; 14)). For the weighted graphs, we devise a routine for performing an approximate Bellman-Ford exploration up to a given hop-depth d, using d passes and $\tilde{O}(n)$ space.

On the other hand, the interconnection step requires implementing simultaneous BFS explorations (or Bellman-Ford Explorations for weighted graphs) originated at multiple sources. A crucial property that enabled (22; 23) to implement it in the static streaming setting is that one can argue that with high probability, not too many BFS explorations traverse any particular vertex. Let us denote by N, an upper bound on the number of explorations (traversing any particular vertex). In the dynamic streaming setting, however, at any point of the stream, there may well be much more than N explorations that traverse a specific vertex $v \in V$, based on the stream of updates observed so far. Storing data about all these explorations would make the space requirement of the algorithm prohibitively large.

To resolve this issue (and a number of related similar issues), we incorporate a *sparse recovery* routine into our algorithms. Sparse recovery is a fundamental and well-studied primitive in the dynamic streaming setting (33; 17; 38; 9). It is defined for an input which is a stream of (positive and negative) updates to an n-dimensional vector $\vec{a} = (a_1, a_2, \ldots, a_n)$. In the *strict* turnstile setting, which is sufficient for our application, ultimately each coordinate a_i (i.e., at the end of the stream) is non-negative, even though negative updates are allowed and intermediate values of coordinates may be negative. In the *general* turnstile model coordinates of the vector $\vec{a}$ may be negative at the end of the stream as well. The *support* of $\vec{a}$, denoted $supp(\vec{a})$, is defined as the set of its non-zero coordinates. For a parameter s, an *s-sparse recovery* routine returns the vector $\vec{a}$, if $|supp(\vec{a})| \leq s$, and returns failure otherwise. (It is typically also allowed to return failure with some small probability $\delta > 0$, given to the routine as a parameter, even if $|supp(\vec{a})| \leq s$.)

Most of sparse recovery routines are based on 1-sparse recovery, i.e., the case $s = 1$. The first 1-sparse recovery algorithm was devised by Ganguly (33), and it applies to the strict turnstile setting. The space requirement of the algorithm of (33) is $O(\log n)$. The result was later extended to the general turnstile setting by Cormode and Fermini (17) (See also, (45)). We devise an alternative streaming algorithm for this basic task in the strict turnstile setting. The space complexity of our algorithm is $O(\log n)$, like that of (33). The processing time-per-item of Ganguly's algorithm is however $O(1)$, instead of $polylog(n)$ of our algorithm. [1] Nevertheless, we believe that our new algorithm for this task is of independent interest. Chapter 5 is devoted to our new sparse recovery procedure, and its applications to ℓ_0-sampling. In Section 5.1, we describe this procedure, and in Section 5.2, we show how it can be used to build ℓ_0-samplers, (See Section 5.2 for their definition) with complexity that matches the state-of-the-art bounds for ℓ_0-samplers due to Jowhari, Sağlam and Tardos (39), but are arguably somewhat simpler.

For the greater part of the paper we analyze our algorithm in terms of the aspect ratio Λ of the input graph, given by $\Lambda = \frac{max_{u,v\in V} d_G(u,v)}{min_{u,v\in V d_G(u,v)}}$. (All dependencies are polylogarithmic in Λ.) In Section 4.2, however, we show that Klein-Subramania's weight reduction (42) (see also (23)) can be implemented in the dynamic streaming model. As a result, we replace all appearances of $\log\Lambda$ in the hopset's size, hopbound and number of passes of our construction by $O(\log n)$. However, the space complexity of our algorithm still mildly depends on $\log\Lambda$. Specifically, it is $\tilde{O}(n^{1+\rho}) + \tilde{O}(n)\cdot\log\Lambda$. In all existing dynamic streaming algorithms for computing multiplicative spanners or computing approximate shortest paths in weighted graphs (7; 40; 31; 32), both the spanner's size and the space requirements are $\tilde{O}(n^{1+1/\kappa}\cdot\log\Lambda)$. Completely eliminating the dependence on $\log\Lambda$ from these results is left as an open problem.

[1] If the algorithm knows in advance the dimension n of the vector $\vec{a}$ and is allowed to compute during preprocessing, before seeing the stream, a table of size n, then our algorithm can also have $O(1)$ processing time per update. This scenario occurs in dynamic streaming graph algorithms, including those discussed in the current paper.

2.1.5 Outline

The rest of the chapter is organized as follows. Section 2.2 provides necessary definitions and concepts that apply to Part I of this thesis (this chapter as well as Chapter 3, 4 and 5). Sections 2.3 and 2.4 provide the sub-routines required for our main algorithms presented in Chapters 3 and 4. Section 2.3 describes an algorithm for building a BFS forest of a given depth rooted at a subset of vertices of an unweighted input graph. Section 2.4 describes an algorithm for performing an approximate Bellman-Ford exploration rooted at a subset of vertices of a weighted input graph.

2.2 Preliminaries

2.2.1 Streaming Model

In the streaming model of computation, the set of vertices V of the input graph is known in advance and the edge set E is revealed one at a time. In an *insertion-only stream* the edges can only be inserted, and once inserted an edge remains in the graph forever. In a *dynamic stream*, on the other hand, the edges can be added as well as removed. We will consider unweighted graphs for our spanner construction algorithm and weighted graphs for our hopset construction algorithm. For an unweighted input graph, the stream S arrives as a sequence of edge updates $S = \langle s_1, s_2, \cdots \rangle$, where $s_t = (e_t, eSign_t)$, where e_t is the edge being updated. For a weighted input graph, the stream S arrives as a sequence of edge updates $S = \langle s_1, s_2, \cdots \rangle$, where $s_t = (e_t, eSign_t, eWeight_t)$, where e_t is the edge being updated and $eWeight_t$ is its weight. In unweighted as well weighted case, the $eSign_t \in \{+1, -1\}$ value of an update indicates whether the edge e_t is to be added or removed. A value of $+1$ indicates addition and a value of -1 indicates removal. There is no restriction on the order in which the $eSign$ value of a specific edge e changes. The multiplicity of an edge e is defined as $f_e = \sum_{t, e_t = e} eSign_t$. We assume that for every edge e, $f_e \in \{0, 1\}$ at that at the end of the stream. The order in which updates arrive may change from one pass of the stream to the other, while the final adjacency matrix of the graph at the end of every pass remains the same. We assume that the length of the stream or the number of updates we receive is

$poly(n)$. For more details on the streaming model of computation for graphs, we refer the reader to the survey (44) on graph streaming algorithms.

Definition 2.3. *For a vertex $v \in V$ and a vertex set $\mathscr{U} \subseteq V$, the* **degree of v with respect to $\mathscr{U}$** *is the number of edges connecting v to the vertices in $\mathscr{U}$.*

For a weighted undirected graph $G = (V, E, \omega)$, we assume that the edge weights are scaled so that the minimum edge weight is 1. Let $maxW$ denote the maximum edge weight $\omega(e)$, $e \in E$. For a non-edge $(u,v) \notin E$, we define $\omega((u,v)) = \infty$.

Denote also by Λ the *aspect ratio* of the graph, i.e., the maximum *finite* distance between some pair u, v of vertices (assuming that the minimum edge weight is 1).

Definition 2.4. *Given a weighted graph $G(V,E,\omega)$, a positive integer parameter t, and a pair $u, v \in V$ of distinct vertices, a* **t-bounded u-v *path*** *in G is a path between u and v that contains no more than t edges (also known as* hops*).*

Definition 2.5. *Given a weighted graph $G(V,E,\omega)$, a positive integer parameter t, and a pair $u, v \in V$ of distinct vertices, t-**bounded distance** between u and v in G denoted $d_G^{(t)}(u,v)$ is the length of the shortest t-bounded u-v path in G.*

2.2.2 Samplers

The main technical tool in our algorithms is a space-efficient sampling technique which enables us to sample a single vertex or a single edge from an appropriate subset of the vertex set or the edge set of the input graph, respectively. Most graph streaming algorithms use standard ℓ_0-sampler due to Jowhari et al. (39) as a black-box to sample edges or vertices from a graph. An ℓ_0-sampler enables one to sample almost uniformly from the support of a vector. We present an explicit construction of a sampling technique inspired by ideas from (41; 34; 16). Our construction is arguably simpler than the standard ℓ_0-sampler due to Jowhari et al. (39) and its space cost is at par with their sampler. In contrast to (39) which can handle positive as well as negative updates and final multiplicities (also referred to as *general turnstile stream*), our sampling technique works on streams with positive as well as negative updates provided the final multiplicity of each element is non-negative (also referred to as *strict turnstile stream*). This is a reasonable assumption for graph streaming algorithms.

For our spanner construction algorithm, we devise two samplers: *FindParent* and *FindNewVisitor* for unweighted graphs. For our hopset construction algorithm we devise two more samplers: *GuessDistance* and *FindNewCandidate*, which are essentially weighted graph counterparts of *FindParent* and *FindNewVisitror*, respectively. We will describe each of these samplers in detail in the sequel. The procedure *FindParent* works on unweighted graphs and enables us to find the parent of a given input vertex in a Breadth First Search (henceforth, BFS) forest rooted at a subset of the vertex set V of the input graph. The procedure *GuessDistance* works on weighted graphs and enables us to find the parent of a given vertex in a forest spanned by an *approximate* Bellman-Ford exploration. It also returns the approximate distance of the input vertex to the set of roots of the exploration. The procedure *FindNewVisitor* helps us to implement multiple simultaneous BFS traversals, each rooted at a different vertex in a subset S of the vertices of an unweighted input graph. The procedure *FindNewVisitor* enables us to sample, for a given $v \in V$, the root of one of the BFS explorations that v belongs to. The procedure *FindNewCandidate* is a counterpart of procedure *FindNewVisitor* Although our samplers *FindParent* and *FindNewVisitor* (and their counterparts for weighted graphs) are used in a specific context in our algorithm, they can be adapted to work in general to sample elements of any type from a dynamic stream with non-negative multiplicities. A variant of *FindParent* was described in (34; 41) in the context of dynamic and low-communication distributed graph algorithms. In the context of dynamic graph streams, we have adapted it to work as a sampler for sampling elements (in our case edges of a graph) whose multiplicity at the end of the stream is either 0 or 1. On the other hand, our second sampler, *FindNewVisitor* is more general and to the best of our knowledge, new. It can sample elements with non-negative multiplicities. As an example, *FindNewVisitor* can be adapted to sample edges from a multigraph in distributed, dynamic and dynamic streaming models.

The sampler *FindNewVisitor* (and also its weighted counterpart *FindNewCandidate*) is based on Jarnik's construction of convexly independent sets (50), and is related to constructions of lower bounds for distance preservers due to Coppersmith and Elkin (16).

2.2.3 Hash Functions

Algorithms for sampling from a dynamic stream are inherently randomized and often use hash functions as a source of randomness. A hash function h maps elements from a given input domain to an output domain of bounded size. Ideally, we would like to draw our hash function randomly from the space of all possible functions on the given input/output domain. However, since we are concerned about the space used by our algorithm, we will rely on hash functions with limited independence. A family of functions $H = \{h : \mathscr{U} \to [m]\}$, from a universe $\mathscr{U}$ to $[m]$, for some positive integer m, is said to be *k-wise independent*, if it holds that, when h is chosen uniformly at random from H then for any k distinct elements $x_1, x_2, \cdots, x_k \in \mathscr{U}$, and any k elements $z_1, z_2, \cdots, z_k \in [m]$, $x_1, x_2, \cdots, x_k$ and mapped by h to $z_1, z_2, \cdots, z_k$ with probability $1/m^k$, i.e., as if they were perfectly random. Such functions can be described more compactly, but are sufficiently random to allow formal guarantees to be proven.

The following lemma summarizes the space requirement of limited independence hash functions:

Lemma 2.6 ((13))**.** *A function drawn from a family of k-wise independent hash functions can be encoded in $O(k \log n)$ bits.*

Specifically, we will be using *pairwise independent* hash functions.

The following lemma, a variant of which has also been proved in (34; 41) in a different context, is proved here for the sake of completeness.

Lemma 2.7. *Let $h : \mathscr{U} \to [2^\lambda]$ be a hash function sampled uniformly at random from a family of pairwise independent hash functions $\mathscr{H}$. If we use h to hash elements of a given set $\mathscr{S} \subseteq \mathscr{U}$ such that $|\mathscr{S}| = s$, then a specific element $d \in \mathscr{S}$ hashes to the set $[2^{\lambda - \lceil \log s \rceil - 1}]$ and no other element of $\mathscr{S}$ does so with probability at least $\frac{1}{8s}$.*

Proof. Denote $t = \lambda - \lceil \log s \rceil - 1$. Let d^{Only} be the event that only the element $d \in \mathscr{S}$ and no other element $d' \in \mathscr{S}$ hashes to the set $[2^{\lambda - \lceil \log s \rceil - 1}] = [2^t]$. Note that $\frac{1}{4s} \leq \frac{2^t}{2^\lambda} \leq \frac{1}{2s}$. It follows that

$$\begin{aligned}
\Pr_{h\sim\mathscr{H}}[d^{Only}] &= \Pr_{h\sim\mathscr{H}}\left[h(d)\in[2^t] \bigwedge_{d'\in\mathscr{S}\setminus\{d\}} h(d')\notin[2^t]\right] \\
&= \Pr_{h\sim\mathscr{H}}\left[h(d)\in[2^t]\right]\cdot \Pr_{h\sim\mathscr{H}}\left[\bigwedge_{d'\in\mathscr{S}\setminus\{d\}} h(d')\notin[2^t] \mid h(d)\in[2^t]\right] \\
&\geq \Pr_{h\sim\mathscr{H}}\left[h(d)\in[2^t]\right]\cdot\left(1-\sum_{d'\in\mathscr{S}\setminus\{d\}}\Pr_{h\sim\mathscr{H}}\left[h(d')\in[2^t] \mid h(d)\in[2^t]\right]\right)
\end{aligned}$$

By pairwise independence,

$$\Pr_{h\sim\mathscr{H}}\left[h(d')\in[2^t] \mid h(d)\in[2^t]\right] = \Pr_{h\sim\mathscr{H}}\left[h(d')\in[2^t]\right]$$

$$\begin{aligned}
\text{Hence, } \Pr_{h\sim\mathscr{H}}[d^{Only}] &\geq \Pr_{h\sim\mathscr{H}}\left[h(d)\in[2^t]\right]\cdot\left(1-\sum_{d'\in\mathscr{S}\setminus\{d\}}\Pr_{h\sim\mathscr{H}}\left[h(d')\in[2^t]\right]\right) \\
&= \frac{2^t}{2^\lambda}\cdot\left(1-\sum_{d'\in\mathscr{S}\setminus\{d\}}\frac{2^t}{2^\lambda}\right) \geq \frac{1}{4s}\cdot\left(1-\sum_{d'\in\mathscr{S}\setminus\{d\}}\frac{1}{2s}\right) \\
&= \frac{1}{4s}\cdot\left(1-(s-1)\frac{1}{2s}\right) > \frac{1}{4s}\cdot\frac{1}{2} = \frac{1}{8s}
\end{aligned}$$

□

Lemma 2.7 implies the following corollary:

Corollary 2.8. *Let $h:\mathscr{U}\to[2^\lambda]$ be a hash function sampled uniformly at random from a family of pairwise independent hash functions $\mathscr{H}$. If we use h to hash elements of a given set $\mathscr{S}\subseteq\mathscr{U}$ with $|\mathscr{S}|=s$, then exactly one element in $\mathscr{S}$ hashes to the set $[2^t]$, $t=\lambda-\lceil\log s\rceil-1$, with probability at least $\frac{1}{8}$.*

Proof. Let *OneElement* be the event that exactly one of the s elements in the set $\mathscr{S}$ hashes to the set $[2^t]$. The event *OneElement* can be described as the event d^{Only} from Lemma 2.7 occurring for one of the elements $d\in\mathscr{S}$, i.e.,

$$\begin{aligned}
\Pr_{h\sim\mathscr{H}}[OneElement] &= \sum_{d\in\mathscr{S}}\Pr_{h\sim\mathscr{H}}\left[d^{Only}\right] \\
&\geq \sum_{d\in\mathscr{S}}\frac{1}{8s} = 1/8
\end{aligned}$$

□

2.2.4 Vertex Encodings

We assume that the vertices have unique IDs from the set $\{1,\ldots,n\}$. The maximum possible ID (which is n) of a vertex in the graph is denoted by $maxVID$. The binary representation of the ID of a vertex v can be obtained by performing a name operation $name(v)$.

We also need the following standard definitions of convex combination, convex hull and a convexly independent set.

Definition 2.9. *Given a finite number of vectors $x_1, x_2, \cdots, x_k$ in $\mathbb{R}^d$, a* **convex combination** *of these vectors is a vector of the form $\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_k x_k$, where the real numbers α_i satisfy $\alpha_i \geq 0$ and $\alpha_1 + \alpha_2 + \cdots \alpha_k = 1$.*

Definition 2.10. *The* **convex hull** *of a set $\mathscr{X}$ of vectors in $\mathbb{R}^d$ is the set of all convex combinations of elements of $\mathscr{X}$.*

Definition 2.11. *A set of vectors $x_1, x_2, \cdots, x_k \in \mathbb{R}^d$ is called a* **convexly independent set** *(*CIS *henceforth), if for every index $i \in [n]$, the vector x_i cannot be expressed as a convex combination of the vectors $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$.*

We will use the following *CIS*-based encoding for the vertices of the graph:
CIS Encoding Scheme ν: We assign a unique code in $\mathbb{Z}^2$ to every vertex $v \in V$. The encoding scheme works by generating a set of n convexly independent (See Definition 2.11) integer vectors in $\mathbb{Z}^2$. Specifically, our encoding scheme uses as its range, the extremal points of the convex hull (See Definition 2.10) of $Ball_2(R) \cap \mathbb{Z}^2$, where $Ball_2(R)$ is a two-dimensional disc of radius R centered at origin. A classical result by Jarník (50), later refined by Balog and Bárány (10), states that the number of extremal points of the convex hull of a set of integer points of a disc of radius R is $\Theta(R^{2/3})$. We set $R = \Theta(n^{3/2})$ to allow for all the possible $n = \Theta(R^{2/3})$ vertices to be encoded in $O(\log n)$ bits. The encoding of any vertex v can be obtained by performing an encoding operation denoted by $\nu(v)$.

The following lemma will be useful later in Section 3.1.2 and Section 4.1.2 to detect if the sampling procedure succeeded in sampling exactly one vertex from a desired subset of the set V.

Lemma 2.12. *Let $c_1, c_2, \cdots, c_n$ be non-negative integer coefficients of a linear combination of a set $P = \{p_1, p_2, \cdots p_n\}$ of n convexly independent points in $\mathbb{Z}^2$ such that $\frac{\sum_{j=1}^{n} c_j \cdot p_j}{\sum_{j=1}^{n} c_j} = p_i$, for some $p_i \in P$. Then $c_j = 0$ for every $j \neq i$.*

Proof. The expression $\frac{\sum_{j=1}^{n} c_j \cdot p_j}{\sum_{j=1}^{n} c_j}$ is a convex combination of points $p_1, p_2, \ldots, p_n$, since for every j, we have, $0 \leq \frac{c_j}{\sum_{j=1}^{n} c_j} \leq 1$ and $\sum_{j=1}^{n} \frac{c_j}{\sum_{j=1}^{n} c_j} = 1$. Since P is a CIS, by Definition 2.11, no point $p_i \in P$ can be represented as a convex combination of other points in P. Therefore, $c_j = 0$ for every $j \neq i$. □

2.3 BFS Forest

In this section, we describe an algorithm that generates a BFS forest rooted at a given set of source vertices of an input unweighted graph in dynamic streaming model.

2.3.1 General Outline

Given a graph $G(V, E)$, a set of source vertices $S \subseteq V$ and a depth parameter η, the algorithm outputs a set of edges $E_S^{\eta} \subseteq E$ of non-overlapping BFS explorations up to depth η, each rooted at a specific member of S. Initially, E_S^{η} is set to $\emptyset$. The algorithm proceeds in phases 1 to η, where for each $p \in [\eta]$, we discover the edges belonging to the layer p of the BFS forest in phase p. The layer p of the BFS forest is the set of vertices of G that are at distance p from S.

In each phase, we make one pass through the stream. Let $V_p \subseteq V$ denote the set of vertices belonging to the p^{th} layer of the forest. The set $V_p^{unc} = V \setminus \bigcup_{k \in [0,p]} V_k$ is the set of vertices that do not belong to any of the first p layers. The set V_0 is initialized to the set S and the set V_0^{unc} is set to $V \setminus V_0 = V \setminus S$.
Phase p starts by receiving as input, the sets V_{p-1} and V_{p-1}^{unc} computed in the previous phase. We invoke for each vertex $x \in V_{p-1}^{unc}$, a randomized procedure called *FindParent* to sample an edge (if exists) between x and some vertex $y \in V_{p-1}$.

The pseudocode for procedure *FindParent* is given in Algorithm 1. Its verbal description is provided right after that.

Algorithm 1 Pseudocode for Procedure *FindParent*

Procedure FindParent(x,h)

▷ Initialization

$slots \leftarrow \emptyset$ ▷ An array with λ elements indexed from 1 to λ, where $\lambda = \lceil \log n \rceil$.

▷ Each element of slots is a tuple $(xCount, xName)$. For a given index $1 \leq k \leq \lambda$, *xCount* and *xName* of $slots[k]$ can be accessed as $slots[k].xCount$ and $slots[k].xName$, respectively.

▷ $slots[k].xCount$ is number of sampled edges (x,y) with $h(y) \in [2^k]$. It is initialized as 0.

▷ $slots[k].xName$ is encoding of the (binary) names of the endpoints y of the sampled edges (x,y) with $h(y) \in [2^k]$. It is initialized as ϕ.

▷ Update Stage

while (there is some update $(e_t,\ eSign_t)$ in the stream) **do**
    if (e_t is incident on x and some $y \in V_{p-1}$) **then**
        $k \leftarrow \lceil \log h(y) \rceil$
        repeat
            $slots[k].xCount \leftarrow slots[k].xCount + eSign_t$
            $slots[k].xName \leftarrow slots[k].xName \bigoplus name(y)$
            $k = k + 1$
        until $k > \lambda$

▷ Recovery Stage

if (*slots* vector is empty) **then**
    return ϕ
else if ($\exists$ index $k \mid slots[k].xCount = 1$) **then**
    return $slots[k].xName$
else
    return $\perp$

The procedure *FindParent* takes as input the ID of a vertex and a hash function h chosen at random from a family of pairwise independent hash functions. A successful invocation of *FindParent* for an input vertex x in phase p returns an edge that connects x to some vertex in V_{p-1}, if there is at least one such edge in E, and ϕ

otherwise. Note that *FindParent* is a randomized procedure and may fail to sample an edge (with a constant probability) between x and V_{p-1}, even when such an edge exists. It returns an error $\perp$ in that case.

Before we start making calls to procedure *FindParent*, we sample uniformly at random a set of functions H_p from a family of pairwise independent hash functions $h : \{1,2,\ldots,maxVID\} \rightarrow \{1,\ldots,2^{\lambda}\}$, where $\lambda = \lceil \log maxVID \rceil = \lceil \log n \rceil$. Recall that $maxVID$ is the maximum possible vertex identity. The size of the set H_p will be specified later in the sequel. For every vertex $x \in V_{p-1}^{unc}$, we make $|H_p|$ parallel calls to procedure *FindParent*, one for each $h \in H_p$. As shown in the sequel, a single call to procedure *FindParent* succeeds only with a constant probability. Hence multiple parallel calls are required to boost the probability of successfully finding a parent for a given vertex. The set V_{p-1} computed in phase $p-1$ is made available in the global storage for all the calls to procedure *FindParent* in the phase p to access.

In the following section, we describe in detail the concepts used to implement the procedure *FindParent*.

2.3.2 Procedure FindParent

For a given vertex $x \in V_{p-1}^{unc}$, let $d_x^{(p-1)}$ be the degree of x with respect to set V_{p-1}. In what follows, we will refer to an edge between x and some $y \in V_{p-1}$ as a *candidate edge*. A simple randomized technique to find a parent for x is by sampling its incident edges that connect it to the set V_{p-1} with probability $\frac{1}{d_x^{(p-1)}}$ (by flipping a biased coin) and keeping track of all the updates to the sampled edges. A given edge can appear or disappear multiple times in the stream and one needs to remember the random bit for every candidate edge (the result of coin flip for the edge when it appeared for the first time). Remembering random bits is required in order to treat every update to a given candidate edge consistently as the stream progresses. This requires remembering $O(n)$ bits per vertex. Instead, we use a pairwise independent hash function to assign hash values to the candidate edges in the range $\{1,2,\ldots,2^{\lambda}\}$, where $\lambda = \lceil \log maxVID \rceil$. If we knew the exact value of $d_x^{(p-1)}$, we could sample every new candidate edge witnessed by x with probability $1/d_x^{(p-1)}$ to extract exactly one of them in expectation. However, all we know about $d_x^{(p-1)}$ is that it is at most n. We therefore sample every new candidate edge on a range of

probabilities. We use an array *slots* of λ elements (the structure of each element will be described later in the sequel) indexed by *slot-levels* from 1 to $\lambda = \lceil \log n \rceil$ to implement sampling on a range of probabilities. We want a given candidate edge (x,y) to be sampled into slot-level k with probability $1/2^{\lambda-k}$. When $d_x^{(p-1)} \approx 2^{\lambda-k}$, with a constant probability there is exactly one candidate edge that gets mapped to $slots[k]$. Every new candidate edge $e = (x,y)$ witnessed by x with $y \in V_{p-1}$ is assigned a hash value $h(y)$ by h. A given edge $e = (x,y)$ gets mapped into $slots[k]$, if $h(y) \in [2^k]$. Note that a given candidate edge may be assigned to multiple slot-levels. In every element of *slots*, we maintain a tuple $(xCount, xName)$, and *xCount* and *xName* of $slots[k]$ can be accessed as $slots[k].xCount$ and $slots[k].xName$, respectively.

The field $xCount \in \mathbb{Z}$ at slot-level k maintains the number of candidate edges with hash values in $[2^k]$. It is initialized to 0 at the start of the stream. Every time an update to a candidate edge $e = (x,y)$ with $h(y) \in [2^k]$ appears on the stream, $slots[k].xCount$ is updated by adding the *eSign* value of e to its current value. The final value of the *xCount* field is thus given by the following expression:

$$slots[k].xCount = \sum_{(e_t,\ eSign_t) | e_t = (x,y) \text{ for some } y \in V_{p-1} \text{ and } h(y) \in [2^k]} eSign_t$$

The field *xName* at slot-level k is a bit string which maintains the bitwise XOR of the *binary* names of all the candidate edges sampled at slot-level k. It is initalized as an empty string at the start of the stream. Every time an update to a candidate edge $e = (x,y)$ with $h(y) \in [2^k]$, $y \in V_{p-1}$, appears on the stream, $slots[k].xName$ is updated by performing a bitwise XOR of its current value with $name(y)$. The final value of the *xName* field is thus given by the following expression:

$$slots[k].xName = \bigoplus_{(e_t,\ eSign_t) | e_t = (x,y) \text{ for some } y \in V_{p-1} \text{ and } h(y) \in [2^k]} name(y)$$

At the end of the stream, if the *slots* array is empty, then there are no edges incident on x that connect it to the set V_{p-1} and the *FindParent* procedure returns ϕ. (Note that $slots[\lambda]$ is an encoding of all the candidate edges incident on x.) If there is a slot-level k such that $slots[k].xCount = 1$, then only one candidate edge is mapped to slot-level k and $slots[k].xName$ gives us the name of the other endpoint of this edge. The procedure *FindParent* returns $slots[k].xName$ as a parent of x. If the *slots*

array is not empty but there is no slot level with its $xCount = 1$, then the procedure *FindParent* has failed to find a parent for x and returns an error $\perp$.

If the input vertex x has a non-zero degree with respect to the set V_{p-1}, we need to make sure that for some $1 \leq k \leq \lambda$, only one candidate edge will get mapped to $slots[k]$. By Corollary 2.8, only one of the $d_x^{(p-1)}$ candidate edge gets mapped to the set $[2^k]$, for $k = \lambda - \lceil \log d_x^{(p-1)} \rceil - 1$, with at least a constant probability. Therefore, a single invocation of *FindParent* succeeds with at least a constant probability. Since we are running $|H_p|$ parallel invocations of *FindParent*, we pick the output of a successful invocation of procedure *FindParent* as the parent. (See Section 2.3.1; H_p is a set of randomly sampled hash functions.) If multiple invocations are successful, we use the output of one of them arbitrarily. In the case that all the invocations of *FindParent* return an error, the algorithm terminates with an error. In the sequel we show that when the set H_p is appropriately sized, the event of all the invocations of procedure *FindParent* for a given vertex failing has very low probability.

At the end of phase p, if the algorithm has not terminated with an error, every vertex $x \in V_{p-1}^{unc}$ for which we have sampled an edge to the set V_{p-1}, is added to the set V_p. Every sampled edge is added to the set E_S^{η}. The set V_p^{unc} is updated as $V_p^{unc} = V_{p-1}^{unc} \setminus V_p$.

Lemma 2.13. *For $|H_p| = c_1 \log_{8/7} n$ for some $c_1 \geq 1$, at least one of the $|H_p|$ invocations of procedure* FindParent *for a given vertex in phase p succeeds with probability at least $1 - \frac{1}{n^{c_1}}$.*

Proof. The procedure *FindParent* relies on the ability of the random pairwise hash function to hash exactly one edge in the target range of $[2^{\lambda - \left\lceil \log d_x^{(p-1)} \right\rceil - 1}]$. By Corollary 2.8, this happens with at least a constant probability of $1/8$. If we invoke procedure *FindParent* $c_1 \log_{8/7} n$ times in parallel using independently chosen at random hash functions, then all of them fail with a probability at most $(7/8)^{c_1 \log_{8/7} n} = \frac{1}{n^{c_1}}$. Therefore, at least one of the $|H_p|$ invocations succeeds with probability at least $1 - \frac{1}{n^{c_1}}$. □

Next, we analyze the space requirements of procedure *FindParent*.

Lemma 2.14. *The procedure* FindParent *uses $O(\log^2 n)$ bits of memory.*

Proof. The input to this procedure is the ID of a vertex x and a pairwise independent hash function h. This consumes $O(\log n)$ bits. The procedure also needs access to the set of vertices V_{p-1} of the previous layer. We will not charge this procedure for the space required for storing V_{p-1}, since it is output by the phase $p-1$ and is passed on to phase p as an input. We instead charge phase $p-1$ globally for its storage. Similarly, we do not charge each invocation of *FindParent* in phase p for the storage of the hash function h. Rather it is charged to phase p globally. Inside the procedure, the *slots* vector is an array of length λ and $\lambda = O(\log n)$. Every element of *slots* stores two variables *xCount* and *xName* each of which consumes $O(\log n)$ bits. Thus the overall space required by this procedure is $O(\log^2 n)$ bits. □

We now proceed to analyzing the space requirements of the entire algorithm.

Lemma 2.15. *In each of the η phases, our BFS forest construction algorithm uses $O(n\log^3 n)$ memory.*

Proof. In any phase $p \geq 1$, we try to find a parent for every vertex in the set V_{p-1}^{unc}. This requires making multiple simultaneous calls to procedure *FindParent*. By Lemma 2.13, we need to make $O(\log n)$ parallel calls to procedure *FindParent* per vertex. For this we sample $O(\log n)$ pairwise independent hash functions. Every single pairwise independent hash function requires $O(\log n)$ bits of storage (Lemma 2.6) and thus the set H_p requires $O(\log^2 n)$ bits of storage. By Lemma 2.14, a single call to procedure *FindParent* uses $O(\log^2 n)$ bits. Thus making $O(\log n)$ parallel calls (by Lemma 2.13) needs $O(\log^3 n)$ bits per vertex. The set V_{p-1}^{unc} has size $O(n)$. Thus the overall cost of all the calls to procedure *FindParent* is $O(n\log^3 n)$. As an output, phase p generates the set V_p and the set of edges belonging to the layer p of the BFS which is then added to the final output set E_S^{η}. Both these sets are of size $O(n)$ and each element of these sets requires $O(\log n)$ bits. Thus the cost of maintaining the output of phase p is bounded by $O(n\log n)$ bits. Hence the overall storage cost of phase p is dominated by the calls to procedure *FindParent*. The overall storage cost of any phase is therefore $O(n\log^3 n)$ bits. □

In the following lemma, we provide an inductive proof of the correctness of our algorithm. Recall that $|H_p| = c_1 \log_{8/7} n$, where, $c_1 > 0$ is a positive constant.

Lemma 2.16. *After p phases of the algorithm described in Section 2.3.1, the algorithm has constructed a BFS forest to depth p rooted at $S \subseteq V$ with probability at least $1 - p/n^{c_1-1}$.*

Proof. The proof follows by induction on the number of phases, p, of the algorithm. The base case for $p = 0$ holds trivially. For the inductive step, we assume that after k phases of our algorithm, the set of output edges E_S^η forms a BFS forest to depth k with probability at least $1 - k/n^{c_1-1}$. This implies that all the vertices within distance k from S have found a parent in the BFS forest with probability at least $1 - k/n^{c_1-1}$. In phase $k+1$, we make $|H_{k+1}|$ parallel calls to procedure *FindParent* for every vertex not yet in the forest. For all the vertices at a distance more than $k+1$ from the set S, all the calls to procedure *FindParent* return ϕ in phase $k+1$. Let x be a vertex at distance $k+1$ from the set S. By Lemma 2.13, at least one of the $|H_{k+1}|$ independent calls to procedure *FindParent* made for x in phase $k+1$ succeeds in finding a parent for x with probability at least $1 - \frac{1}{n^{c_1}}$. Since there can be at most $O(n)$ vertices at distance $k+1$ from set S, by union bound, phase $k+1$ fails to find a parent for one of these vertices with probability at most $1/n^{c_1-1}$. Taking a union bound over the failure probability of first k phases from induction hypothesis with the failure probability of phase $k+1$, we get that all the vertices within distance $k+1$ from the set S successfully add their parent edges in the BFS forest to the output set E_S^η with probability at least $1 - (k+1)/n^{c_1-1}$. □

Lemmas 2.15 and 2.16 imply the following theorem:

Theorem 2.17. *For a sufficiently large positive constant c, given a depth parameter η, an input graph $G(V,E)$, and a subset $S \subseteq V$, the algorithm described in Section 2.3.1 generates with probability at least $1 - \frac{1}{n^c}$, a BFS forest of G of depth η rooted at vertices in the set S in η passes through the dynamic stream using $O_c(n\log^3 n)$ space in every pass.*

Note also that the space used by the algorithm on different passes can be reused, i.e., the total space used by the algorithm is $O_c(n\log^3 n)$.

2.4 Approximate Bellman-Ford Explorations

In this section, we describe an algorithm for performing a given number of iterations of an approximate Bellman-Ford exploration from a given subset $S \subseteq V$ of *source* vertices in a weighted undirected graph $G(V,E,\omega)$ with aspect ratio Λ. We assume throughout that the edge weights are positive numbers between 1 and $maxW$. Note that $\Lambda \leq (n-1) \cdot maxW$. Recall that for a pair $u,v \in V$ of distinct vertices and an integer $t \geq 0$, the t-bounded distance between u and v in G, denoted $d_G^{(t)}(u,v)$, is the length of a shortest t-bounded u-v path in G. (See Definitions 2.4 and 2.5.) For a given vertex $v \in V$ and a set $S \subseteq V$, the t-bounded distance between v and S in G, denoted $d_G^{(t)}(v,S)$, is the length of a shortest t-bounded path between v and some $s \in S$ such that $d_G^{(t)}(v,s) = \min\{d_G^{(t)}(s',v) \mid s' \in S\}$.

2.4.1 General Outline

Given an n-vertex weighted graph $G(V,E,\omega)$, a set $S \subseteq V$ of vertices, an integer parameter $\eta > 0$ and an error parameter $\zeta \geq 0$, an (η,ζ)-Bellman-Ford exploration (henceforth, BFE) of G rooted at S outputs for every vertex $v \in V$, a $(1+\zeta)$-approximation of its η-bounded distance to to the set S. Throughout the execution of our algorithm, we maintain two variables for each vertex $v \in V$. One of them is a current estimate of v's η-bounded distance to set S, denoted $\hat{d}(v)$, and the other is the ID of v's neighbour through which it gets its current estimate, denoted $\hat{p}(v)$, and called the *parent* of v.

We start by initializing $\hat{d}(s) = 0$, $\hat{p}(s) = \perp$, for each $s \in S$ and $\hat{d}(v) = \infty$, $\hat{d}(v) = \perp$ for each $v \in V \setminus S$. As the algorithm proceeds, $\hat{d}(v)$ and $\hat{p}(v)$ values of every vertex $v \in V \setminus S$ are updated to reflect the current best estimate of v's η-bounded distance to the set S. The final value of $\hat{d}(v)$ for each $v \in V$ is such that $d_G^{(t)}(v,S) \leq \hat{d}(v) \leq (1+\zeta) \cdot d_G^{(t)}(v,S)$, and the final value of $\hat{p}(v)$ for each $v \in V$ contains the ID of v's parent on the forest spanned by (η,ζ)-BFE of G rooted at the set S.

The algorithm proceeds in phases, indexed by p, $1 \leq p \leq \eta$. We make one pass through the stream in each phase.

Phase p: In every phase, we search for every vertex $v \in V \setminus S$, a *better* (smaller than the current value of $\hat{d}(v)$) estimate (if exists) of its η-bounded distance to the

set S, by keeping track of updates to edges $e = (v,u)$ incident to v. Specifically, we divide the search space of potential better estimates, $[1, 2 \cdot \Lambda]$, into sub-ranges $I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$, for $j \in \{0,1,\ldots,\gamma\}$, where $\gamma = \lceil \log_{1+\zeta'} 2 \cdot \Lambda \rceil - 1$ and ζ' is set to $\zeta/2\eta$ for technical reasons to be expounded later in the sequel. For $j = 0$, we make the sub-range $I_0 = \left[(1+\zeta')^0, (1+\zeta')^1\right]$ closed to include the value 1. Recall that we are doing a $(1+\zeta)$-approximate Bellman-Ford exploration (and not an exact one). Due to this, some of the better estimates we get in a given phase may be between Λ and $(1+\zeta) \cdot \Lambda \leq 2 \cdot \Lambda$, where Λ is the aspect ratio of the input graph. We therefore keep our search space from 1 to 2Λ instead of Λ.

In more detail, we make for for each $v \in V \setminus S$, γ *guesses*, one for each sub-range. In a specific guess for a vertex v corresponding to sub-range $\left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$ for some j, we make multiple simultaneous calls to a randomized procedure called *GuessDistance* which samples an edge (if exists) between v and some vertex u such that

$$\hat{d}(u) + \omega(v,u) \in I_j.$$

The exact number of calls we make to procedure *GuessDistance* in each guess will be specified later in the sequel.

The smallest index $j \in [0,\gamma]$, for which the corresponding guess denoted $Guess_v^{(j)}$ successfully samples an edge which gives a distance estimate better than the current estimate of v, is chosen to update $\hat{d}(v)$.

The pseudocode for procedure *GuessDistance* is given in Algorithm 2. Its verbal description is provided right after that.

The procedure *GuessDistance* can be viewed as an adaptation of procedure *Find-Parent* from Section 2.3.2 for weighted graphs. It enables us to find an estimate of η-bounded distance of an input vertex x to the set S in a given range of distances. It takes as input the ID of a vertex, a hash function h chosen at random from a family of pairwise independent hash functions and an input range $I = (low, high]$. (The input range may be closed as well.) A successful invocation of procedure *Guess-Distance* for an input vertex x and input range I, returns a tuple $(dist, parent)$, (if there is at least one edge (x,y) in G such that $\hat{d}(y) + \omega(x,y) \in I$, and ϕ otherwise), where $dist$ is an estimate of x's η-bounded distance to the set S in the range I, and $parent$ is the $parent$ of x in the forest spanned by (η,ζ)-BFE of G rooted at set S.

Algorithm 2 Pseudocode for Procedure *GuessDistance*

1: **Procedure GuessDistance**(x,h,I)

▷ Initialization

2: $slots \leftarrow \emptyset$ ▷ An array with λ elements indexed from 1 to λ, where $\lambda = \lceil \log n \rceil$.
▷ Each element of slots is a tuple $(xCount, xDist, xName)$. For a given index $1 \leq k \leq \lambda$, fields *xCount*, *xDist* and *xName* of $slots[k]$ can be accessed as $slots[k].xCount$, $slots[k].xDist$ and $slots[k].xName$, respectively.

3:
▷ $slots[k].xCount$ is the number of sampled edges (x,y) with $h(y) \in [2^k]$. Initially, it is set to 0.
▷ $slots[k].xDist$ is the distance estimate for x provided by an edge (x,y) with $h(y) \in [2^k]$. initially, it is set to 0.
▷ $slots[k].xName$ is encoding of the names of the endpoints y of sampled edges (x,y) with $h(y) \in [2^k]$. Initially, it is set to ϕ.

▷ Update Stage

4: **while** (there is some update $(e_t,\ eSign_t,\ eWeight_t)$ in the stream) **do**
5: **if** (e_t is incident on x and some y such that $\hat{d}(y) + eWeight_t \in I$) **then**
6: $k \leftarrow \lceil \log h(y) \rceil$
7: **repeat**
8: $slots[k].xCount \leftarrow slots[k].xCount + eSign_t$
9: $slots[k].xDist \leftarrow slots[k].xDist + (\hat{d}(y) + eWeight_t) \cdot eSign_t$
10: $slots[k].xName \leftarrow slots[k].xName \bigoplus name(y)$
11: $k = k + 1$
12: **until** $k > \lambda$

▷ Recovery Stage

13: **if** (*slots* array is empty) **then**
14: return (ϕ, ϕ)
15: **else if** ($\exists$ index $k \mid slots[k].xCount = 1$) **then**
16: return $(slots[k].xDist, slots[k].xName)$
17: **else**
18: return $(\perp, \perp)$

The procedure *GuessDistance* may fail to return (with a constant probability) a distance estimate in the desired range, even when such an estimate exists. It returns an error, denoted by $(\perp, \perp)$, in that case.

As we did for procedure *FindParent* in Section 2.3, before we start making calls to procedure *GuessDistance*, we sample uniformly at random a set of functions H_p of size $c_1 \log_{8/7} n$ from a family of pairwise independent hash functions $h : \{1,\ldots,maxVID\} \rightarrow \{1,\ldots,2^{\lambda}\}$, where $\lambda = \lceil \log n \rceil$ and c_1 is an appropriate constant. For every guess for a given vertex $x \in V \setminus S$ and a given subrange I_j, we make $|H_p|$ parallel calls to procedure *GuessDistance*, one for each $h \in H_p$, to get an estimate of $d_G^{(\eta)}(x,S)$ in the given subrange. The multiple parallel calls are required since a single call to procedure *GuessDistance* succeeds only with a constant probability, while we need to succeed with high probability.

Additionally, before we start the phase p, we create for each $v \in V \setminus S$, a copy $\hat{d}'(v)$ of its current distance estimate $\hat{d}(v)$. Any update to the distance estimate of a vertex v during phase p is made to its *shadow* distance estimate $\hat{d}'(v)$. On the other hand, the variable $\hat{d}(v)$ for vertex $v \in V \setminus S$ remains unchanged during the execution of phase p. At the end of phase p, we update $\hat{d}(v)$ as $\hat{d}(v) = \hat{d}'(v)$. The purpose of using the shadow variable is to avoid any issues arising due to simultaneous reading from and writing to the distance estimate variable of a vertex by multiple parallel calls to procedure *GuessDistance*.

2.4.2 Procedure GuessDistance

The overall structure and technique of procedure *GuessDistance* is similar to that of procedure *FindParent*. (See Section 2.3.2.) For a given vertex x, and a given distance range I, let $y \in \Gamma_G(x)$ be such that

$$\hat{d}(y) + \omega(x,y) \in I \tag{2.2}$$

In what follows, we will refer to a vertex $y \in \Gamma_G(x)$ for which Equation 2.2 holds as a *candidate neighbour* and the corresponding edge (x,y) as a *candidate edge* in the range I. For a given vertex x, let $c_x^{(p,j)}$ be the number of candidate neighbours of x in the sub-range I_j. A call to procedure *GuessDistance* for vertex x with input range $I = I_j$ works by sampling a *candidate* neighbour with probability $\frac{1}{c_x^{(p,j)}}$. As described in Section 2.3.2, one of the ways to sample with a given probability in a dynamic streaming setting is to use hash functions. We therefore use a pairwise independent hash function as in Section 2.3.2 to assign hash values to the candidate edges in the range $\{1,\ldots,2^{\lambda}\}$, where $\lambda = \lceil \log n \rceil$. As in the case of *FindParent*,

we only know an upper bound of n and not the exact value of $c_x^{(p,j)}$. Therefore, we try to guess $c_x^{(p,j)}$ on a geometric scale of values $2^{\lambda-k}$, $k = 1,2,\ldots,\lambda$, and sample every candidate neighbour on a range of probabilities corresponding to our guesses of $c_x^{(p,j)}$. To implement sampling on a range of probabilities, we use an array *slots* of λ elements indexed by *slot-levels* from 1 to λ. Every new candidate neighbour y witnessed by x is assigned a hash value $h(y)$ by h.

In every element of *slots*, we maintain a tuple $(xCount, xDist, xName)$, and *xCount*, *xDist* and *xName* of $slots[k]$ can be accessed as $slots[k].xCount$, $slots[k].xDist$ and $slots[k].xName$, respectively.
The variable $xCount \in \mathbb{Z}$ at slot-level k maintains the number of candidate neighbours with hash values in $[2^k]$. It is initialized to 0 at the beginning of the stream. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^k]$ appears on the stream, $slots[k].xCount$ is updated by adding the $eSign_t$ value of e_t to its current value. The variable *xDist* at slot-level k is an estimate of η-bounded distance of x limited to the input distance range I provided by edge (x,y) with $h(y) \in [2^k]$. Initially, it is set to 0. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^k]$ appears on the stream, $slots[k].xDist$ is updated by adding the value of the expression $(\hat{d}(y) + eWeight_t) \cdot eSign_t$ to its current value. (Recall that it is initialized as 0.) The variable *xName* is encoding of the names of endpoints y of the sampled edges (x,y) with $h(y) \in [2^k]$. It is set to ϕ initially. Every time an update to a candidate edge $e_t = (x,y)$ with $h(y) \in [2^k]$ appears on the stream, $slots[k].xName$ is updated by performing a bitwise XOR of its current value with $name(y)$.

At the end of the stream, if the *slots* array is empty, then there are no candidate neighbours in $\Gamma_G(x)$ and the procedure *GuessDistance* returns (ϕ,ϕ) (see lines 13 and 14 of Algprothm 2.). If there is a slot-level k such that $slots[k].xCount = 1$ (Line 15 of Algorithm 2), then only one candidate neighbour is mapped to slot-level k. In this case, $slots[k].xDist$ gives us an estimate of x's η-bounded distance to the set S in the input distance range I, and $slots[k].xName$ gives us the name of x's parent on the forest spanned by the (η,ζ)-BFE of G rooted at set S. Indeed, if no smaller scale estimate will be discovered, the vertex recorded in $slots[k].xName$ will become the parent of x in the forest. The procedure *GuessDistance* returns $(slots[k].xDist, slots[k].xName)$. If the *slots* vector is not empty but there is no

slot level with $xCount = 1$, then the procedure *GuessDistance* has failed to find a distance estimate in the input range I for x, and thus it returns an error $(\perp, \perp)$.

If the input vertex x has some candidate neighbours in the input distance range, we need to make sure that for some $1 \leq k \leq \lambda$, only one candidate neighbour will get mapped to $slots[k]$. By Corollary 2.8, only one of the $c_x^{(p,j)}$ candidate neighbours gets mapped to the set $[2^k]$, for $k = \lambda - \lceil \log c_x^{(p,j)} \rceil - 1$, with at least a constant probability. Therefore, a single invocation of procedure *GuessDistance* for a given vertex x and a given distance range succeeds with at least a constant probability. Since we are running $|H_p|$ parallel invocations of procedure *GuessDistance* for a given input vertex x and a given distance range I, we pick the output of a successful invocation of procedure *GuessDistance* as an estimate for x in the input range. If multiple invocations in a guess are successful, we use the output of the one with the smallest return value. In the case that all the invocations of *GuessDistance* in a guess return an error, the algorithm terminates with an error. In the sequel we show that when the set H_p is appropriately sized, the event of all the invocations of procedure *GuessDistance* in a given guess failing has a very low probability.

Once all the $\gamma = O(\frac{\log \Lambda}{\zeta'})$ guesses for a given vertex x have completed their execution without failure, we pick the smallest index j for which the corresponding guess $guess_x^{(j)}$ has returned a finite (non-failure) value, and compare this value with $\hat{d}(x)$. If this value gives a better estimate than the current value of $\hat{d}(x)$, we update the corresponding shadow variable $\hat{d}'(x)$, and the parent variable $\hat{p}(x)$. At the end of phase p, if the algorithm has not terminated with an error, for every vertex $x \in V \setminus S$, we update its current distance estimate variable with the value in the corresponding shadow variable as $\hat{d}(x) = \hat{d}'(x)$.

In the following lemma, we analyze the success probability of guessing the η-bounded distance of a specific vertex in a given distance range in phase p.

Lemma 2.18. *For $|H_p| = c_1 \log_{8/7} n$ for some $c_1 \geq 1$, at least one of the $|H_p|$ invocations of procedure* GuessDistance *in a given guess for a vertex x, and distance sub-range $I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$ for some j, in a specific phase p succeeds with probability at least $1 - \frac{1}{n^{c_1}}$.*

Proof. The procedure *GuessDistance* relies on the ability of the random pairwise independent hash function to hash exactly one edge in the target range of

$[2^{\lambda-\left\lceil \log c_x^{(p,j)} \right\rceil - 1}]$. By Corollary 2.8, this happens with at least a constant probability of $1/8$. If we invoke procedure *GuessDistance* $c_1 \log_{8/7} n$ times in parallel using independently chosen at random hash functions, then all of them fail with a probability at most $(7/8)^{c_1 \log_{8/7} n} = \frac{1}{n^{c_1}}$. Therefore, at least one of the $|H_p|$ invocations succeeds with probability at least $1 - \frac{1}{n^{c_1}}$. □

Next, we analyze the space requirements of procedure *GuessDistance*.

Lemma 2.19. *The procedure* GuessDistance *uses* $O(\log n(\log n + \log \Lambda))$ *bits of memory.*

Proof. The input to this procedure is the ID of a vertex x, a pairwise independent hash function h and variables *low* and *high*, that define the input range I. The ID of the vertex and the representation of the hash function h consume $O(\log n)$ bits. The variables *low* and *high* correspond to distances in the input graph and are upper bounded by the aspect ratio Λ of the graph. Therefore both these variables consume $O(\log \Lambda)$ bits each. We do not charge each invocation of *GuessDistance* in phase p for the storage of the hash function h. Rather it is charged to phase p globally. Inside the procedure, the *slots* vector is an array of length λ and $\lambda = O(\log n)$. Every element of *slots* stores three variables *xCount*, *xDist* and *xName*. The variables *xCount* and *xName* consume $O(\log n)$ bits. The variable *xDist* is a distance estimate and thus consumes $O(\log \Lambda)$ bits. Thus the overall space required by this procedure is $O(\log n(\log n + \log \Lambda))$ bits. □

We now proceed to analyzing the space requirements of the entire algorithm.

Lemma 2.20. *In each of the* η *phases, our approximate Bellman-Ford exploration algorithm uses* $O(n \cdot \log^2 n \frac{\log \Lambda}{\zeta'}(\log n + \log \Lambda))$ *bits of memory.*

Proof. In any phase $p \geq 1$, we search for a possible better estimate (if exists) of $d_G^{\eta}(v,S)$ for every vertex $v \in V \setminus S$. This requires making $\gamma = \lceil \log_{(1+\zeta')} 2 \cdot \Lambda \rceil - 1$ guesses. Each guess in turn makes $|H_p| = c_1 \log_{8/7} n$ simultaneous calls to procedure *GuessDistance*. Therefore, in total, we make $O(\log_{1+\zeta'} \Lambda \cdot \log_{8/7} n)$ parallel calls to procedure *GuessDistance* for each $v \in V \setminus S$. By Lemma 2.19, a single call to procedure *GuessDistance* uses $O(\log n(\log n + \log \Lambda))$ bits. Thus making $O(\log_{1+\zeta'} \Lambda \cdot \log_{8/7} n)$ parallel calls needs $O(\log^2 n \log_{1+\zeta'} \Lambda(\log n + \log \Lambda))$ bits per vertex.

We sample $O(\log_{8/7} n)$ pairwise independent hash functions. Every single pairwise independent hash function requires $O(\log n)$ bits of storage (Lemma 2.6) and thus the set H_p requires $O(\log^2 n)$ bits of storage. We also store three variables $\hat{d}(v)$, $\hat{d}'(v)$ and $\hat{p}(v)$ for every vertex $v \in V \setminus S$. Each of the distance variables $\hat{d}(v)$ and $\hat{d}'(v)$ uses $O(\log \Lambda)$ bits, making the overall cost of their storage $O(n \log \Lambda)$. Each of the parent variables $\hat{p}(v)$ uses $O(\log n)$ bits, making the overall cost of their storage $O(n \log n)$. Hence the overall storage cost of phase p is dominated by the calls to procedure *GuessDistance*. The overall storage cost of any phase is therefore $O(n \cdot \log^2 n \cdot \log_{1+\zeta'} \Lambda(\log n + \log \Lambda))$ bits. □

Observe that the space used in one phase can be reused in the next phase, and this bound is the total space complexity of the algorithm.

In the following lemma, we provide an inductive proof of the correctness of our algorithm. Recall that $|H_p| = c_1 \log_{8/7} n$, where $c_1 > 0$ is a positive constant, and that $\zeta' = \zeta/2\eta$.

Lemma 2.21. *After p phases of our approximate Bellman-Ford exploration algorithm, the following holds for every vertex v within p hops from the set S of source vertices:*

$$d_G^{(p)}(v,S) \leq \hat{d}(v) \leq (1+\zeta')^p \cdot d_G^{(p)}(v,S),$$

with probability at least $1 - p/n^{c_1-1}$. (The left-hand inequality holds with probability 1, and the right-hand inequality holds with probability at least $1 - p/n^{c_1-1}$.)

Proof. The proof follows by induction on the number of phases, p, of the algorithm. The base case for $p = 0$ holds trivially. For the inductive step, we assume that after k phases of our algorithm, with probability at least $1 - k/n^{c_1-1}$, the following holds: For every vertex v within k hops from the set S,

$$d_G^{(k)}(v,S) \leq \hat{d}(v) \leq (1+\zeta')^k \cdot d_G^{(k)}(v,S).$$

In phase $k+1$, we make γ guesses of a new (better) estimate for every $v \in V \setminus S$. We then update the current estimate $\hat{d}(v)$ of v with the smallest guessed value which is better (if any) than the current estimate. Denote by $u \in \Gamma_G(v)$ the neighbour of v on a shortest $(k+1)$-bounded path from v to the set S. By inductive hypothesis, with probability at least $1 - k/n^{c_1-1}$, all k-bounded estimates provide stretch at most

$(1+\zeta')^k$. In particular, $d_G^{(k)}(u,S) \leq \hat{d}(u) \leq (1+\zeta')^k \cdot d_G^{(k)}(u,S)$. Denote by $j = j_v$, the index of a sub-range such that

$$\hat{d}(u) + \omega(u,v) \in I_j.$$

During the execution of the j^{th} guess for vertex v in phase $k+1$, we sample a candidate neighbour $u' \in \Gamma_G(v)$ such that $\hat{d}(u') + \omega(u',v) \in I_j$. Note that u is also a candidate neighbour. By Lemma 2.18, the probability that the procedure *Guess-Distance* fails to find a distance estimate for vertex v in this sub-range is at most $1/n^{c_1}$. By union-bound, the probability that for *for some* vertex $v \in V \setminus S$, we fail to find an estimate for $d_G^{(k+1)}(v,S)$ in the appropriate sub-range is at most $1/n^{c_1-1}$. (Our overall probability of failing to find an estimate of $d_G^{(k+1)}(v,S)$ for some vertex v in the appropriate sub-range is therefore at most $1/n^{c_1-1}$ plus k/n^{c_1-1} from the inductive hypothesis. In total, the failure probability is at most $\frac{k+1}{n^{c_1-1}}$, as required.) We assume henceforth that the j^{th} guess for vertex v is successful.

By induction hypothesis, $\hat{d}(u) \leq (1+\zeta')^k \cdot d_G^{(k)}(u,S)$. Therefore,

$$\begin{aligned}
\hat{d}(u) + \omega(u,v) &\leq (1+\zeta')^k \cdot d_G^{(k)}(u,S) + \omega(u,v) \\
&\leq (1+\zeta')^k \cdot (d_G^{(k)}(u,S) + \omega(u,v)) \\
&= (1+\zeta')^k \cdot d_G^{(k+1)}(v,S).
\end{aligned}$$

Moreover, $(\hat{d}(u') + \omega(u',v))$ and $(\hat{d}(u) + \omega(u,v))$ belong to the same sub-range I_j, and thus,

$$\hat{d}(u') + \omega(u',v) \leq (1+\zeta') \cdot (\hat{d}(u) + \omega(u,v)) \leq (1+\zeta')^{k+1} \cdot d_G^{(k+1)}(v,S).$$

For the lower bound, let $i \leq j$ be the minimum index such that procedure *Guess-Distance* succeeds in finding a neighbour u_i' of v with $(\hat{d}(u_i') + \omega(u_i',v)) \in I_i$. Then, with probability 1 we have, $\hat{d}(u_i') \geq d_G^{(k)}(u_i',S)$, and thus,

$$\hat{d}(v) = \hat{d}(u_i') + \omega(u_i',v) \geq d_G^{(k)}(u_i',S) + \omega(u_i',v) \geq d_G^{(k+1)}(v,S).$$

□

Lemmas 2.20 and 2.21 imply the following Theorem:

Theorem 2.22. *For a sufficiently large positive constant c, given an integer parameter η, an error parameter ζ, an input graph $G(V,E,\omega)$, and a subset $S \subseteq V$,*

the algorithm described in Section 2.4.1 performs, with probability at least $1-\frac{1}{n^c}$, *a* $(1+\zeta)$*-approximate Bellman-Ford exploration of* G *rooted at the set* S *to depth* η, *and outputs for every* $v \in V$, *an estimate* $\hat{d}(v)$ *of its distance to set* S *and* v*'s parent* $\hat{p}(v)$ *on the forest spanned by this exploration such that*

$$d_G^{(\eta)}(v,S) \leq \hat{d}(v) \leq (1+\zeta) \cdot d_G^{(\eta)}(v,S)$$

in η *passes through the dynamic stream using*

$$O_c(\eta/\zeta \cdot \log^2 n \cdot \log\Lambda(\log n + \log\Lambda)) \text{ space in every pass.}$$

The stretch and the space bound follow from Lemmas 2.20 and 2.21 by substituting $\zeta' = \frac{\zeta}{2\eta}$. Note also that the space used by the algorithm on different passes can be reused, i.e., the total space used by the algorithm is $O_c(\eta/\zeta \cdot \log^2 n \cdot \log\Lambda(\log n + \log\Lambda))$.

3

Approximate Shortest Paths in Unweighted Graphs

In this chapter, we present an algorithm for constructing near-additive spanners in dynamic streams and show how a near-additive spanner can be used to compute approximate shortest paths from a set $S \subset V$ of source vertices.

3.1 Construction of Near-Additive Spanners

We use the *superclustering and interconnection* approach introduced by Elkin and Peleg in (26), which was later refined by Elkin and Neiman (22) (randomized version) and Elkin and Matar (21) (deterministic version). Specifically, we adapt the randomized algorithm of (22) to work in the dynamic streaming setting. The main ingredient of both the superclustering and interconnection steps is a set of BFS explorations up to a given depth in the input graph from a set of chosen vertices. As was shown in (22), their algorithm for constructing near-additive spanners can be easily modified to work with the insertion-only streaming model. This is done by identifying the edges spanned by each of the BFS explorations of depth δ (for an integer parameter $\delta \geq 1$) by making δ passes through the stream. Other parts of the spanner construction, such as identifying the vertices of the graph from which to perform BFS explorations and subsequently adding a subset of edges spanned by these explorations to the spanner, can be performed offline. Given parameters $\varepsilon > 0,\ \kappa = 1,2,\ldots$ and $1/\kappa \leq \rho < 1/2$, the basic version of their streaming algo-

rithm constructs a spanner with the same stretch and size as their centralized algorithm, using $O(n^{1+\rho} \cdot \log n)$ space whp and $O(\beta)$ passes through the stream. Recall that $\beta = \beta(\varepsilon, \kappa)$ is defined as $\beta = O(\frac{\log \kappa}{\varepsilon})^{\log \kappa}$ (See also Section 2.1). They also provide a slightly different variant of their streaming algorithm which allows one to trade space for the number of passes. This variant uses only $O(n \log n + n^{1+\frac{1}{\kappa}})$ expected space, but it requires $O((n^{\rho}/\rho) \cdot \log n \cdot \beta)$ passes.

We devise a technique to perform BFS traversals up to a given depth from a set of chosen vertices in the graph in the dynamic streaming setting, and as in (22), perform the rest of the work offline. The algorithm for creating a BFS forest starting from a subset of vertices in the graph is described in Section 2.3. We use the algorithm for creating a BFS forest from a subset of vertices as a subroutine in the superclustering step of our main algorithm. An even bigger challenge we face is during the interconnection step, where each vertex in the graph needs to identify all the BFS explorations it is a part of, and find its path to the source of each such exploration. Due to the dynamic nature of the stream, a given vertex may find itself on a lot more explorations than it finally ends up belonging to. We deal with this problem by combining a delicate encoding/decoding scheme for the IDs of exploration sources with a space-efficient sampling technique.

We first provide a high-level overview of the algorithm for constructing the spanner (26; 22; 21).

Let $G = (V, E)$ be an unweighted, undirected graph on n vertices and let $\varepsilon > 0$, $\kappa = 2, 3, \ldots$ and $1/\kappa \leq \rho < 1/2$ be parameters. The algorithm constructs a sparse $(1+\varepsilon, \beta)$ spanner $H = (V, E_H)$, where $\beta = \left(\frac{\log \kappa\rho + 1/\rho}{\varepsilon}\right)^{\log \kappa\rho + 1/\rho}$ and $|E_H| = O_{\varepsilon,\kappa}\left(n^{1+1/\kappa}\right)$.

The algorithm begins by initializing E_H as an empty set and proceeds in phases. It starts by partitioning the vertex set V into singleton clusters $P_0 = \{\{v\} \mid v \in V\}$. Each phase i for $i = 0, \ldots, \ell$, receives as input a collection of clusters P_i, the distance threshold parameter δ_i and the degree parameter deg_i. The maximum phase index ℓ is set as $\ell = \lfloor \log \kappa\rho \rfloor + \lceil \frac{\kappa+1}{\kappa\rho} \rceil - 1$. The values of δ_i and d_i for $i = 0, 1, \ldots, \ell$, will be specified later in the sequel.

In each phase, the algorithm samples a set of clusters from P_i and these sampled clusters join the nearby unsampled clusters to create bigger clusters called

superclusters. Every cluster created by our algorithm has a designated *center* vertex. We denote by r_C the center of cluster C and say that C is *centered around* r_C. In particular, each singleton cluster $C = \{v\}$ is centered around v. For a cluster C, we define $Rad(C) = max\{d_H(r_C, v) \mid v \in C\}$. For a set of clusters P_i, $Rad(P_i) = \max_{C \in P_i}\{Rad(C)\}$. For a collection P_i, we denote by CP_i the set of centers of clusters in P_i, i.e., $CP_i = \{r_C \mid C \in P_i\}$. A cluster $C \in P_i$ centered around r_C is considered *close* to another cluster $C' \in P_i$ centered around $r_{C'}$, if $d_G(r_C, r_{C'}) \leq \delta_i$.

Each phase i, except for the last one, consists of two steps, the *superclustering* step and the *interconnection* step. For a given set of clusters, interconnecting every pair of clusters within a specific distance from each other by adding shortest paths between their respective centers to the spanner guarantees a pretty good stretch for all the vertices in these clusters. However, if a center is close to a lot of other centers, i.e., it is *popular*, interconnecting it to all the nearby centers can add a lot of edges to the spanner. In order to avoid adding too many edges to the spanner while maintaining a good stretch, the process of interconnecting nearby clusters is preceded by the process of superclustering.

The *superclustering* step of phase i randomly samples a set of clusters in P_i and builds larger clusters around them. The sampling probabilities will be specified in the sequel. For each new cluster C, a BFS tree of C is added to the spanner H. The collection of the new larger clusters is passed on as input to phase $i+1$.

In the *interconnection* step of phase i, the clusters that were not superclustered in this phase are connected to their nearby clusters. For each cluster center r_C that was not superclustered, paths to all the nearby centers in CP_i (whether superclustered or not) are added to the spanner H. Since r_C was not superclustered, it does not have any sampled cluster centers nearby, as otherwise such a center would have superclustered it. This ensures that, with high probability, we do not add too many edges to the spanner during the *interconnection* step.

In the last phase ℓ the superclustering step in skipped and we go directly to the interconnection step. As is shown in (22), the input set of clusters to the last phase P_ℓ is sufficiently small to allow us to interconnect all the centers in P_ℓ to one another using few edges.

Next we describe the input parameters, the degree parameter d_i and the distance threshold parameter δ_i of the phase i, for each $i = 0, 1, \ldots, \ell$. The distance threshold parameter δ_i is defined as $\delta_i = (1/\varepsilon)^i + 4R_i$, where R_i is determined by the following recurrence relation: $R_0 = 0$, $R_{i+1} = R_i + \delta_i$. As is shown in (22), R_i is an upper bound on the radius of the clusters in P_i. The distance threshold parameter δ_i determines the radii of superclusters, and it also affects the definition of nearby clusters for the interconnection step. The degree threshold parameter deg_i of phase i is used to define the sampling probability with which the centers of clusters in P_i are selected to grow superclusters around them. Specifically, in phase i, $i = 0, 1, \ldots \ell - 1$, each cluster center $r_C \in CP_i$ is sampled independently at random with probability $1/deg_i$. The sampling probability affects the number of superclusters created in each phase and hence the number of phases of the algorithm. It also affects the number of edges added to the spanner during the interconnection step. We partition the first $\ell - 1$ phases into two stages based on how the degree parameter grows in each stage. The two stages of the algorithm are the *exponential growth stage* and the *fixed grown stage*. In the *exponential growth stage*, which consists of phases $0, 1, \ldots, i_0 = \log\lfloor \kappa\rho \rfloor$, we set $deg_i = n^{\frac{2^i}{\kappa}}$. In the *fixed growth stage*, which consists of phases $i_0 + 1, i_0 + 2, \ldots, i_1 = i_0 + \lceil \frac{\kappa+1}{\kappa\rho} \rceil$, we set $deg_i = n^{\rho}$. Observe that for every index i, we have $deg_i \leq n^{\rho}$.

3.1.1 Superclustering

In this section, we describe how the superclustering step of each phase $i \in \{0, 1, \ldots, \ell - 1\}$ is executed. The input to phase i is a set of clusters P_i. The phase i begins by sampling each cluster $C \in P_i$ independently at random (henceforth, i.a.r.) with probability $1/deg_i$. Let S_i denote the set of sampled clusters. We now have to conduct a BFS exploration to depth δ_i in G rooted at the set $CS_i = \bigcup_{C \in S_i} \{r_C\}$. At this point, we need to move to the dynamic stream to extract the edges of our BFS exploration. To do so, we invoke the BFS construction algorithm described in Section 2.3.1 with $\eta = \delta_i$ and the set $S = CS_i$ as input. As a result a forest F_i rooted at the centers of the clusters in S_i is constructed. By Theorem 2.17, the construction fo F_i requires δ_i passes and $O(n \log^3 n)$ space whp.

For an unsampled cluster center $r_{C'}$ of a cluster $C' \in P_i \setminus S_i$ such that $r_{C'}$ is spanned by F_i, let r_C be the root of the forest tree in F_i to which $r_{C'}$ belongs. The cluster C' now gets superclustered into a cluster $\widehat{C}$ centered around r_C. The center r_C of C becomes the new cluster center of $\widehat{C}$, i.e., $r_{\widehat{C}} = r_C$. The vertex set of the new supercluster $\widehat{C}$ is the union of the vertex set of the original cluster C, with the vertex sets of all clusters C' which are superclustered into $\widehat{C}$. We denote by $V(C)$ the vertex set of a cluster C. For every cluster center $r_{C'}$ that is spanned by the tree in F_i rooted at r_C, the path in F_i from r_C to $r_{C'}$ is added to the edge set E_H of our spanner H. Recall that E_H is initialized as an empty set (See Section 3.1.).

Let $\widehat{P}_i$ denote the set of new superclusters $\widehat{C}$, that were created by the superclustering step of phase i. We set $P_{i+1} = \widehat{P}_i$. By Theorem 2.17, the superclustering step of phase i generates whp, a forest of the input graph $G(V,E)$, rooted at the set $CS_i \subseteq V$ in δ_i passes. We conclude that:

Lemma 3.1. *For a given set of sampled cluster centers $CS_i \subseteq V$ and a sufficiently large constant c, the superclustering step of phase i builds with probability at least $1 - 1/n^c$, disjoint superclusters that contain all the clusters with centers within distance δ_i from the set of centers CS_i. It does so in δ_i passes through the stream, using $O_c(n\log^3 n)$ space in every pass.*

3.1.2 Interconnection

Next we describe the interconnection step of each phase $i \in \{0,1,\ldots,\ell\}$. Let U_i denote the set of clusters of P_i that were not superclustered into clusters of $\widehat{P}_i$. For the phase ℓ, the superclustering step is skipped and we set $U_\ell = P_\ell$.

In the interconnection step of phase $i \geq 1$, we want to connect every cluster $C \in U_i$ to every other cluster $C' \in P_i$ that is close to it. To do this, every cluster center r_C of a cluster $C \in U_i$ performs a BFS exploration up to depth $\frac{1}{2}\delta_i$, i.e., half the depth of BFS exploration which took place in the superclustering step, as in (22). For each cluster center $r_{C'}$ of some cluster $C' \in P_i$ which is discovered by the exploration initiated in r_C, the shortest path between r_C and $r_{C'}$ is inserted into the edge set E_H of our spanner. In the first phase $i = 0$, however, we set the exploration depth δ_0 to 1, i.e., to the same value as in the superclustering step. Essentially, for every vertex $v \in U_0$, we add edges to all its neighbours to H.

Having identified the members of U_i, we turn to the stream to find the edges belonging to the BFS explorations performed by the centers of clusters in U_i. The problem here is that we need to perform many BFS explorations in parallel. More precisely, there are up to $|P_i|$ explorations in phase i. By Lemma 3.5 of (22), $|P_i| = n^{1-\frac{2^i-1}{\kappa}}$ in expectation for $i \in \{0,1,\ldots,i_0\}$ and $|P_i| \leq n^{1+1/\kappa-(i-i_0)\rho}$ in expectation for $i \in \{i_0+1, i_0+2, \ldots, \ell\}$. Recall that $i_0 = \lfloor \log \kappa\rho \rfloor$. Invoking Theorem 2.17 for $\eta = \delta_i/2$, $S = \{r_C\}$, for some cluster center r_C of a cluster in U_i, a BFS exploration of depth $\delta_i/2$, rooted at r_C requires $O(n\log^3 n)$ space and $\delta_i/2$ passes. Running $|P_i|$ explorations in G requires either $O(|P_i| \cdot n\log^3 n)$ space or $|P_i| \cdot \delta_i/2$ passes. Both these resource requirements are prohibitively large.

We state the following Lemma from (22) here for completeness.
We refer the reader to (22) for the proof.

Lemma 3.2 ((22))**.** *For any vertex $v \in V$, the expected number of explorations that visit v in the interconnection step of phase i is at most deg_i. Moreover, for any constant c_1', with probability at least $1 - 1/n^{c_1'-1}$, no vertex v is explored by more than $c_1' \cdot \ln n \cdot d_i$ explorations in phase i.*

In (22), Lemma 3.2 is used to argue that the overall space used by their streaming algorithm in phase i is $O(n \cdot deg_i \log n)$ in expectation. Furthermore, since $deg_i \leq n^\rho$ for all $i \in \{0,1,\ldots,\ell\}$, the space used by their streaming algorithm is $O(n^{1+\rho}\log n)$ in expectation in every pass. Unfortunately, this argument does not help us to bound the space usage of our algorithm in the dynamic setting. When edges may appear as well as disappear, a given vertex v may appear on a lot more explorations than deg_i as the stream progresses. Lemma 3.2 only guarantees that ultimately paths to at most deg_i centers in U_i will survive for v in expectation. If we record for every $v \in V$, all the explorations passing through v to identify the ones that finally survive, we incur a cost of $O(|P_i| \cdot n\log^3 n)$ space for interconnection during phase i, which is prohibitively large.

To tackle this problem, we devise a randomized technique for every vertex to efficiently identify all the (*surviving*) explorations that it gets visited by in phase i. For every vertex $v \in V$ with a non-empty subset $U_i^v \subseteq U_i$ of explorations that visit v, we find for every cluster $C \in U_i^v$, a neighbour of v on a shortest path between v and the center r_C of C. (See Figure 3.1.)

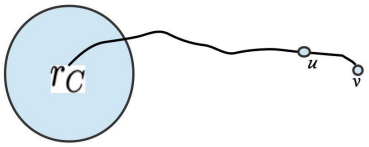

Figure 3.1: A cluster $C \in U_i^v$. The algorithm finds the neighbour u of v on the shortest $r_C - v$ path.

Throughout the interconnection step of phase i, we maintain for each vertex $v \in V$, a running set L_v of exploration sources that visited v. Each vertex s in L_v is a center of a cluster $C \in U_i$. We will call the set L_v the *visitor list* of v. Initially the visitor lists of all the vertices are empty, except for the centers of clusters in U_i. The center r_C of every $C \in U_i$ is initialized with a single element r_C in its visitor list.

The interconnection step of phase i is carried out in $\lfloor \delta_i/2 \rfloor$ sub-phases. Each sub-phase of the interconnection step makes two passes through the stream. In the following section, we describe the purpose of each of the $\lfloor \delta_i/2 \rfloor$ sub-phases of the interconnection step and the way they are carried out.

Sub-phase j of interconnection step

We discover the edges belonging to the layer j of interconnection in the sub-phase j. By layer j of interconnection, we mean the set containing every vertex v in V, whose distance to one or more cluster centers in U_i is exactly j. Note that a given vertex v may belong to more than one layer of interconnection since it may be at different distances from different exploration sources, and we need to identify all the exploration sources in U_i that are within distance $\lfloor \delta_i/2 \rfloor$ from v.

The information regarding the j^{th} layer of interconnection is stored in a set called S_j. Formally, the set S_j consists of tuples of the form (v,s,k), where s is an exploration source at distance j from v, and k is the number of neighbours of v at a distance $j-1$ from s. While the visitor list L_v of a specific vertex $v \in V$ maintains a list of all the exploration sources that visit v in all the sub-phases of the interconnection step, the set S_j is a global list that stores for each vertex $v \in V$, the information about the exploration sources that visited v during sub-phase j.

Before we start the sub-phase j, we create for each $v \in V$, a copy L'_v of its running visitor list L_v. Any new explorations discovered during the sub-phase j are added to the shadow visitor list L'_v. Specifically, L_v is the list of those cluster centers from U_i whose explorations visited v *before* sub-phase j started, and L'_v is the list of those centers that visited v on one of the first j sub-phases.

In each of the $\lfloor \delta_i/2 \rfloor$ sub-phases, we make two passes through the stream. In the first pass of sub-phase j, we construct the set S_j. In more detail, for each vertex $v \in V$, we use a sampler repeatedly in parallel (the exact number of parallel repetitions will be specified later in the sequel) to extract whp all the exploration sources (if there are any) at a distance j from v. A tuple (v, s, k_v), for some $k_v \geq 1$, is added to the set S_j for every source s extracted by the sampler. The visitor list L_v of v is also updated with the new exploration sources that were observed in this sub-phase. Specifically, all newly observed exploration sources are added to L'_v. At the end of the sub-phase we set $L_v \leftarrow L'_v$.

The second pass of sub-phase j uses the sets S_j and S_{j-1} to find for every $v \in S_j$, its parent on every exploration whose source is at distance j from v. Note that a parent of v on an exploration rooted at the source s is a vertex at distance $j-1$ from s. Therefore, we need the set S_{j-1} to extract an edge between v and some vertex u such that a tuple (u, s, k_u), for some $k_u \geq 1$, belongs to the set S_{j-1}.

The set S_{j-1}, which is constructed during the first pass of phase $j-1$, is used as an input for the second pass of phases $j-1$ and j. It is therefore kept in global storage until the end of phase j.

We next describe how we construct the set S_j during the first pass of sub-phase j.

First pass of sub-phase j of phase i: Let c'_1 be a sufficiently large positive constant (See Lemma 3.2.), and let $\mathcal{N}_i = c'_1 \cdot deg_i \cdot \ln n$. For each $v \in V$, we make $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$ *attempts* in parallel, for some sufficiently large constant $c_4 \geq 1$. In each attempt, we invoke a randomized procedure *FindNewVisitor* to find an exploration source in U_i at a distance j from v. The pseudocode for procedure *FindNewVisitor* is given in Algorithm 3. The procedure *FindNewVisitor* takes as input the ID of a vertex v and a hash function h, chosen at random from a family of pairwise independent hash functions. It returns a tuple (s, d_s), where s is the ID of an exploration source at distance j from v, and d_s is the number of neighbours of v

that are at distance $j-1$ to s. This source s is then added to the shadow visitor list L'_v of the vertex v. If there are no exploration sources at distance j from v, procedure *FindNewVisitor* returns a tuple (ϕ,ϕ). If there are some exploration sources at distance j from v but procedure *FindNewVisitor* fails to isolate an ID of such a source, it returns $(\perp,\perp)$.

Before we start making our attempts in parallel, we sample uniformly at random a set H_j of μ_i functions from a family of pairwise independent hash functions $h:\{1,2,\ldots,maxVID\}\rightarrow\{1,\ldots,2^{\lambda}\}$, where $\lambda=\lceil\log maxVID\rceil=\lceil\log n\rceil$. Having sampled the set H_j of hash functions, for every vertex $v\in V$, we make $\mu_i=|H_j|$ parallel calls to procedure *FindNewVisitor*(v,h), one call for each function $h\in H_j$.

Note that the visitor lists of all the vertices in V are visible to all the calls to procedure *FindNewVisitor*, which are made in parallel.

Procedure FindNewVisitor: A call to procedure *FindNewVisitor* for a vertex v tracks the edges between v and every vertex u with some explorations in its visitor list L_u that v has not seen so far. Let $d_v^{(j)}$ be the number of exploration sources at distance j from v. For every pair of vertices $\{v,u\}$, ultimately either the edge $e=(v,u)$ belongs to G and then $f_e=1$, or it does not, i.e., $f_e=0$. (Recall that $f_e=\sum_{t,e_t=e} eSign_t$ is the multiplicity of edge e in the stream.) If we knew the exact value of $d_v^{(j)}$, we could sample every new exploration source witnessed by v with probability $1/d_v^{(j)}$ to extract exactly one of them in expectation. However, all we know about $d_v^{(j)}$ is that it is at most deg_i in expectation (Lemma 3.2) and at most $O(deg_i\cdot\ln n)$ whp. We therefore sample every new exploration source seen by v on a range of probabilities, as we did for procedure *FindParent* in Section 2.3.2. We use an array *slots* of λ elements (the structure of each element will be described later in the sequel), indexed by *slot-levels* from 1 to $\lambda=\lceil\log n\rceil$, to implement sampling on a range of probabilities. We want a given source s to be sampled into slot-level k with probability $1/2^{\lambda-k}$. When $d_v^{(j)}\approx 2^{\lambda-k}$, with a constant probability there is exactly one exploration source that gets mapped to $slots[k]$.

One way to sample every exploration seen by v with a given probability is to flip a biased coin. As was discussed in Section 2.3.2 in the description of procedure *FindParent*, naively, this requires remembering the random bits for every new exploration source seen by v. To avoid storing that much information while still treating all the updates (additions/deletions) to a given exploration source con-

Algorithm 3 Pseudocode for procedure *FindNewVisitor*

1: **Procedure FindNewVisitor**(v,h) ▷ Initialization
2: $slots \leftarrow \emptyset$
▷ An array with $\lambda = \lceil \log n \rceil$ elements indexed from 1 to λ.
3:
▷ Each element of slots is a tuple $(sCount, sNames)$. For a given index $1 \leq k \leq \lambda$, fields *sCount* and *sNames* of $slots[k]$ can be accessed as $slots[k].sCount$ and $slots[k].sNames$, respectively.
4:
▷ $slots[k].sCount$ counts the new exploration sources seen by v with hash values in $[2^k]$.
▷ $slots[k].sNames$ is an encoding of the names of new exploration sources seen by v with hash values in $[2^k]$.
▷ Update Stage
5: **while** (there is some update $(e_p,\ eSign_p)$ in the stream) **do**
6: **if** $(e_p = (v,u)$ satisfies $L_u \setminus L_v \neq \emptyset)$ **then**
7: **for each** $s \in L_u \setminus L_v$ **do**
8: $k \leftarrow \lceil \log h(s) \rceil$
9: **repeat** ▷ Update $slots[k]$ for all $\lceil \log h(s) \rceil \leq k \leq \lambda$
10: $slots[k].sCount \leftarrow slots[k].sCount + eSign_p$
11: $slots[k].sNames \leftarrow slots[k].sNames + \nu(s) \cdot eSign_p$
12: ▷ The function ν is described in Section 2.2.4.
13: ▷ The addition in line 11 is a vector addition.
14: $k = k+1$
15: **until** $k > \lambda$
▷ Recovery Stage
16: **if** (*slots* vector is empty) **then**
17: return (ϕ, ϕ)
18: **else if** ($\exists$ index k s.t. $\frac{slots[k].sName}{slots[k].sCount} = \nu(s)$ for some s in V) **then**
19: return $(s, slots[k].sCount)$
20: **else**
21: return $(\perp, \perp)$

sistently, we use pairwise independent hash functions for sampling explorations. Given a hash function $h : \{1,2,\ldots,maxVID\} \to \{1,\ldots,2^{\lambda}\}$, every new exploration source s witnessed by v is assigned a hash value $h(s)$ by h. A given source s gets mapped into $slots[k]$ if $h(s) \in [2^k]$, i.e., this happens with probability $1/2^{\lambda-k}$. The description of procedure *FindNewVisitor* is similar to procedure *FindParent* from Section 2.3.2 up to this point. The major difference between procedure *FindParent* and procedure *FindNewVisitor* is in the information that we store about every sample in a given slot. We cannot afford storing the IDs of all the sampled exploration sources as v may appear on many more explorations than it ends up on. Every new exploration source s assigned to $slots[k]$ is first encoded using the CIS encoding scheme ν described in Section 2.2.4. In every element of *slots*, we maintain a tuple $(sCount, sNames)$, where $sCount \in \mathbb{Z}$ at slot-level k maintains the number of new exploration sources seen by v with hash values in $[2^k]$, and $sNames \in \mathbb{Z}^2$ maintains the vector sum of encodings of the IDs of new exploration sources seen by v with hash values in $[2^k]$. This will be discussed in detail in the sequel. The fileds *sCount* and *sName* of $slots[k]$ can be accessed as $slots[k].sCount$ and $slots[k].sName$, respectively.

As the stream progresses, every time we encounter an exploration source s with $h(s) \in [2^k]$, we update the *sCount* value of $slots[k]$ with the *eSign* value of the edge from which s was extracted. (See line 10 of Algorithm 3.) Also, we update the *sNames* of $slots[k]$ by adding $\nu(s) \cdot eSign_p$ to it (see line 11 of Algorithm 3), where $\nu(s)$ is the encoding of the source s and $eSign_p$ is the *eSign* value of the edge from which s was extracted. (This addition sums up vectors in $\mathbb{Z}^2$.) In line 18 of Algorithm 3, we use Lemma 2.12 to determine if there is a slot-level k such that only one exploration source was sampled at that level. Note that the CIS encoding scheme that we use here is more general and can also be used in the implementation of procedure *FindParent*. The bitwise XOR-based technique that we use in procedure *FindParent* is an existing technique based on (34) and (41) that works for sampling a non-zero element from a Boolean vector. The CIS-based technique, on the other hand, allows one to sample a non-zero element from a vector with non-negative entries.

If there is a slot-level k for which $\frac{slots[k].sName}{slots[k].sCount} = \nu(s)$ for some $s \in V$, then by Lemma 2.12, s is the only exploration source sampled at slot-level k. The value of

sCount at slot-level k will then be the number of neighbours of v at distance $j-1$ from s.

We need to make sure that for some $1 \leq k \leq \lambda$, exactly one exploration source will get mapped to $slots[k]$. By Corollary 2.8, exactly one exploration source gets mapped to $slots[k]$ for $k = \lambda - \lceil \log d_v^{(j)} \rceil - 1$, with at least a constant probability. (Here $\mathscr{S}$ is the set of exploration sources at distance j from v and $s = |\mathscr{S}| = d_v^{(j)}$.) Therefore, a single call to procedure *FindNewVisitor* succeeds with at least a constant probability.

Analysis of first pass: We now analyze the success probability and space requirements of the first pass of sub-phase j of interconnection step.

Recall that, for every vertex $v \in V$, we make $\mu_i = 16 \cdot c_4 \cdot \mathscr{N}_i \cdot \ln n$ parallel attempts to isolate the exploration sources that visit v during sub-phase j of the interconnection step of phase i.

Lemma 3.3. *On any single attempt for a vertex $v \in V$, a given exploration source s at distance j from v is discovered with probability at least $\frac{1}{16\mathscr{N}_i}$.*

Proof. Recall that by Lemma 3.2, with probability at least $1 - \frac{1}{n^{c_1'-1}}$, the number $d_v^{(j)}$ of the exploration sources that visit v is at most $\mathscr{N}_i$. For a specific exploration source s that visits v during sub-phase j, let $DISC^{(s)}$ denote the event that it is discovered in a specific attempt. Then:

$$\begin{aligned} Pr\left[DISC^{(s)}\right] &\geq Pr\left[DISC^{(s)} \mid d_v^{(j)} \leq \mathscr{N}_i\right] \cdot Pr\left[d_v^{(j)} \leq \mathscr{N}_i\right] \\ &\geq Pr\left[DISC^{(s)} \mid d_v^{(j)} \leq \mathscr{N}_i\right] \cdot \left(1 - \frac{1}{n^{c_1'-1}}\right) \\ &\geq \frac{1}{8\mathscr{N}_i}\left(1 - \frac{1}{n^{c_1'-1}}\right) \\ &\geq \frac{1}{16\mathscr{N}_i} \end{aligned}$$

Note that the third inequality follows by applying Lemma 2.7 to the event $\{DISC^{(s)} \mid d_v^{(j)} \leq \mathscr{N}_i\}$. □

In the next lemma we argue that procedure *FindNewVisitor* does not require too much space.

Lemma 3.4. *The procedure FindNewVisitor uses $O(\log^2 n)$ bits of memory.*

Proof. Procedure *FindNewVisitor* receives as input two variables: the ID of a vertex v and a pairwise independent hash function h. The ID of any vertex requires $O(\log n)$ bits of space and by Lemma 2.6, a pairwise independent hash function can be encoded in $O(\log n)$ bits too. The visitors lists of all the vertices are available in global storage. The internal variable $slots$ is an array of size $\lceil \log n \rceil$. Each element of the array $slots$ stores an integer counter $sCounter$ of size $O(\log n)$ bits and an integer vector $sNames$ in $\mathbb{Z}^2$, which also requires $O(\log n)$ bits of space (See Section 2.2.4). The space usage of $slots$ array is therefore $O(\log^2 n)$ bits. It follows thus that procedure *FindNewVisitor* uses $O(\log^2 n)$ bits of memory. □

For a vertex $v \in V$, if there are no exploration sources at a distance j from v, all the calls to procedure *FindNewVisitor* in all the attempts return (ϕ, ϕ). For all those vertices, we do not need to update their visitor lists. For every other vertex $v \in V$, each attempt yields the name of an exploration source at a distance j from v with at least a constant probability. We extract the names of all the *distinct* exploration sources from the results of successful attempts and add tuples $(v, s, sCount)$ to the set S_j. Recall that the set S_j contains tuples (v, s, k_v), where s is an exploration source at distance j from v and k_v is the number of neighbours of v that are at distance $j-1$ from s. In addition, the source s is added to the visitor list L'_v of vertex v.

We next show that making $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$ attempts in parallel for every vertex $v \in V$ ensures that all the relevant exploration sources for every vertex are extracted whp.

Lemma 3.5. *Let c_3 be a sufficiently large constant. For a given vertex $v \in V$, with probability at least $1 - 1/n^{c_3}$, all the exploration sources at a distance j from v will be successfully extracted in $\mu_i = 16 \cdot c_4 \cdot \ln n \cdot \mathcal{N}_i$ attempts made in parallel for v in the first pass of sub-phase j.*

Proof. For a given vertex v, let $d_v^{(j)}$ be the number of explorations that are at a distance j from v. By Lemma 3.3, on each single attempt (out of μ_i attempts) for a vertex v, a specific exploration source that visits v is isolated with probability at least $1/16\mathcal{N}_i$, independently of other attempts. Thus, for a given exploration source s, the probability that no attempt will isolate it is at most $\left(1 - \frac{1}{16\mathcal{N}_i}\right)^{16 \cdot c_4 \cdot \ln n \cdot \mathcal{N}_i} \leq 1/n^{c_4}$. Hence, by union-bound over all the exploration sources at distance j from

v, all the exploration sources will be isolated during $16 \cdot c_4 \cdot \ln n \cdot \mathcal{N}_i$ attempts, with probability at least $1 - \frac{1}{n^{c_4-1}}$. Thus, for $c_3 = c_4 - 1$, with probability at least $1 - 1/n^{c_3}$, all the exploration sources at a distance j from v will be successfully extracted. □

We next provide an upper bound on the space usage of the first pass of the interconnection step.

Lemma 3.6. *The overall space usage of the first pass of every sub-phase of interconnection is $O(n^{1+\rho}\log^4 n)$ bits.*

Proof. The first pass of every sub-phase makes $\mu_i = O(deg_i \cdot \log^2 n)$ attempts in parallel for every $v \in V$. Recall that for all i, $deg_i \leq n^\rho$ (See Section 3.1). Combining this fact with Lemma 3.4, we get that the space usage of all the invocations of procedure *FindNewVisitor* for all the n vertices during the first pass is $O(n^{1+\rho}\log^4 n)$. In addition, we use a set of $O(deg_i \cdot \log^2 n) = O(n^\rho \cdot \log^2 n)$ randomly sampled hash functions, one hash function per attempt. Each hash function can be encoded using $O(\log n)$ bits. The overall space used by the storage of hash functions during the first phase is thus $O(n^\rho \log^3 n)$. As an output, we produce the set S_j, which consists of tuples (v,s,k) of $O(\log n)$ bits each. By Lemma 3.2, a vertex v is visited by at most $O(deg_i \log n) \leq O(n^\rho \log n)$ explorations whp in the phase i. In any case, we record just $O(n^\rho \cdot \log n)$ of them, even if v is visited by more explorations. Hence, the storage of S_j requires $O(n^{1+\rho}\log^2 n)$ bits. Finally, we need to store the visitor lists of all $v \in V$. By Lemma 3.2, no vertex is visited by more than $O(deg_i \log n) = O(n^\rho \log n)$ explorations whp. As above, we record just $O(n^\rho \cdot \log n)$ of the visitors for v. We need to store $O(\log n)$ bits of information for every exploration source that visited a given vertex. The overall storage cost of all the visitor lists of all the vertices is therefore $O(n^{1+\rho}\log^2 n)$ bits. Thus, the storage cost of first pass of every sub-phase is dominated by the cost of parallel invocations of procedure *FindNewVisitor*. This makes the overall cost of first pass of every sub-phase $O(n^{1+\rho}\log^4 n)$. □

Second pass of sub-phase j of Phase i: The second pass of sub-phase j starts with the sets S_{j-1} and S_j as input. Recall that the set S_j consists of tuples for all the vertices in V that are at distance j from one or more exploration sources in U_i. The algorithm also maintains an additional intermediate edge set $\hat{H}_i$, which will

contain all the BFS trees rooted at cluster centres r_C, $C \in U_i$, constructed to depth $\delta_i/2$. Inductively, we assume that before sub-phase j starts, the edge set $\hat{H}_i$ contains the first $j-1$ levels of these trees. Note that since by Lemma 3.2, whp, every vertex v is visited by $O(deg_i \cdot \log n)$ explorations rooted at $\{r_C\}_{C \in U_i}$, it follows that, whp, $|\hat{H}_i| = \tilde{O}(n \cdot deg_i) = O(n^{1+\rho} \cdot \log n)$. Thus our algorithm can store the set $\hat{H}_i$. We find for every tuple (v,s,k) in S_j, v's parent p_s on the exploration rooted at s by invoking procedure *FindParent* (described in Section 2.3.2) $O(\log n)$ times. As a result, an edge (v,p_s) between v and p_s is added to the edge set $\hat{H}_i$.

We sample uniformly at random a set of pairwise independent hash functions H'_j, $|H'_j| = c_1 \cdot \log_{8/7} n$, from the family of functions $h : \{1,2,\ldots,maxVID\} \rightarrow \{1,2,\ldots,2^\lambda\}$, $\lambda = \lceil \log n \rceil$. These functions will be used by invocations of procedure *FindParent*.

We need to change slightly the original procedure *FindParent* (Section 2.3.2) to work here. Specifically, we change the part where we decide whether to sample an incoming edge update or not (Line 15 of Algorithm 1). It is updated to check if the edge e_p is incident between the input vertex v and some vertex u such that for some k, the tuple (u,s,k) belongs to the set S_{j-1}. Recall that for a tuple $(v,s,k) \in S_j$, k is the number of neighbours of v that are at a distance $j-1$ from s. This information can be used to optimize the space usage of procedure *FindParent* by a factor of $O(\log n)$. Since we know the probability ($\approx 1/k$) with which to sample every candidate edge for v, we can get rid of the array *slots* and maintain only two running variables *xCount* and *xName* corresponding to slot-level $\lambda - \lceil \log k \rceil - 1$.

Finally, after all the $\delta_i/2$ sub-phases are over, we extract from $\hat{H}_i$ edges that need to be added to the spanner H offline, during post-processing. Specifically, for every cluster center r_C, $C \in U_i$, we consider the BFS tree $T(r_C)$ rooted at r_C of depth $\delta_i/2$, which is stored in $\hat{H}_i$. For any leaf z of $T(r_C)$ which is not a center of a cluster $C' \in P_i$, the leaf z and the the edge connecting z to its parent p_z in $T(r_C)$ are removed from $T(r_C)$ (and thus from $\hat{H}_i$). This process is then iterated, until all leaves of $T(r_C)$ are cluster centers. This is done for all cluster centers r_C, $C \in U_i$, one after another. The resulting edge set H'_i (a subset of $\hat{H}_i$) is then added to the spanner H.

Observe that this edge set H'_i is precisely the union of all shortest paths $r_C - r'_C$, for $C \in U_i$ and $C' \in P_i$, such that $d_G(r_c, r'_C) \leq \delta_i/2$. It follows that, (see (22)), its size

is at most $\delta_i/2 \cdot |U_i| \cdot deg_i = \tilde{O}(\delta_i \cdot n^{1+1/\kappa})$. This bound can be further refined by optimizing the degree sequence $(deg_i)_{i=1}^{\ell}$. (See (22) for details.)

Analysis of Second Pass: We now analyze the space requirements of the second pass of sub-phase j of interconnection step.

Lemma 3.7. *The overall space usage of the second pass of every sub-phase of interconnection is* $O(n^{1+\rho}\log^4 n)$.

Proof. The second pass of every sub-phase invokes procedure *FindParent* $O(\log n)$ times in parallel for every tuple in the set S_j. By Lemma 2.14, each invocation of procedure *FindParent* uses $O(\log^2 n)$ bits of space. The number of elements in S_j is at most $O(n^{1+\rho}\log n)$. (Recall that by Lemma 3.2, whp there are at most $\tilde{O}(n^{\rho})$ explorations per vertex. But even if there are more explorations, our algorithm records just $\tilde{O}(n^{\rho})$ explorations per vertex.) Therefore the overall cost of all the invocations of procedure *FindParent* is $O(n^{1+\rho}\log^4 n)$. In addition, we need to store a set of $O(\log n)$ hash functions of size $O(\log n)$ each in global storage. This requires $O(\log^2 n)$ bits of space. Therefore, the overall storage cost of the second pass of any sub-phase is dominated by the space required for invocations of *FindParent*. Hence the overall space requirement of second pass of interconnection is $O(n^{1+\rho}\log^4 n)$. □

In the following lemma we prove the correctness of the interconnection step.

Lemma 3.8. *For a sufficiently large constant* c'*, after* j *sub-phases of phase* i *of the interconnection step, with probability at least* $1 - j/n^{c'}$*, for every cluster* $C \in U_i$ *and for every vertex* v *within distance* j *from the center* r_C *of* C*, a shortest path between* r_C *and* v *is added to the edge set* $\hat{H}_i$.

Proof. The proof follows by induction on the number of sub-phases, j, of the interconnection step of phase i. The base case for $j = 0$ holds trivially. For the inductive step, we assume that after $j = t$ sub-phases of interconnection step (Section 3.1.2), for every cluster $C \in U_i$ and for every vertex v within distance t from the center r_C of C, a shortest path between r_C and v has been added to $\hat{H}_i$ with probability at least $1 - t/n^{c'}$. Given this assumption, we only need to prove that in the sub-phase $t+1$, we find for every cluster $C \in U_i$ and for every vertex v at distance $t+1$ from the center r_c of C, a parent for v on the BFS exploration rooted

at r_C with probability at least $1 - 1/n^{c'}$. In the first pass of sub-phase $t+1$, for every vertex $v \in V$, we make $\mu_i = 16 \cdot c_4 \cdot \ln n \cdot \mathcal{N}_i$ attempts to extract all the cluster centers at distance $t+1$ from v. By Lemma 3.5, each such center gets extracted with probability at least $1 - 1/n^{c_3}$. There are no more than n clusters in U_i. Applying union bound over all the clusters in U_i and over all the vertices at distance $t+1$ from one or more centers in U_i, we successfully extract all the exploration sources at distance $t+1$ from every vertex in the sub-phase $t+1$ with probability at least $1 - 1/n^{(c_3-2)}$. In the second pass of sub-phase $t+1$, we try to find a parent for v on every exploration at distance $t+1$ by making multiple parallel calls to procedure *FindParent*. By Lemma 2.13, we succeed in finding a parent for v on a single BFS exploration with probability at least $1 - 1/n^{c_1}$. By union bound over all the clusters in U_i and all the vertices at distance $t+1$ from one or more centers, the second pass of sub-phase $t+1$ succeeds with probability at least $1 - 1/n^{c_1-2}$. Taking a union bound on both the passes of sub-phase $t+1$, we get that for an appropriate constant c', in the sub-phase $t+1$, for every cluster $C \in U_i$ and for every vertex v at distance $t+1$ from the center r_c of C, we find a parent for v on the BFS exploration rooted at r_C with probability at least $1 - 1/n^{c'}$. □

Lemmas 3.6, 3.7 and 3.8 together imply the following corollary about the interconnection step of phase i:

Corollary 3.9. *For a sufficiently large constant c'', after $\lfloor \delta_i/2 \rfloor$ sub-phases of phase i of the interconnection step, the following holds with probability at least $1 - 1/n^{c''}$:*

1. *The interconnection step of phase i makes δ_i passes through the stream, and the total required space is $O(n^{1+\rho} \log^4 n)$ bits.*
2. *For every cluster $C \in U_i$ and every other cluster $C' \in P_i$ such that the centers r'_C of C' is within distance $\lfloor \delta_i/2 \rfloor$ from center r_C of C, a shortest $r_C - r_{C'}$ path between them is added to the spanner.*

3.1.3 Putting Everything Together

Lemma 3.1 and Corollary 3.9 imply that, whp, our algorithm simulates phase i of (22). The following lemma follows by induction on the number of phases of our algorithm.

Lemma 3.10. *After ℓ phases, whp, our spanner construction algorithm simulates the algorithm of (22) in the dynamic streaming setting.*

Next, we provide a bound on the number of passes of our algorithm.

Lemma 3.11. *Our spanner construction algorithm makes $O(\beta)$ passes in total.*

Proof. In a given phase i of our construction algorithm, the superclustering step makes δ_i passes and the interconnection step makes $2\lfloor \delta_i/2 \rfloor$ passes. The number of passes of phase i is therefore bounded by $O(\delta_i)$. Note that $\sum_{i=1}^{\ell} \delta_i = O(\beta)$, where β is the additive term in the stretch of our construction (See (22)). The number of passes made altogether is thus bounded by $O(\beta)$. □

The stretch and sparsity analysis of our dynamic streaming algorithm remains the same as that of the centralized algorithm of (22). Hence we obtain the following analogue of Corollary 3.2 of (22) for the dynamic streaming setting.

Theorem 3.12. *For any unweighted graph $G(V,E)$ on n vertices, parameters $0 < \varepsilon < 1$, $\kappa \geq 2$, and $\rho > 0$, our dynamic streaming algorithm computes a $(1+\varepsilon,\beta)$-spanner with $O_{\varepsilon,\kappa,\rho}(n^{1+1/\kappa})$ edges, in $O(\beta)$ passes using $O(n^{1+\rho}\log^4 n)$ space with high probability, where β is given by:*

$$\beta = \left(\frac{\log \kappa\rho + 1/\rho}{\varepsilon}\right)^{\log \kappa\rho + 1/\rho}.$$

In the following section, we show some applications of our construction of near-additive spanners.

3.2 $(1+\varepsilon)$-Approximate Shortest Paths in Unweighted Graphs

An immediate application of our dynamic streaming algorithm for constructing $(1+\varepsilon,\beta)$-spanners is a dynamic streaming algorithm for computing *all pairs almost shortest paths* (APASP) with multiplicative stretch $1+\varepsilon$ and additive stretch

β (henceforth, $(1+\varepsilon,\beta)$-APASP) in unweighted undirected graphs. The algorithm uses $O(\beta)$ passes over dynamic stream and $\tilde{O}(n^{1+\rho})$ space. Our $(1+\varepsilon,\beta)$-APASP algorithm computes a $(1+\varepsilon,\beta)$-spanner with $O_{\varepsilon,\kappa,\rho}(n^{1+1/\kappa})$ using Theorem 3.12, and then computes offline all pairs exact shortest paths in the spanner.

We note also that within almost the same complexity bounds, the algorithm can also compute $(1+\varepsilon)$-approximate shortest paths $S\times V$ (henceforth, $(1+\varepsilon)$-ASP), for a subset S of size n^{ρ} of designated sources. Specifically, the algorithm computes the $(1+\varepsilon,\beta)$-APASP in the way described above. It then uses $O(\beta/\varepsilon)$ more passes to compute BFS trees rooted in each of the sources $s\in S$ to depth β/ε in the original graph G. The space usage of this step is $\tilde{O}(|S|\cdot n)=\tilde{O}(n^{1+\rho})$. (see Theorem 2.17)

As a result, for every pair $(s,v)\in S\times V$ such that $d_G(s,v)\leq\beta/\varepsilon$, our algorithm returns an exact distance. For each pair $(s,v)\in S\times V$ with $d_G(s,v)>\beta/\varepsilon$, the estimate computed using $(1+\varepsilon,\beta)$-APASP algorithm provides a purely multiplicative stretch of $1+O(\varepsilon)$. The algorithm returns the minimum of these two estimates.

By setting $\kappa=1/\rho$ we obtain:

Theorem 3.13. *For any undirected n-vertex graph $G=(V,E)$, and any $\varepsilon>0$, $\rho>0$, our dynamic streaming algorithm computes $(1+\varepsilon,\beta)$-APASP and $(1+\varepsilon)$-ASP for a set S of $|S|=n^{\rho}$ sources using $\beta=O(\frac{1}{\rho\varepsilon})^{\frac{1}{\rho}(1+o(1))}$ passes and $\tilde{O}(n^{1+\rho})$ memory.*

One noteable point on the tradeoff curve is $\rho=\sqrt{\frac{\log\log n}{\log n}}$. Then we get $2^{O(\sqrt{\log n\cdot\log\log n})}$ passes and $n\cdot 2^{O(\sqrt{\log n\cdot\log\log n})}$ space. Also using $\rho=\frac{(\log\log n)^c}{\log n}$ for sufficiently large constant c, we get $n^{o(1)}$ passes and $\tilde{O}(n)$ space.

4

Approximate Shortest Paths in Weighted Graphs

In this chapter we present an algorithm for constructing hopsets with stretch $(1+\varepsilon)$ (for some constant precision parameter $\varepsilon > 0$), size $\tilde{O}(n^{1+1/\kappa})$ (for a constant integer parameter κ) and constant hopbound $\beta = \beta(\varepsilon, \kappa)$, in dynamic streaming model. We call these hopsets *near-exact hopsets*. We then use our near-exact hopsets to compute approximate shortest paths.

4.1 Construction of Near-Exact Hopsets

Our hopset construction algorithm is based on superclustering and interconnection approach that was originally devised for the construction of near-additive spanners (26). (See Chapter 3 for more details.) Elkin and Neiman (23) used the superclustering and interconnection approach for the construction of hopsets with constant hopbound in various models of computation including the insertion-only streaming model. We adapt here the insertion-only streaming algorithm of (23) to work in the dynamic streaming setting.

The main ingredient of both the superclustering and interconnection steps is a set of Bellman-Ford explorations (*BF explorations* henceforth) up to a given distance in the input graph from a set of chosen vertices. The insertion-only streaming algorithm of (23) identifies all the edges spanned by $\Theta(\beta)$ iterations of certain BF explorations up to a distance δ from a set of chosen vertices, by making $\Theta(\beta)$ passes

through the stream. Other parts of the hopset construction, such as identifying the vertices of the graph from which to perform BF explorations and subsequently adding edges corresponding to certain paths traversed by these explorations to the hopset, are performed offline.

We devise a technique to perform a given number of iterations of a BF exploration from a set of chosen vertices and up to a given distance in the graph in the dynamic streaming setting, and as in (23), perform the rest of the work offline. The difference however is that in the dynamic streaming setting, we do not perform an exact and deterministic BF exploration (as in (23)). A randomized algorithm for performing an approximate BF exploration originated at a subset of source vertices in a weighted graph, that succeeds whp, is described in Section 2.4. We use this algorithm as a subroutine in the superclustering step of our main algorithm.

The interconnection step is more challenging and involves performing multiple simultaneous BF explorations in a weighted graph, each from a separate source vertex. Here, each vertex in the graph needs to identify all the BF explorations it is a part of, and to find its (approximate) distance to the source of each such exploration. Due to the dynamic nature of the stream, a given vertex may find itself on a lot more explorations than it finally ends up belonging to. As shown in Section 3.1.2 in the context of near-additive spanner construction, this can be dealt with by combining a delicate encoding/decoding scheme for the IDs of exploration sources with a space-efficient sampling technique. We adapt here the technique used in Section 3.1.2 to work in weighted graphs.

In the following section, we provide an overview of our hopset construction algorithm.

4.1.1 Overview

Our hopset construction algorithm takes as input an n-vertex weighted undirected graph $G = (V, E, \omega)$, and parameters $0 < \varepsilon' < 1/10$, $\kappa = 1, 2, \ldots$ and $1/\kappa < \rho < 1/2$, and produces as output a $(1+\varepsilon', \beta')$-hopset of G. The hopbound parameter β' is a function of ε', Λ, κ, ρ and is given by

$$\beta' = O\left(\frac{\log\Lambda}{\varepsilon'}\cdot(\log\kappa\rho+1/\rho)\right)^{\log\kappa\rho+1/\rho} \tag{4.1}$$

Let $k = 0,1,\ldots,\lceil\log\Lambda\rceil-1$. Given two parameters $\varepsilon > 0$ and $\beta = 1,2,\ldots$, a set of weighted edges H_k on the vertex set V of the input graph is said to be a $(1+\varepsilon,\beta)$-*hopset for the scale* k or a *single-scale hopset*, if for every pair of vertices $u,v \in V$ with $d_G(u,v) \in (2^k, 2^{k+1}]$ we have that:

$$d_G(u,v) \leq d^{(\beta)}_{G_k}(u,v) \leq (1+\varepsilon)\cdot d_G(u,v),$$

where $G_k = (V, E\cup H_k, \omega_k)$ and $\omega_k(u,v) = min\{\omega(u,v), \omega_{H_k}(u,v)\}$, for every edge $(u,v) \in E\cup H_k$.

Let $\varepsilon > 0$ be a parameter that will be determined later in the sequel. Set also $\ell = \lfloor\log\kappa\rho\rfloor + \lceil\frac{\kappa+1}{\kappa\rho}\rceil - 1$. Let $\beta = (1/\varepsilon)^\ell$.

The algorithm constructs a separate $(1+\varepsilon,\beta)$-hopset H_k for every scale $(2^0,2^1],(2^1,2^2],\ldots,(2^{\lceil\log\Lambda\rceil-1}, 2^{\lceil\log\Lambda\rceil}]$ one after another. For $k \leq \lfloor\log\beta\rfloor - 1$, we set $H_k = \phi$. We can do so because for such a k, it holds that $2^{k+1} \leq \beta$, and for every pair of vertices u,v with $d_G(u,v) \leq 2^{k+1}$, the original graph G itself contains a shortest path between u and v that contains at most β edges. (We remark that after rescaling, we will have $\beta' = \beta$. See Section 4.1.3.) In other words, $d_G(u,v) = d^{(\beta)}_G(u,v)$. Denote $k_0 = \lfloor\log\beta\rfloor$ and $k_\lambda = \lceil\log\Lambda\rceil - 1$. We construct a hopset H_k for every $k \in [k_0, k_\lambda]$.

During the construction of the hopset H_k for some $k \geq k_0$, we need to perform explorations from certain vertices in V up to distance $\delta \leq 2^{k+1}$ in G. An exploration up to a given distance from a certain vertex in G may involve some paths with up to $n-1$ hops. This can take up to $O(n)$ passes through the stream. We overcome this problem by using the hopset edges $H^{(k-1)} = \bigcup_{k_0\leq j\leq k-1} H_j$ for constructing hopset H_k. The hopset H_k has to take care of all pairs of vertices u,v with $d_G(u,v) \in (2^k,2^{k+1}]$, whereas the edges in $E\cup H^{(k-1)}$ provide a $(1+\varepsilon_{k-1})$-approximate shortest path with up to β hops, for every pair u,v with $d_G(u,v) \leq 2^k$. The value of ε_{k-1} will be specified later in the sequel. Denote by $G^{(k-1)}$ the graph obtained by adding the edge set $H^{(k-1)}$ to the input graph G. Instead of conducting

explorations from a subset $S \subseteq V$ up to distance $\delta \leq 2^{k+1}$ in the input graph G, we perform $2\beta+1$ iterations of BF algorithm on the graph $G^{(k-1)}$ up to distance $(1+\varepsilon_{k-1})\cdot\delta$. The following lemma from (23) shows that $2\beta+1$ iterations of BF algorithm on $G^{(k-1)}$ up to distance $(1+\varepsilon_{k-1})\cdot\delta$ suffice to reach all the vertices within distance δ from set S in the original graph G. We refer the reader to Lemma 3.9 (and its preamble) of (23) for the proof.

Lemma 4.1. *(23) For $u,v \in V$ with $d_G(u,v) \leq 2^{k+1}$, the following holds:*

$$d^{(2\beta+1)}_{G^{(k-1)}}(u,v) \leq (1+\varepsilon_{k-1})\cdot d_G(u,v) \tag{4.2}$$

4.1.2 Constructing H_k

We now proceed to the construction of the hopset H_k for the scale $(2^k, 2^{k+1}]$, for some $k \in [k_0, k_\lambda]$. The algorithm is based on the superclustering and interconnection approach. The overall structure and technique of the construction of a single scale hopset is similar to that of the construction of a near-additive sparse spanner. (See Section 3.1.) The spanner construction algorithm of Section 3.1 works on an unweighted input graph and selects a subset of edges of the input graph as output. On the other hand, the hopset construction algorithm presented here works on a weighted input graph and produces as output a set of new weighted edges that need to be added to the input graph.

The algorithm starts by initializing the hopset H_k as an empty set. As in the construction of near-additive spanners (See Section 3.1.), the algorithm proceeds in phases $0,1,\ldots,\ell$. The maximum phase index ℓ is set as $\ell = \lfloor \log \kappa\rho \rfloor + \lceil \frac{\kappa+1}{\kappa\rho} \rceil - 1$. Throughout the algorithm, we build clusters of nearby vertices. The input to phase $i \in [0,\ell]$ is a set of clusters P_i, a distance threshold parameter δ_i and a degree parameter deg_i. For phase 0, the input P_0 is a partition of the vertex set V into singleton clusters. The definitions of the center r_C of a cluster C, its radius $Rad(C)$ and the radius of a partition $Rad(P_i)$ remain the same as in the case of spanner construction. (See Section 3.1 for more details.) Note, however that in the current context, the distances are in a weighted graph, $G^{(k-1)}$, rather than in the the unweighted input graph G, as it was the case in the construction of spanners.

The degree parameter deg_i follows the same sequence as in the construction of near-additive spanners. The set of phases $[0,\ell]$ is partitioned into two stages based on how the degree parameter changes from one phase to the next. (See Section 3.1 for more details.) The distance threshold parameter grows at the same steady rate (increases by a factor of $1/\varepsilon$) in every phase.

For clarity of presentation, we first define the sequence of the distance threshold parameters for hopset H_k as if all the explorations during the construction of H_k are exact and are performed on the input graph G (as in the centralised setting) itself. Then we modify this sequence to account for the fact that the explorations during the construction of H_k are actually conducted on the graph $G^{(k-1)}$ and not on the input graph G. The sequence of the distance threshold parameters for the centralized construction as defined in (23) is given by $\alpha = \alpha^{(k)} = \varepsilon^{\ell} \cdot 2^{k+1}$, $\delta_i = \alpha(1/\varepsilon)^i + 4R_i$, where $R_0 = 0$ and $R_{i+1} = R_i + \delta_i = \alpha(1/\varepsilon)^i + 5R_i$ for $i \geq 0$. Here α can be perceived as a unit of distance. To adjust for the fact that explorations are performed on the graph $G^{(k-1)}$, we multiply all the distance thresholds δ_i by a factor of $1+\varepsilon_{k-1}$, the stretch guarantee of the graph $G^{(k-1)}$. We further modify this sequence to account for the fact that our BF explorations (during superclustering as well as interconnection) in the dynamic stream are not exact and incur a multiplicative error. Throughout the construction of H_k, we set the multiplicative error of every approximate BF Exploration we perform to $1+\chi$, for a parameter $\chi > 0$ which will be determined later. Therefore we multiply all the distance thresholds by a factor of $1+\chi$. We define $R'_i = (1+\chi)\cdot(1+\varepsilon_{k-1})R_i$ and $\delta'_i = (1+\chi)\cdot(1+\varepsilon_{k-1})\delta_i$ for every $i \in [0,\ell]$. In the centralized setting, R_i serves as an upper bound on the radii of the input clusters of phase i. As a result of rescaling, R'_i becomes the new upper bound on the radii of input clusters of phase i.

All phases of our algorithm except for the last one consist of two steps, a superclustering step and an interconnection step. In the last phase, the superclustering step is skipped and we go directly to the interconnection step. The last phase is called the *concluding* phase.

The *superclustering* step of phase i randomly samples a set of clusters in P_i and builds larger clusters around them. The sampling probability for phase i is $1/deg_i$. In the insertion-only algorithm of (23), for every unsampled cluster center r'_C within distance δ_i (in G) from the set of sampled centers, an edge (r_C, r'_C) between r'_C and

a nearest sampled center r_C of weight $\omega_{H_k}(r_C, r'_C) = d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$ is added into the hopset H_k. In the dynamic stream, the distance exploration we do in $G^{(k-1)}$ is not exact and we have an estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$ which is stretched at most by a multiplicative factor of $1+\chi$. Hence in our algorithm, $\omega_{H_k}(r_C, r'_C) \leq (1+\chi) \cdot d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$. The collection of the new larger clusters $\hat{P}_i$ is passed on as input to phase $i+1$. In the *interconnection* step of phase i, the clusters that were not superclustered in this phase are connected to their nearby clusters. In the insertion-only algorithm of (23), $2\beta+1$ iterations of a BF exploration from the center r_C of every cluster in $U_i = P_i \setminus P_{i+1}$ are used to identify every other cluster in U_i whose center is within distance $\delta_i/2$ (in G) from r_C. For every center r'_C within distance $\delta_i/2$ (in G) from the center r_C of $C \in U_i$, an edge (r_C, r'_C) of weight $\omega_{H_k}(r_C, r'_C) = d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$ is added into the hopset H_k. In the dynamic stream, we do $2\beta+1$ iterations of a $(1+\chi)$-approximate BF exploration from every center. Therefore as in superclustering step, the weights of hopset edges added during interconnection step are stretched at most by a factor of $1+\chi$. In the concluding step ℓ, we skip the superclustering step. As was shown in (23), the input set of clusters to the last phase P_ℓ is sufficiently small to allow us to interconnect all the centers in P_ℓ to one another using few hopset edges.

We are now ready to describe in detail, the execution of superclustering step. The interconnection step will be described after that.

Superclustering: The phase i begins by sampling each cluster $C \in P_i$ independently at random with probability $1/deg_i$. Let S_i denote the set of sampled clusters. We now have to conduct (approximate) distance exploration up to depth δ'_i in $G^{(k-1)}$ rooted at the set $CS_i = \bigcup_{C \in S_i}\{r_C\}$. By Lemma 4.1, this can be achieved by $2\beta+1$ iterations of BF algorithm on the graph $G^{(k-1)}$. For this, we invoke the approximate BF exploration algorithm of Section 2.4 on graph $G^{(k-1)}$ with set CS_i as the set S of source vertices and parameters $\eta = 2\beta+1$, $\zeta = \chi$.

One issue with invoking the Algorithm of Section 2.4 as a blackbox for graph $G^{(k-1)}$ is that only the edges of the input graph G appear on the stream and the edge set $H^{(k-1)}$ of all the lower level hopsets is available offline. We therefore slightly modify the algorithm of Section 2.4 and then invoke the modified version with $S = CS_i$, $\eta = 2\beta+1$ and $\zeta = \chi$. In the modified version, at the end of each pass through the stream, for every vertex $v \in V$, we scan through the edges incident

to v in the set $H^{(k-1)}$ and update its distance estimate $\hat{d}(v)$ as:

$$\hat{d}(v) = \min\{\hat{d}(v), \min_{(v,w)\in H^{(k-1)}}\{\hat{d}(w) + \omega_{H^{(k-1)}}(v,w)\}\}.$$

The parent of v, $\hat{p}(v)$ is also updated accordingly. Note that this modification does not affect the space complexity, stretch guarantee or the success probability of the algorithm of Section 2.4. The upper bound on the stretch guarantee still applies since we update the distance estimate of a given vertex v only if the estimate provided by the edges in the set $H^{(k-1)}$ is better than v's estimate from the stream. The success probability and space complexity are unaffected since the modification deterministically updates the distance estimates and does not use any new variables. This provides us with a $(1+\chi)$-approximation of $d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i)$, for all $v \in V$.

Hence, by Theorem 2.22, an invocation of modified version of approximate BF algorithm of Section 2.4 during the the superclustering step of phase i generates whp, an approximate BF exploration of the graph $G^{(k-1)}$, rooted at the set $CS_i \subseteq V$ in $2\beta+1$ passes. It outputs for every $v \in V$ an estimate $\hat{d}(v)$ of its distance to set CS_i such that:

$$d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i) \leq \hat{d}(v) \leq (1+\chi)\cdot d^{(2\beta+1)}_{G^{(k-1)}}(v, CS_i). \tag{4.3}$$

Moreover, the set of parent variables $\hat{p}(v)$ of every $v \in V$ with $\hat{d}(v) < \infty$ span a forest F of $G^{(k-1)}$ rooted at the set of sampled centers CS_i. For every vertex v, one can compute its path to the root r_C of the tree in forest F, to which v belongs, through a chain of parent pointers. For every cluster center $r_{C'}$, $C' \in P_i \setminus S_i$, such that $\hat{d}(r_{C'}) \leq \delta'_i$, the algorithm adds an edge $(r_C, r_{C'})$ of weight $\hat{d}(r_{C'})$ to the hopset H_k, where r_C is the root of the tree in F to which $r_{C'}$ belongs. We also create a supercluster rooted at r_C which contains all the vertices of C' as above. Note that if $d_G(r_C, r_{C'}) \leq \delta_i$, then by equations (4.2) and (4.3), $\hat{d}(r_{C'}) \leq (1+\chi)\cdot(1+\varepsilon_{k-1})d_G(r_C, r_{C'}) = \delta'_i$. Therefore, the edge $(r_C, r_{C'})$ will be added in to the hopset and the cluster C' will be superclustered into a supercluster centered at r_C.

We conclude that:

Lemma 4.2. *For a given set of sampled cluster centers $CS_i \subseteq V$ and a sufficiently large constant c, the following holds with probability at least least $1 - 1/n^c$:*

1. *The superclustering step of phase i creates disjoint superclusters that contain all the clusters with centers within distance δ_i (in G) from the set of centers*

CS_i. It does so in $2\beta+1$ passes through the stream, using $O_c(\beta/\chi \cdot \log^2 n \cdot \log\Lambda(\log n + \log\Lambda))$ space.

2. *For every unsampled cluster center r_C within distance δ_i (in G) from the set CS_i, an edge to the nearest center $r'_C \in CS_i$ of weight $\omega_{H_k}(r_C, r'_C) \leq (1+\chi) \cdot d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C) \leq (1+\chi)\cdot(1+\varepsilon_{k-1}) d_G(r_C, r'_C)$ is added into the hopset H_k, where ε_{k-1} is the stretch guarantee of the graph $G^{(k-1)}$.*

Interconnection: Next we describe the interconnection step of each phase $i \in \{0,1,\ldots,\ell\}$. Recall that U_i is the set of clusters of P_i that were not superclustered in phase i. Let CU_i be the set of centers of clusters in U_i, i.e., $CU_i = \bigcup_{C\in U_i}\{r_C\}$. For the phase ℓ, the superclustering step is skipped and we set $U_\ell = P_\ell$.

In the interconnection step of phase $i \geq 0$, we want to connect every cluster $C \in U_i$ to every other cluster $C' \in U_i$ that is close to it. To do this, we want to perform $2\beta+1$ iterations of a $(1+\chi)$-approximate Bellman-Ford exploration from every cluster center $r_C \in CU_i$ *separately* in $G^{(k-1)}$. These explorations are, however, conducted to a bounded depth (in terms of number of hops), and to bounded distance. Specifically, the hop-depth of these explorations will be at most $2\beta+1$, while the distance to which they are conducted is roughly $\delta_i/2$. For every cluster center $r_{C'}$, $C' \in U_i$ within distance $\delta_i/2$ from r_C in G, we want to add an edge $e = (r_C, r_{C'})$ of weight at most $(1+\chi)\cdot d^{(2\beta+1)}_{G^{(k-1)}}(r_C, r'_C)$ to the hopset H_k. To do so, we turn to the stream to find an estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C)$ for every $v \in V$ and every center $r_C \in U_i$. As discussed in the construction of spanners, we cannot afford to invoke the algorithm of Section 2.4 multiple times in parallel to conduct a separate exploration from every center r_C in CU_i, due to space constraints. (See Section 3.1.2 for more details.) As shown in (23) (See Lemmas 3.2 and 3.3 of (23)), Lemma 3.2 holds in the interconnection step of (a single-scale) hopset construction as well. Specifically, if one conducts Bellman-Ford explorations to depth at most $\delta'_i/2$ in $G^{(k-1)}$ to hop-depth at most $2\beta+1$, then, with high probability, every vertex is traversed by at most $O(deg_i \ln n)$ explorations.

Therefore, we adapt the randomized technique of Section 3.1.2 to efficiently identify for every $v \in V$, the sources of all the explorations it gets visited by in phase i. Moreover, for every vertex $v \in V$ with a non-empty subset $U^v_i \subseteq U_i$ of explorations that visit v, we find for every cluster $C \in U^v_i$, an estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C)$. Note,

however, that not all the edges of the graph $G^{(k-1)}$ on which we have to perform our Bellman-Ford explorations are presented on the stream. We show in the sequel, how we adjust the distance estimates of every vertex $v \in V$ by going through the edges of the lower level hopsets $H^{(k-1)}$ offline.

Throughout the interconnection step of phase i, we maintain for every vertex $v \in V$, a set $LCurrent_v$ (called *estimates list* of v) of sources of Bellman-Ford explorations that visited v so far. Each element of $LCurrent_v$ is a tuple $(s, \hat{d}(v,s))$, where s is the center of some cluster in U_i, and $\hat{d}(v,s)$ is the current estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$. For any center $s' \in CU_i$, for which we do not yet have a tuple in $LCurrent_v$, $\hat{d}(v,s')$ is implicitly defined as ∞. Initially, the estimates lists of all the vertices are empty, except for the centers of clusters in U_i. The estimates list of every center $r_C \in CU_i$ is initialized with a single element $(r_C, 0)$ in it. The interconnection step of phase i is carried out in $2\beta+1$ sub-phases. In the following section, we describe the purpose of each of the $2\beta+1$ sub-phases of the interconnection step and the way they are carried out.

Sub-phase p of interconnection step: Denote $\zeta' = \frac{\chi}{2\cdot(2\beta+1)}$. Our goal is to ensure that by the end of sub-phase p, for every vertex $v \in V$ and every exploration source $s \in CU_i$ with a p-bounded path to v in $G^{(k-1)}$, there is a tuple $(s, \hat{d}(v,s))$ in the estimates list $LCurrent_v$ such that:

$$d^{(p)}_{G^{(k-1)}}(v,s) \leq \hat{d}(v,s) \leq (1+\zeta')^p \cdot d^{(p)}_{G^{(k-1)}}(v,s).$$

To accomplish this, in every sub-phase p, we search for every vertex $v \in V$, a *better* (smaller than the current value of $\hat{d}(v,s)$) estimate (if exists) of its $(2\beta+1)$-bounded distance to every source $s \in CU_i$, by keeping track of edges $e = (u,v)$ incident to v in $G^{(k-1)}$. In each of the $2\beta+1$ sub-phases, we make two passes through the stream. For a given vertex $v \in V$, an exploration source $s \in CU_i$ is called an *update candidate* of v in sub-phase p, if a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ is available in sub-phase p through some edge $e = (u,v)$ on the stream. (Recall that the current estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s')$ for some source $s' \in CU_i$ for which we do not yet have an entry in $LCurrent_v$ is ∞.) Note that a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$, for some vertex v and some source s in sub-phase p, may also be available through some edges in $H^{(k-1)}$. We therefore go through the edge set $H^{(k-1)}$ offline at the

end of every sub-phase and update all our estimates lists with the best available estimates in $H^{(k-1)}$.

In the first pass of sub-phase p, we identify for every $v \in V$, all of v's update candidates in sub-phase p. All of these update candidates are added to a list called the *update list* of v, denoted $LUpdate_v$. Each element of $LUpdate_v$ is a tuple $(s, range, r)$, where s is the ID of an exploration source in CU_i for which a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ is available, *range* is the distance range $I = (low, high]$ in which the better estimate is available, and r is the number of vertices $u \in \Gamma_G(v)$, such that $\hat{d}(u,s) + \omega(u,v) \in range$.

The second pass of sub-phase p uses the update list of every vertex $v \in V$ to find a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$, for every update candidate s in $LUpdate_v$. The new better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ for every source s in $LUpdate_v$ is then used to update the estimates list $LCurrent_v$ of v.

First pass of sub-phase p of phase i: By Lemma 3.2, the number of explorations that visit a vertex $v \in V$ during the interconnection step of phase i is at most d_i in expectation and at most $c'_1 \cdot \ln n \cdot deg_i$ whp, where c'_1 is a sufficiently large positive constant. Hence, the number of update candidates of v in any sub-phase of interconnection step of phase i is at most $c'_1 \cdot \ln n \cdot deg_i$ whp. (Recall that all the explorations are restricted to distance at most $\delta'_i/2$.)

As in Section 3.1.2, we denote $\mathcal{N}_i = c'_1 \cdot \ln n \cdot deg_i$ and $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$, where $c_4 \geq 1$ is a sufficiently large positive constant.

At a high level, in the first pass of every sub-phase, we want to recover, for every vertex $v \in V$, a vector (containing sources of explorations that visit v in sub-phase p) with at most $\mathcal{N}_i$ elements in its support. In other words, we want to perform an s-sparse recovery for every vertex $v \in V$, where $s = \mathcal{N}_i$. In the unweighted case in Section 3.1.2, we perform $\mathcal{N}_i$-sparse recovery for a given vertex v by multiple simultaneous invocations of a sampler *FindNewVisitor* that samples (with at least a constant probability) one exploration source out of at most $\mathcal{N}_i$ sources that visit v. In the weighted case, we do something similar but with a more involved sampling procedure called *FindNewCandidate*. The pseudocode for procedure *FindNewCandidate* is given in Algorithm 4. The procedure *FindNewCandidate* enables us

to sample an update candidate s of v (if exists), with a better (than the current) estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in a a specific distance range.

For every vertex $v \in V$, we divide the possible range of better estimates of v's $(2\beta+1)$-bounded distances to its update candidates, into sub-ranges on a geometric scale. We then invoke the procedure *FindNewCandidate* repeatedly in parallel to perform an $\mathcal{N}_i$-sparse recovery for v on every sub-range. Specifically, we divide the search space of potential better estimates, $[1,\delta'_i/2]$, into sub-ranges $I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$, for $j \in \{0,1,\ldots,\gamma\}$, where $\gamma = \lceil \log_{1+\zeta'} \delta'_i/2 \rceil - 1$. For $j=0$, we make the sub-range $I_0 = \left[(1+\zeta')^0,(1+\zeta')^1\right]$ closed to include the value 1. Note that we are only interested in distances at most $\delta'_i/2$. Therefore we restrict our search for distance estimates to the range $[1,\delta'_i/2]$, as opposed to the search range $[1,\Lambda]$ that we had in Section 2.4.1.

In more detail, we make for for each $v \in V$ and for each sub-range I_j, μ_i *attempts* in parallel. In a specific attempt for a given vertex v and a given sub-range I_j, we make a single call to procedure *FindNewCandidate* which samples an update candidate s (if exists) of v with a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in the sub-range I_j. Henceforth, we will refer to an update candidate s of a vertex v with a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in a given distance range I, as the *update candidate of v in the range I*.

The procedure *FindNewCandidate* can be viewed as an adaptation of procedure *FindNewVisitor* from Section 3.1.2 for weighted graphs. It takes as input the ID of a vertex, a hash function h chosen at random from a family of pairwise independent hash functions and an input range $I = (low, high]$. (The input range may be closed as well.) A successful invocation of *FindNewCandidate* for an input vertex v and a distance range I returns a tuple (s,c_s), where s is the ID of an update candidate of v in the range I, and c_s is the number of edges $(v,u) \in E$ such that $\hat{d}(u,s) + \omega(v,u) \in I$. If there is no update candidate of v in the input range I, procedure *FindNewCandidate* returns a tuple (ϕ,ϕ). If there are update candidates of v in the input range, but procedure *FindNewCandidate* fails to isolate an ID of such a candidate, it returns $(\perp,\perp)$.

Before we start making our attempts in parallel, we sample uniformly at random a set of functions H_p ($|H_p| = \mu_i$) from a family of pairwise independent hash

functions $h : \{1,\ldots,maxVID\} \rightarrow \{1,\ldots,2^{\lambda}\}$, where $\lambda = \lceil \log maxVID \rceil = \lceil \log n \rceil$. Then, for every vertex $v \in V$ and every distance sub-range I_j, $j \in \{0,1,\ldots,\gamma\}$, we make μ_i parallel calls to procedure $FindNewCandidate(v,h,I_j)$, one call for each $h \in H_p$.

Procedure FindNewCandidate: As mentioned above, the procedure *FindNewCandidate* is similar to procedure *FindNewVisitor* (See Algorithm 3) of Section 3.1.2. It uses a function h chosen uniformly at random from a family of pairwise independent hash functions to sample for the input vertex v, an update candidate of v in the input range I. Just like procedure *FindNewVisitor*, it also uses the CIS-based encoding scheme ν described in Section 2.2.4 to encode the names of the exploration sources it samples, and uses Lemma 2.12 to check (See line 21 of Algorithm 4), if it has successfully isolated the ID of a single update candidate in the desired distance range. We will mainly focus here on the details of Algorithm 4 which are different from that of Algorithm 3. We refer the reader to Sections 3.1.2 and 2.2.4 for a detailed exposition of our sampling technique and the CIS-based encoding scheme.

The procedure *FindNewCandidate* (Algorithm 4) differs from procedure *FindNewVisitor* (Algorithm 3) mainly in its input parameters and its handling of the incoming edges during the *Update Stage*. (See lines 6 to 18.) Specifically, procedure *FindNewCandidate* takes an additional input parameter I corresponding to a range of distances. It looks for an update candidate of input vertex v in the input range I. The update stage of a call to procedure *FindNewCandidate* for an input vertex v and an input distance range I proceeds as follows. For every update $(e_t,\ eSign_t,\ eWeight_t)$ to an edge e_t incident to v and some vertex u, we look at every exploration source s in the estimates list $LCurrent_u$ of u, (see line 8 of Algorithm 4) and check whether the distance estimate of v to s via edge $e_t = (v,u)$ is better than the current value of $\hat{d}(v,s)$, and whether it falls in the input distance range I. (See line 10 of Algorithm 4.) If this is the case, then, we sample s just like we sample new exploration sources in *FindNewVisitor*. This completes the description of procedure *FindNewCadidate*.

As in procedure *FindNewVisitor*, by Corollary 2.8, a single call to procedure *FindNewCandidate* succeeds with at least a constant probability.

For a vertex $v \in V$, if there are no update candidates of v in sub-phase p, all the calls to procedure *FindNewCandidate* in all the attempts return (ϕ, ϕ). For every such vertex, we do not need to add anything to its update list $LUpdate_v$. At the end of the first pass, if no invocation of procedure *FindNewCandidate* returns as error, we extract for every vertex $v \in V$ and every distance range I_j ($j \in \{0,1,\ldots,\gamma\}$), all the distinct update candidates of v in the range I_j sampled by μ_i attempts made for v and sub-range I_j. For a given update candidate s of v, let $j = j_{v,s}$ be the smallest index in $\{0,1,\ldots,\gamma\}$, such that a tuple (s,c_s) (for some $c_s > 0$) is returned by a call to procedure $FindNewCandidate(v,h,I_j)$. We add a tuple (s,I_j,c_s) to the list of update candidates $LUpdate_v$ of v. Recall that the set $LUpdate_v$ of vertex v contains tuples $(s,range,r_s)$, where s is the ID of an update candidate of v, *range* is the distance range in which a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ lies, and r is the number of edges $(u,v) \in \Gamma_G(v)$ such that $\hat{d}(u,s) + \omega(u,v) \in range$.

Analysis of first pass: We now analyze the success probability and space requirements of the first pass of sub-phase p of interconnection step. Recall that, in sub-phase p, for every vertex $v \in V$ and every distance sub-range
$I_j = \left((1+\zeta')^j,\ (1+\zeta')^{j+1}\right]$ ($j \in \{0,1,\ldots,\gamma\}$, where $\gamma = \lceil \log_{1+\zeta'} \delta'_i/2 \rceil - 1$), we make $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$ parallel attempts or calls to procedure *FindNewCandidate* to isolate all the update candidates of v in the range I_j.

We first show that making $\mu_i = 16 \cdot c_4 \cdot \mathcal{N}_i \cdot \ln n$ attempts in parallel for a given vertex $v \in V$ and a given distance range I_j, $j \in \{0,1,\ldots,\gamma\}$, ensures that a specific update candidate of vertex v in a specific distance range I in sub-phase p is extracted whp.

Lemma 4.3. *For a given vertex $v \in V$ and a specific distance sub-range I_j, during sub-phase p, a given update candidate s of v in the range I_j is discovered with probability at least $1 - 1/n^{c_4}$.*

Proof. Let $d_v^{(p,j)}$ be the number of update candidates of v in the range I_j in sub-phase p. By Lemma 3.2, with probability at least $1 - \frac{1}{n^{c'_1 - 1}}$, the number of the exploration sources that visit v during interconnection step of phase i is at most $\mathcal{N}_i$. Observe that $\mathcal{N}_i$ is an upper bound on the number of update candidates of v (over the entire distance range $[1, \delta'_i/2]$) during sub-phase p. It follows therefore that $d_v^{(p,j)} \leq \mathcal{N}_i$. For a specific update candidate s of v in the range I_j in sub-phase p, let $DISC^{(s)}$ denote the event that it is discovered in a specific attempt. Then:

$$\begin{aligned} Pr\left[DISC^{(s)}\right] &\geq Pr\left[DISC^{(s)} \mid d_v^{(p,j)} \leq \mathscr{N}_i\right] \cdot Pr\left[d_v^{(p,j)} \leq \mathscr{N}_i\right] \\ &\geq Pr\left[DISC^{(s)} \mid d_v^{(p,j)} \leq \mathscr{N}_i\right] \cdot \left(1 - \frac{1}{n^{c_1-1}}\right) \\ &\geq \frac{1}{8\mathscr{N}_i}\left(1 - \frac{1}{n^{c_1'-1}}\right) \\ &\geq \frac{1}{16\mathscr{N}_i} \end{aligned}$$

Note that the third inequality follows by applying Lemma 2.7 to the event $\{DISC^{(s)} \mid d_v^{(j)} \leq \mathscr{N}_i\}$.

Thus, for a given update candidate of v in the sub-range I_j, the probability that none of the $\mu_i = 16 \cdot c_4 \cdot \mathscr{N}_i \cdot \ln n$ attempts will isolate it is at most $\left(1 - \frac{1}{16\mathscr{N}_i}\right)^{16 \cdot c_4 \cdot \ln n \cdot \mathscr{N}_i} \leq 1/n^{c_4}$. □

Next, we analyze the space requirements of procedure *FindNewCandidate*. Procedure *FindNewCandidate* is similar to procedure *FindNewVisitor* of Section 3.1.2 in terms of its sampling technique. In addition to all the variables that procedure *FindNewVisitor* uses, procedure *FindNewCandidate* also uses distance variables *low* and *high*, that define the input range $I = (low, high]$, in which it looks for an update candidate of its input vertex. Each of these distance variables consume $O(\log \Lambda)$ bits. Adding the cost of additional variables used in procedure *FindNewCandidate* to the space usage of procedure *FindNewVisitor* (Lemma 3.4), we get the following lemma:

Lemma 4.4. *The procedure FindNewCandidate uses* $O(\log^2 n + \log \Lambda)$ *bits of memory.*

We next provide an upper bound on the space usage of the first pass of the interconnection step.

Lemma 4.5. *The overall space usage of the first pass of every sub-phase of interconnection is*

$$O(n^{1+\rho} \cdot \frac{\log \Lambda}{\zeta'} \cdot \log^2 n \cdot (\log^2 n + \log \Lambda)) \text{ bits.}$$

Proof. The first pass of every sub-phase makes $\gamma \cdot \mu_i = (\lceil \log_{1+\zeta'} \delta_i'/2 \rceil - 1) \cdot \mu_i = O(\log_{1+\zeta'} \Lambda \cdot deg_i \cdot \log^2 n)$ attempts in parallel for

every $v \in V$. Recall that for all i, $deg_i \leq n^\rho$ (See Section 3.1). Combining this fact with Lemma 4.4, we get that the space usage of all the invocations of procedure *FindNewCandidate* for all the n vertices during the first pass is $O(n^{1+\rho} \cdot \log_{1+\zeta'} \Lambda \cdot \log^2 n \cdot (\log^2 n + \log \Lambda))$. We use $|H_p| = \mu_i$ hash functions during the first pass. Each hash function can be encoded using $O(\log n)$ bits. The overall space used by the storage of hash functions during the first phase is thus $O(n^\rho \cdot \log^3 n)$. As an output, we produce an update list $LUpdate_v$ for every $v \in V$. Each of these update lists consists of tuples $(s, range, r)$ of $O(\log n + \log \lambda)$ bits each. By Lemma 3.2, a vertex v is visited by at most $O(deg_i \log n) \leq O(n^\rho \log n)$ explorations whp in the phase i. In any case, we record just $O(n^\rho \cdot \log n)$ of them, even if v is visited by more explorations. Hence, the storage of all the update lists during a given sub-phase requires $O(n^{1+\rho} \log n(\log n + \log \lambda))$ bits. Finally, we need to store the estimates lists $LCurrent_v$ of all $v \in V$. This requires at most $O(n^{1+\rho} \log n(\log n + \log \Lambda))$ bits of space. Thus, the storage cost of first pass of every sub-phase is dominated by the cost of parallel invocations of procedure *FindNewCandidate*. This makes the overall cost of first pass of every sub-phase

$$O(n^{1+\rho} \cdot \frac{\log \Lambda}{\zeta'} \cdot \log^2 n \cdot (\log^2 n + \log \Lambda)) \text{ bits.}$$

□

Second pass of sub-phase j of phase i: The second pass of sub-phase p starts with the update lists $LUpdate_v$ of every $v \in V$. Recall that the update list $LUpdate_v$ of a given vertex $v \in V$ consists of tuples of the form $(s, range, r)$, where s is an exploration source in CU_i for which a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ is available in the distance sub-range $range$, and r is the number of edges in the edge set E of the original graph G through which the better estimate is available. We find for every tuple $(s, range, r)$ in $LUpdate_v$, a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ in the sub-range $range$, by invoking procedure *GuessDistance* (described in Section 2.4.2) $O(\log n)$ times.

We sample uniformly at random a set of $c_1 \log_{7/8} n$ pairwise independent hash functions H'_p from the family $h : \{1, \ldots, maxVID\} \to \{1, 2, \ldots, 2^\lambda\}$ ($\lambda = \lceil \log n \rceil$), to be used by invocations of procedure *GuessDistance*.

We need to change the original procedure *GuessDistance* (Section 2.4.2) slightly to work here. Specifically, we need to change the part where we decide whether

to sample an incoming edge update or not (Line 5 of Algorithm 2). It should be updated to check if the edge e_t is incident between the input vertex v and some vertex u such that there is a tuple $(s, \hat{d}(u,s))$ in the estimates list of u and that $(\hat{d}(u,s) + eWeight_t) \in range$ and $\hat{d}(u,s) + eWeight_t < \hat{d}(v,s)$. Note that the current estimate $\hat{d}(v,s)$ of input vertex v's distance to its update candidate s is either available in its estimates list $LCurrent_v$ or is implicitly set to ∞. The latter happens if v has not yet been visited by the exploration rooted at source s.

At the end of the second pass, we have the results of all the invocations of procedure *GuessDistance*, for a given vertex v corresponding to the tuple $(s, range, r) \in LUpdate_v$. We update the corresponding tuple $(s, \hat{d}(v,s))$ in the estimates list $LCurrent_v$ of v with the minimum value returned by any invocation of *GuessDistance* for vertex v. If an entry corresponding to s is not present in the estimates list $LCurrent_v$ at this stage (i.e., $\hat{d}(v,s) = \infty$ as above), then we add a new tuple to the estimates list of v. Finally, the updates lists of all the vertices are cleared to be re-used in the next sub-phase. So far, we have only looked at the edges of the original graph presented to us in the stream while looking for better estimates of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$. Recall that we need to perform $2\beta+1$ iterations of the Bellman-Ford algorithm in the graph $G^{(k-1)}$ which is a union of the original graph G and $H^{(k-1)} = \bigcup_{\lfloor \log \beta \rfloor \leq j \leq k-1} H_j$ of all the lower level hopsets. Having updated the estimates lists of all the vertices with the best estimate available from the stream, at the end of second pass of sub-phase p we go through the edges of the lower level hopsets and check for each $v \in V$ whether a better estimate of $d^{(2\beta+1)}_{G^{(k-1)}}(v,s)$ for any source $s \in CU_i$ is available through one of the hopset edges. If this is the case, then we update the estimates lists accordingly.

Analysis of Second Pass: We now analyze the space requirements of the second pass of sub-phase j of interconnection step.

Lemma 4.6. *The overall space usage of the second pass of every sub-phase of the interconnection step is* $O(n^{1+\rho} \cdot \log^3 n \cdot (\log n + \log \Lambda))$.

Proof. The second pass of every sub-phase invokes procedure *GuessDistance* $O(\log n)$ times in parallel for every tuple in the update list $LUpdate_v$ of every $v \in V$. The number of elements in the update list $Lupdate_v$ of a given vertex v is at most $O(n^\rho \log n)$. (Recall that by Lemma 3.2, whp there are at most $\tilde{O}(n^\rho)$ explorations

per vertex. But even if there are more explorations, our algorithm records just $\tilde{O}(n^{\rho})$ explorations per vertex.) Therefore, we make a total of $O(n^{1+\rho} \cdot \log^2 n)$ calls to procedure *GuessDistance* during the second pass of any sub-phase. By Lemma 2.19, each invocation of procedure *GuessDistance* uses $O(\log n \cdot (\log n + \log \Lambda))$ bits of space. Therefore the overall cost of all the invocations of procedure *GuessDistance* is $O(n^{1+\rho} \cdot \log^3 n \cdot (\log n + \log \Lambda))$. In addition, we need to store a set of $O(\log n)$ hash functions of size $O(\log n)$ each in global storage. This requires $O(\log^2 n)$ bits of space. Therefore, the overall storage cost of the second pass of any sub-phase is dominated by the space required for invocations of *GuessDistance*. Hence the overall space requirement of second pass of interconnection is $O(n^{1+\rho} \cdot \log^3 n \cdot (\log n + \log \Lambda))$. □

Recall that $\zeta' = \frac{\chi}{2\cdot(2\beta+1)}$, $|H'_p| = c_1 \log_{8/7} n$ and $\mu_i = c_4 \cdot \ln n \cdot d_i$, where $c_1, c_4 > 0$ are positive constants.

Lemma 4.7. *For a sufficiently large constant c', with probability at least $1 - p/n^{c'-1}$, after p sub-phases of the interconnection step of phase i, the following holds for a given cluster $C \in U_i$ and for every vertex v within p hops from the center r_C of C in $G^{(k-1)}$:*

There is a tuple $(r_C, \hat{d}(v, r_C))$ in the estimates list $LCurrent_v$ of v such that

$$d^{(p)}_{G^{(k-1)}}(v, r_C) \leq \hat{d}(v, r_C) \leq (1+\zeta')^p \cdot d^{(p)}_{G^{(k-1)}}(v, r_C)$$

(The left-hand inequality holds with probability 1, and the right-hand inequality holds with probability at least $1 - p/n^{c'-1}$.)

Proof. The proof follows by induction on the number of phases, p, of the algorithm. The base case for $p = 0$ holds trivially. For the inductive step, we assume that the statement of the lemma holds for $p = t$, for some $0 \leq t < 2\beta + 1$, and prove it for $p = t + 1$. Let v be a vertex with a $(t+1)$-bounded shortest path to r_C in $G^{(k-1)}$. Denote by $u \in \Gamma_G(v)$, the neighbour of v on a shortest $(t+1)$-bounded path between v and r_C. By inductive hypothesis, with probability at least $1 - t/n^{c'-1}$, every vertex with a t-bounded shortest path to r_C has a tuple for r_C in its estimates list and the corresponding estimate provides a stretch at most $(1+\zeta')^t$. In particular, there is a tuple $(r_C, \hat{d}(u, r_C))$ in the estimates list $LCurrent_u$ of u such

that $d_G^{(t)}(u, r_C) \leq \hat{d}(u, r_C) \leq (1+\zeta')^t \cdot d_G^{(t)}(u, r_C)$. Denote by $j = j_v$ the index of a sub-range such that

$$\hat{d}(u, r_C) + \omega(u, v) \in I_j.$$

In the first pass of sub-phase $t+1$, we make μ_i attempts in parallel to identify all the update candidates of v in the distance range I_j. By Lemma 4.3, r_C will be sampled in one of the μ_i attempts, with probability at least at least $1 - 1/n^{c_4}$. In the second pass of sub-phase $t+1$, we make $O(\log n)$ calls to procedure *GuessDistance* to find an estimate of v's $(t+1)$-bounded distance to the center r_C in the sub-range I_j. By Lemma 2.18, with probability at least $1 - 1/n^{c_1}$, at least one of the calls to procedure *GuessDistance* will successfully return an estimate of $d_G^{(t+1)}(v, r_C)$ in the sub-range I_j. By a union bound over the failure probability of the first two passes for vertex v, we get that for an appropriate constant c', with probability at least, $1 - 1/n^{c'}$, vertex v will be able to find an estimate of $d_G^{(t+1)}(v, r_C)$ in the sub-range I_j. By union bound over all the vertices with a $(t+1)$-bounded shortest path to r_C, we get that with probability at least $1 - 1/n^{c'-1}$, all the vertices with a $(t+1)$-bounded shortest path to r_C will be able to find an estimate of their $(t+1)$-bounded distance to r_C in the appropriate sub-range. The overall failure probability of phase $t+1$ is therefore at most $1/n^{c'-1}$ plus $t/n^{c'-1}$ from the inductive hypothesis. In total, the failure probability is at most $\frac{t+1}{n^{c'-1}}$, as required. We assume henceforth that every vertex will successfully find an estimate of its $(t+1)$-bounded distance to r_C in the appropriate sub-range.

For a given vertex v, during the second pass of sub-phase $t+1$, we sample a candidate neighbour $u' \in \Gamma_G(v)$ such that $\hat{d}(u') + \omega(u', v) \in I_j$.

By induction hypothesis, vertex u has a tuple $(r_C, \hat{d}(u, r_C))$ in its estimates list such that, $\hat{d}(u, r_C) \leq (1+\zeta')^t \cdot d_G^{(t)}(u, r_C)$. Therefore,

$$\begin{aligned}
\hat{d}(u, r_C) + \omega(u, v) &\leq (1+\zeta')^t \cdot d_G^{(t)}(u, r_C) + \omega(u, v) \\
&\leq (1+\zeta')^t \cdot (d_G^{(t)}(u, r_C) + \omega(u, v)) \\
&= (1+\zeta')^t \cdot d_G^{(t+1)}(v, r_C).
\end{aligned}$$

Moreover, $(\hat{d}(u',r_C)+\omega(u',v))$ and $(\hat{d}(u,r_C)+\omega(u,v))$ belong to the same sub-range I_j, and thus,

$$\begin{aligned}\hat{d}(u',r_C)+\omega(u',v) &\leq (1+\zeta')\cdot(\hat{d}(u,r_C)+\omega(u,v))\\ &\leq (1+\zeta')^{t+1}\cdot d_G^{(t+1)}(v,r_C).\end{aligned}$$

Finally, any update made to $\hat{d}(v,r_C)$ offline at the end of the sub-phase $t+1$ does not increase the stretch, since we update $\hat{d}(v,r_C)$ only if there is a smaller estimate available through some edges in $H^{(k-1)}$.

For the lower bound, let $i \leq j$ be the minimum index such that we succeed in finding a neighbour u_i' of v with $(\hat{d}(u_i',r_C)+\omega(u_i',v)) \in I_i$. Then, with probability 1, $\hat{d}(u_i',r_C) \geq d_G^{(t)}(u_i',v)$ and thus,

$$\hat{d}(v,r_C) = \hat{d}(u_i',r_C)+\omega(u_i',v) \geq d_G^{(t)}(u_i',r_C)+\omega(u_i',v) \geq d_G^{(t+1)}(v,r_C).$$

□

Observe that Lemma 4.7 implies that for some $p \geq 1$, a single $(1+\chi)$-approximate Bellman-Ford exploration to hop-depth p, rooted at a specific center $r_C \in CU_i$ (conducted during the interconnection step of phase i) succeeds with probability at least $1-p/n^{c'-1}$. There are at most $d_i \leq n^\rho < n$ centers in CU_i. Taking a union bound over all the centers in CU_i, we get the following lemma:

Lemma 4.8. *For a sufficiently large constant c', with probability at least $1-p/n^{c'-2}$, after p sub-phases of the interconnection step of phase i, the following holds for any cluster $C \in U_i$ and for every vertex v within p hops from the center r_C of C in $G^{(k-1)}$:*

There is a tuple $(r_C,\hat{d}(v,r_C))$ in the estimates list $LCurrent_v$ of v such that

$$d_{G^{(k-1)}}^{(p)}(v,r_C) \leq \hat{d}(v,r_C) \leq (1+\zeta')^p \cdot d_{G^{(k-1)}}^{(p)}(v,r_C)$$

Recall that $\zeta' = \frac{\chi}{2\cdot(2\beta+1)}$. Invoking Lemma 4.8 with $p = 2\beta+1$ and $\zeta' = \frac{\chi}{2\cdot(2\beta+1)}$, implies the following corollary about the interconnection step of phase i:

Corollary 4.9. *For a sufficiently large constant c'', with probability at least $1-1/n^{c''}$, after $2\beta+1$ sub-phases of the interconnection step of phase i, the following*

holds for any cluster $C \in U_i$ and for every vertex v within $2\beta+1$ hops from the center r_C of C in $G^{(k-1)}$:

There is a tuple $(r_C, \hat{d}(v, r_C))$ in the estimates list $LCurrent_v$ of v such that

$$d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C) \leq \hat{d}(v, r_C) \leq (1+\chi) \cdot d^{(2\beta+1)}_{G^{(k-1)}}(v, r_C) \tag{4.4}$$

Finally, after $2\beta+1$ sub-phases of the interconnection step of phase i, we go through the estimates list of every center $r_C \in CU_i$ to check for every center $r'_C \in CU_i$, whether, there is a tuple $(r'_C, \hat{d}(r_C, r'_C)) \in LCurrent_{r_C}$ and $\hat{d}(r_C, r'_C) \leq \delta'_i/2$. Then, for every such center r'_C found, we add an edge (r_C, r'_C) of weight $\hat{d}(r_C, r'_C)$ into hopset H_k. Note that if $d_G(r_C, r_{C'}) \leq \delta_i/2$, then by equations (4.2) and (4.4), $\hat{d}(r_C, r_{C'}) \leq (1+\chi) \cdot (1+\varepsilon_{k-1}) d_G(r_C, r_{C'}) = \delta'_i/2$. Therefore, the edge $(r_C, r_{C'})$ will be added in to the hopset.

Lemmas 4.5, 4.6 and Corollary 4.9 together imply the following corollary about the interconnection step of phase i:

Lemma 4.10. *For a sufficiently large constant c'', after $2\beta+1$ sub-phases of the interconnection step of phase i during the construction of hopset H_k, $k \in [k_0, k_\lambda]$, the following holds with probability at least $1 - 1/n^{c''}$:*

1. *The interconnection step of phase i makes $2\beta+1$ passes through the stream, and the total required space is $O(\frac{\beta}{\chi} \cdot n^{1+\rho} \cdot \log\Lambda \cdot \log^2 n \cdot (\log^2 n + \log\Lambda))$ bits.*

2. *For every cluster $C \in U_i$ and every other cluster $C' \in U_i$ such that the center r'_C of C' is within distance $\delta_i/2$ in G from center r_C of C, an edge $(r_C, r_{C'})$ of weight at most $(1+\chi) \cdot (1+\varepsilon_{k-1}) \cdot d_G(r_C, r'_C)$ is added into hopset H_k, where ε_{k-1} is the stretch guarantee of the graph $G^{(k-1)}$.*

Lemmas 4.2 and 4.10 imply that our algorithm simulates phase i of insertion-only streaming algorithm (of (23)) for the construction of a single scale hopset H_k whp. Note, however, that the edges added to the hopset H_k by our algorithm during any phase i ($0 \leq i \leq \ell$), incur an extra stretch of $(1+\chi)$ compared to the insertion-only algorithm. The reason is that in the insertion-only algorithm, every pair of sufficiently close cluster centres are connected via an edge of weight *exactly equal* to the length of the path between them in $G^{(k-1)}$, while in our algorithm, the weight

of the connecting edge is a $(1+\chi)$-approximation of the length of the path between them in $G^{(k-1)}$.

The following lemma follows by induction on the number of phases of our algorithm.

Lemma 4.11. *After ℓ phases, our single-scale hopset construction algorithm simulates the insertion-only streaming algorithm of (23) for constructing a single-scale hopset H_k for scale $(2^k, 2^{k+1}]$, $k_0 \leq k \leq k_\lambda$, in the dynamic streaming setting whp such that*
any edge e added to the hopset H_k by our algorithm is stretched at most by a factor of $(1+\chi)$ compared to the insertion-only algorithm.

We return the edges of the set $H = \bigcup_{k_0 \leq j \leq k_\lambda} H_j$ as our final hopset.

Next, we analyze the properties of our final hopset H.

4.1.3 Putting Everything Together

Size: The size of our hopset H is the same as that of the insertion-only algorithm of (23), since we follow the same criteria (as in (23)), when deciding which cluster centres to connect via a hopset edge during our construction. Thus, the overall size of the hopset produced by our construction is $O(n^{1+1/\kappa} \cdot \log \Lambda)$ in expectation.
Stretch and Hopbound: Recall that ε_k is the value such that the graph $G^{(k)}$ (which is a graph obtained by adding the edges of hopset $H^{(k)} = \bigcup_{k_0 \leq j \leq k} H_j$ to the input graph G) provides stretch at most $1+\varepsilon_k$. Also, recall that $k_0 = \lfloor \log \beta \rfloor$ and $k_\lambda = \lceil \log \Lambda \rceil$.

Write $c_5 = 2$. We need the following lemma from (23) regarding the stretch of a single scale hopset H_k, $k \in [k_0, k_\lambda]$ produced by the insertion-only algorithm. We refer the reader to Lemma 3.10 and preamble of Theorem 3.11 of (23) for the proof. (Note that Lemma 3.10 and Theorem 3.11 of (23) are proved for the construction of a single scale hopset in the congested clique model. These also apply to their insertion-only construction.(See Section 3.5 of (23).))

Lemma 4.12. *(23) Let $x, y \in V$ be such that $2^k \leq d_G(x,y) \leq 2^{k+1}$, then it holds that*

$$d^{(h_\ell)}_{G \cup H_k}(x,y) \leq (1+\varepsilon_{k-1})(1+16 \cdot c_5 \cdot \ell \cdot \varepsilon) d_G(x,y), \tag{4.5}$$

and $h_\ell = O(\frac{1}{\varepsilon})^\ell$ *is the hopbound.*

Rescaling: Define $\varepsilon'' = 16 \cdot c_5 \cdot \ell \cdot \varepsilon$. Therefore, the stretch of a single scale hopset H_k, $k \in [k_0, k_\lambda]$, produced by the insertion-only algorithm of (23) becomes $(1+\varepsilon_{k-1})(1+\varepsilon'')$.

After rescaling, the hopbound h_ℓ becomes $O(\frac{\ell}{\varepsilon''})^\ell$. Recall that $\ell = \ell(\kappa,\rho) = \lfloor \log(\kappa\rho) \rfloor + \lceil \frac{\kappa+1}{\rho\kappa} \rceil - 1 \leq \log(\kappa\rho) + \lceil 1/\rho \rceil$, is the number of phases of our single-scale hopset construction. It follows that the hopbound of the insertion-only algorithm is

$$\beta_{EN} = O\left(\frac{\log \kappa\rho + 1/\rho}{\varepsilon''}\right)^{\log \kappa\rho + 1/\rho}. \tag{4.6}$$

Observe that for $k = k_0$, graph $G^{(k-1)}$ is the input graph G itself, since H_k for all $k < k_0$ is ϕ. (See Section 4.1.1 for details.) Therefore, $1+\varepsilon_{k-1}$ for $k = k_0$ is equal to 1. It follows therefore that
the stretch $1+\varepsilon_k = 1+\varepsilon_{k_{EN}}$, of the insertion-only algorithm follows the following sequence: $1+\varepsilon_{k_{0EN}} = (1+\varepsilon'')$ and for the higher scales, $1+\varepsilon_{k+1_{EN}} = (1+\varepsilon'') \cdot (1+\varepsilon_{k_{EN}})$.

By Lemma 4.11, the stretch of our single scale hopset construction (Section 4.1.2) for any scale $(2^k, 2^{k+1}]$, $k_0 \leq k \leq k_\lambda$ is $(1+\chi)$ times the stretch of the corresponding hopset produced by the insertion-only algorithm. We set $\chi = \varepsilon''$. Incorporating the additional stretch incurred by our algorithm into the stretch analysis of (23), we get the following lemma about the stretch of our dynamic streaming algorithm

Lemma 4.13. *For* $k \in [k_0, k_\lambda]$, *we have*

$$1+\varepsilon_{k_0} = (1+\varepsilon'')^2$$
$$1+\varepsilon_k = (1+\varepsilon'')^2(1+\varepsilon_{k-1}) \text{ for } k > k_0$$

Observe that Lemma 4.13 implies that the overall stretch of our hopset H is at most $(1+\varepsilon'')^{2\log\Lambda}$.
Recall that the desired stretch of our hopset construction is $1+\varepsilon'$ (see Section 4.1.1), where $\varepsilon' > 0$ is an input parameter of our algorithm.
We set $\varepsilon'' = \frac{\varepsilon'}{4\cdot\log\Lambda}$, and it follows that our overall stretch is

$$\left(1+\frac{\varepsilon'}{4\log\Lambda}\right)^{2\log\Lambda} \leq 1+\varepsilon'$$

.

Plugging in $\varepsilon'' = \frac{\varepsilon'}{4 \cdot \log \Lambda}$ in (4.6), we get the following expression for the hopbound of our dynamic streaming hopset:

$$\beta' = O\left(\frac{\log \Lambda}{\varepsilon'}(\log \kappa\rho + 1/\rho)\right)^{\log \kappa\rho + 1/\rho}. \tag{4.7}$$

(See also (4.1).)

Also recall that we had defined $\beta = (\frac{1}{\varepsilon})^{\ell}$ for using $2\beta + 1$ as the hop-depth of our explorations. After the two rescaling steps as above, we get that $\beta = \beta'$.

Next we analyze the pass complexity of our overall construction.

Lemma 4.14. *Our dynamic streaming algorithm makes $O(\beta' \log \Lambda \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream.*

Proof. In our single scale hopset construction (See Section 4.1.2), we make $O(\beta')$ passes during the superclustering step and $O(\beta')$ passes during the interconnection step of any phase. (Note that $\beta' = \beta$ and $\beta' = \beta'(\varepsilon, \kappa, \rho)$ is given by (4.7).) There are $\ell \leq \log(\kappa\rho) + \lceil 1/\rho \rceil$ phases in total. Thus, we make $O(\beta' \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream during the construction of a single scale hopset. We build at most $\log \Lambda$ single scale hopsets one after the other. Therefore, the overall pass complexity of our hopset construction is $O(\beta' \cdot \log \Lambda \cdot (\log \kappa\rho + 1/\rho))$. □

To summarize, we get the following equivalent of Theorem 3.16 of (23) summarizing our results:

Theorem 4.15. *For any n-vertex graph $G(V, E, \omega)$ with aspect ratio Λ, $2 \leq \kappa \leq (\log n)/4$, $1/\kappa \leq \rho \leq 1/2$ and $0 < \varepsilon' < 1$, our dynamic streaming algorithm computes a $(1+\varepsilon', \beta')$ hopset H with expected size $O(n^{1+1/\kappa} \cdot \log \Lambda)$ and the hopbound β' given by (4.7) whp.*
It does so by making $O(\beta' \cdot \log \Lambda \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream and using $O(\frac{\beta'}{\varepsilon'} \cdot n^{1+\rho} \cdot \log \Lambda \cdot \log^2 n \cdot (\log^2 n + \log \Lambda))$ bits of space.

4.1.4 Path-Reporting Hopsets

In certain applications of hopsets in the streaming model such as in the computation of approximate shortest paths, knowing all the hopset edges is not sufficient. For

every hopset edge $e = (u, v)$, one also needs to know the actual path $\pi(u, v)$ in G that implements e. We say that a hopset H is *path-reporting*, if for every hopset edge $(u, v) \in H$, there exists a path $\pi(u, v)$ between u and v in G such that $\omega_G(\pi(u, v)) = \omega_H(u, v)$ and H has enough enough information to compute $\pi(u, v)$. (See (23) for more information on path-reporting hopsets.) Recall that for a given vertex $v \in V$, the procedure *GuessDistance* executed as part of the $(2\beta' + 1)$-limited Bellman-Ford explorations conducted during our hopset construction gives us an estimate of v's distance to a source s. In addition, it also gives us the name of a neighbour u of v such that u is v's parent on the exploration tree rooted at s. When adding an edge (r_C, r'_C) between two nearby cluster centres to some hopset H_k, $k \in [k_0, k_\lambda]$, the path $\pi(r_C, r'_C)$ between r_C and r'_C in $G^{(k-1)}$ can be deduced from the parent pointers returned by calls to procedure *GuessDistance*. For every hopset edge e added to the hopset H_k for scale index $k \in [k_0, k_\lambda]$, we can store the corresponding path between its endpoints in $G^{(k-1)}$ to the hopset. Note that some of the edges on this path may themselves by hopset edges from a lower level hopset. This is not a problem, since the path corresponding to such an edge e' should have been stored during the construction of the hopset in which e' was added. Since all the explorations conducted during the construction of H_k are $(2\beta' + 1)$-limited, it increases the space usage of the hopset construction by a factor of $O(\beta')$.

To summarize, we get the following equivalent of Theorem 4.15 for path-reporting hopsets:

Theorem 4.16. *For any n-vertex graph $G(V, E, \omega)$ with aspect ratio Λ, $2 \leq \kappa \leq (\log n)/4$, $1/\kappa \leq \rho \leq 1/2$ and $0 < \varepsilon' < 1$, our dynamic streaming algorithm computes a $(1+\varepsilon', \beta')$ hopset H with expected size $O(n^{1+1/\kappa} \cdot \log \Lambda)$ and the hopbound β' given by (4.7) whp.*
It does so by making $O(\beta' \cdot \log \Lambda \cdot (\log \kappa\rho + 1/\rho))$ passes through the stream and using $O(\frac{\beta'^2}{\varepsilon'} \cdot n^{1+\rho} \cdot \log \Lambda \cdot \log^2 n \cdot (\log^2 n + \log \Lambda))$ bits of space.

4.2 Eliminating Dependence on the Aspect Ratio

In this section, we devise a reduction that eliminates the dependence on the aspect ratio Λ of the graph from the hopbound, size and the overall pass complexity of our hopset construction. We note, however, that the space complexity of our algorithm

is still linear in $\log\Lambda$. The result of applying the reduction to our algorithm is summarized in Theorem 4.20.

4.2.1 Overview

In this sub-section, we give a general overview of the reduction for a weighted input graph $G = (V, E, \omega)$, and then describe its implementation in the dynamic streaming model in the subsequent sub-section. Fix a parameter $0 < \varepsilon < 1/2$. Recall that $k_0 = \lfloor \log \beta \rfloor$ and $k_\lambda = \lceil \log \Lambda \rceil - 1$. For every scale index $k \in [k_0, k_\lambda]$, we build a graph $\mathscr{G}_k$ that contains edges with weights in the range $((\varepsilon/n) \cdot 2^k, (1+\varepsilon/2) \cdot 2^{k+1}]$. This graph can be constructed by deleting the *heavy* edges (of weight more than 2^{k+1}) and contracting the *light* edges (of weight less than $(\varepsilon/n) \cdot 2^k$) from G. By contraction, we mean grouping vertices with light edges between them into supervertices which we call *nodes*. Each node of graph $\mathscr{G}_k$ is a subset of V. Observe that the nodes of $\mathscr{G}_k$ are connected components of the graph obtained by deleting all the edges of weight more that $(\varepsilon/n) \cdot 2^k$ from G. The weight of an edge $(X, Y) \in E(\mathscr{G}_k)$ is set to be

$$\mathscr{W}(X, Y) = \omega(x, y) + (\varepsilon/n) \cdot 2^k \cdot (|X| + |Y|), \tag{4.8}$$

where $x \in X$ and $y \in Y$ and edge $(x, y) \in E$ is the shortest edge between a vertex of X and a vertex of Y.

Observe that the minimal weight of an edge in $\mathscr{G}_k$ is at least $(\varepsilon/n) \cdot 2^k + 2(\varepsilon/n) \cdot 2^k > (\varepsilon/n) \cdot 2^{k+1}$ and the maximal weight is at most $2^{k+1} + \varepsilon \cdot 2^k = (1+\varepsilon/2) \cdot 2^{k+1}$. Therefore, for every scale index $k \in [k_0, k_\lambda]$, the corresponding graph $\mathscr{G}_k$ has aspect ratio $O(n/\varepsilon)$.

Every node U in $\mathscr{G}_k$ is assigned a designated center u and we add edges from u to every other vertex in U. These edges are called the *star* edges. Let S_k denote the set of all the star edges of $\mathscr{G}_k$. Consider a contraction of an edge (x', y'), $x' \in X$, $y' \in Y$, connecting nodes X, Y with centers $x^* \in X$ and $y^* \in Y$. Assuming that $|X| \geq |Y|$, the vertex x^* is declared the center of $U = X \cup Y$, and we add to S_k edges from x^* to every vertex of Y. The weight of the edge (x, z) for each $z \in Y$ is set as

$$\mathscr{W}(x, z) = (\varepsilon/n) \cdot 2^k \cdot |U|. \tag{4.9}$$

Observe that the weight of the star edge (x,z) dominates the value of $d_G(x,z)$, since there exists a path between x and z in G consisting of at most $|U|-1$ edges each of weight at most $\varepsilon/n \cdot 2^k$.

For every $k \in [k_0, k_\lambda]$, a separate single-scale $(1+\varepsilon,\beta)$-hopset $\mathscr{H}_k$ for the scale $(2^k, 2^{k+1}]$ is computed on the graph $\mathscr{G}_k$. This is done in parallel for all $k \in [k_0, k_\lambda]$. The ultimate hopset $\mathscr{H}$ is computed as follows. For every scale index $k \in [k_0, k_\lambda]$, and for every edge (X,Y) of weight d in the hopset $\mathscr{H}_k$, a corresponding edge (x^*, y^*) of weight d between the centers x^* and y^* of X and Y respectively is added to the final hopset $\mathscr{H}$. To ensure that the number of hops within each node is also small, we also add to $\mathscr{H}$ the set $S = \bigcup_{k\in[k_0,k_\lambda]} S_k$ of all the star edges.

Analysis

In this section, we analyze the properties of the graphs $\mathscr{G}_k$ for $k \in [k_0, k_\lambda]$ and of the corresponding hopset $S_k \cup \mathscr{H}_k$. Let $\mathscr{V}_k$ be the set of nodes of $\mathscr{G}_k$. For a node $U \in \mathscr{V}_k$, let $S(U)$ denote the set of star edges of the node U, i.e., $S(U) = \{(x,y) \in S | x,y \in U\}$.

Lemma 4.17. $|S| \leq n \log n$.

Proof. The proof follows by induction on the scale index k. For $k = k_0$, a node $U \in \mathscr{V}_{k_0}$ contains $|U| - 1 \leq |U| \cdot \log |U|$ edges.

We assume that the claim holds for some scale index $k \in [k_0+1, k_\lambda]$ and prove that it also holds for $k+1$. Let U be a node in $\mathscr{V}_{k+1}$ and $X_1, X_2, \ldots, X_t$ be nodes in $\mathscr{V}_k$ such that $U = \bigcup_{j\in[1,t]} X_j$ and $|X_1| \geq |X_2| \geq \ldots \geq |X_t|$. Let s denote the size of U and s_j denote the size of the node X_j for $j \in [1,t]$. By induction hypothesis, X_j (for $j \in [1,t]$) contains at most $s_j \cdot \log s_j$ star edges. When we merge $X_1, X_2, \ldots X_t$ to form U, the center x_1 of X_1 becomes the center of U and we add edges from x_1

to all the vertices in $\bigcup_{j\in[2,t]} X_j$. It follows that

$$\begin{aligned}
s &\leq \sum_{j\in[2,t]} s_j + \sum_{j\in[1,t]} s_j \cdot \log s_j \\
&= s_1 \log s_1 + \sum_{j\in[2,t]} s_j(1+\log s_j) = s_1 \log s_1 + \sum_{j\in[2,t]} s_j \log(2s_j) \\
&\leq s_1 \log(s_1+s_2) + \sum_{j\in[2,t]} s_j \log(s_1+s_j) \leq s_1 \log(s_1+s_2) + \sum_{j\in[2,t]} s_j \log(s_1+s_2) \\
&= \log(s_1+s_2) \cdot \sum_{j\in[1,t]} s_j = s\log(s_1+s_2) \leq s\log s.
\end{aligned}$$

Observe that at a certain point when the scale index is sufficiently large, we have a graph with a single node containing all the vertices of the input graph. At this point, we have added at most $n\log n$ star edges to the hopset $\mathscr{H}$. □

Relevant Scales: Recall that we build a separate hopset $\mathscr{H}_k$ for every scale index $k \in [k_0, k_\lambda]$. Specifically, hopset $\mathscr{H}_k$ is used to approximate distances in the range $(2^k, 2^{k+1}]$. If no edge in G has weight in the range $(2^k/n, 2^{k+1}]$, then there is no pair of vertices in V with distance in the range $(2^k, 2^{k+1}]$. In this case, the hopset $\mathscr{H}_k$ is *redundant*. We call a scale index $k \in [k_0, k_\lambda]$ *relevant* if there exists an edge $(u,v) \in E$ such that $\omega(u,v) \in (2^k/n, 2^{k+1}]$, and *redundant* otherwise. Let K be the set of relevant scale indices from $[k_0, k_\lambda]$. Observe that a given edge can induce at most $\log n$ relevant scales and hence $|K| = O(|E| \cdot \log n)$. We construct a hopset $\mathscr{H}_k$ only for a graph $\mathscr{G}_k$ with $k \in K$.

Active Nodes: The nodes of the graphs $\{\mathscr{G}_k\}_{k\in K}$ induce a laminar family $\mathscr{L}$ on V which contains at most $2n-1$ distinct sets. We say that a node U in the graph $\mathscr{G}_k$ is *active* if it has degree at least 1. Denote by n_k the number of active nodes in $\mathscr{G}_k$. By arguments similar to those used in (23; 15), one can bound the number of active nodes in graphs $\{\mathscr{G}_k\}_{k\in K}$ and get that

$$\sum_{k\in K} n_k = O(n\log n) \tag{4.10}$$

Hopset Size: Recall that for every hopset edge (X,Y) of a single-scale hopset $\mathscr{H}_k$ of weight d, we add to the ultimate hopset $\mathscr{H}$, an edge of the same weight between the centers of X and Y. In addition, we add all the star edges of hopsets $\{\mathscr{H}_k\}_{k\in K}$ to $\mathscr{H}$. Also, recall from Section 4.1.3 that the expected size of a single-scale hopset $\mathscr{H}_k$ on a graph on n vertices is $O(n^{1+1/\kappa})$. For every $k \in [k_0, k_\lambda]$, only active nodes

of $\mathscr{G}_k$ take part in the construction of hopset $\mathscr{H}_k$. Therefore, Lemma 4.17 and equation (4.10) together imply that

$$\begin{aligned} |\mathscr{H}| = |S| + \sum_{k\in K} |\mathscr{H}_k| &\leq n\log n + \sum_{k\in K} O(n_k^{1+1/\kappa}) \\ &\leq n\log n + n^{1/\kappa} \sum_{k\in K} O(n_k) = O(n^{1+1/\kappa}\log n) \end{aligned} \tag{4.11}$$

4.2.2 Implementation in Dynamic Streaming model

The algorithm proceeds in two phases. In the first phase, we make one pass through the stream and compute all the nodes of $\{\mathscr{G}_k\}_{k\in K}$. In the second phase, we compute in parallel a single-scale hopset $\mathscr{H}_k$ for every $k \in K$. The details of the two phases are provided below.

Recall that each node U of $\{\mathscr{G}_k\}_{k\in K}$ is a subset of the vertex set V of G and has a designated center u^*. Whenever we contract an edge between nodes X and Y with $|X| \geq |Y|$, the vertices of Y get a new center but the size of the node containing them is at least doubled. This implies that each vertex changes the center of the node containing it at most $\log n$ times. For every vertex $v \in V$, we store a *list* $L(v)$ of pairs (i,c). Each pair $(i,c^*) \in L(v)$ indicates that at scale $i \in K$, the node containing v was merged into a larger node centred at c^*. Initially, $L(v)$ for every $v \in V$ is empty. The set $Lists = \{L(v) | v \in V\}$ encodes the laminar family $\mathscr{L}$ induced by the nodes of $\{\mathscr{G}_k\}_{k\in K}$. The description of first phase up to this point is similar to that of the insertion-only streaming implementation of (23). (See Section 4.2 of (23) for more details.) In (23), the *Lists* data structure and the set S of star edges are recomputed upon arrival of every edge on the stream. In the dynamic streaming setting, this approach is not applicable, because a removal of an edge $e = (x,y)$ from the stream may cause a split of an existing node U that contains both x and y. On the other hand, if there is an alternative $x - y$ path in $G(U)$, the node will stay intact. To be able to distinguish between these two scenarios, one needs to take a different approach.

For each $v \in V$, we update a data structure $XORSlots(v)$ (described in the sequel) upon arrival of edge updates, and later use these data structures to compute *Lists* and star edges offline. The data structure $XORSlots(v)$ for vertex v consists of

$\lambda+1 = O(\log \Lambda)$ arrays, $Slots(v,k)$ for $k \in [k_0, k_\lambda]$. Each $Slots(v,k)$ array is similar to the *Slots* array used in procedure *GuessDistance*. (See Section 2.4.2 for more details.) It enables us to sample from the edges incident on v, an edge with weight in the range $(\varepsilon/n) \cdot (2^{k-1}, 2^k]$ for $k \in [k_0+1, k_\lambda]$ and $(0, (\varepsilon/n) \cdot 2^k]$ for $k = k_0$. As in *Slots* array (Section 2.4.2), elements of $Slots(v,k)$ array are used to sample certain edges incident on v on a range of probabilities and we use a function chosen uniformly at random from a family of pairwise independent hash functions for sampling. Note that we use the same hash function for computing the $Slots(v,k)$ arrays for every $v \in V$ and every $k \in K$. Specifically, the element at index i of $Slots(v,k)$ maintains the bitwise XOR of the binary names of the edges sampled with probability corresponding to the index i. For a given edge $e = (u,v)$, its binary name is a concatenation of the binary representation of the *IDs* of its endpoints, with the smaller of the two *IDs* appearing first. Upon arrival of an update to some edge (u,v) on the stream with weight $\omega(u,v) \in (\varepsilon/n) \cdot (2^{k-1}, 2^k]$ for some $k \in [k_0, k_\lambda]$, the slots array of both u and v for the scale k are updated. Note that for some $v \in V$ and $k \in [k_0, k_\lambda]$, the slots array $Slots(v,k)$ enables us to sample with constant probability (See Section 2.4.2.) an edge $e = (u,v) \in E$ with $\omega(u,v) \in (\varepsilon/n) \cdot (2^{k-1}, 2^k]$, if such an edge exists. We maintain $O(\log n)$ copies of $Slots(v,k)$ for each $k \in [k_0, k_\lambda]$ in our data structure $XORSlots(v)$ to increase the success probability of our sampling procedure. Each of these $O(\log n)$ copies is generated using a different hash function.

For a set $U \subseteq V$, an edge $e = (u,v)$ is called an *outgoing edge* of U, if $u \in U$ and $v \notin U$. An important property of the $XORSlots(v)$ data structures is that for a given set $U = \{u_1, u_2, \ldots, u_m\} \subseteq V$, and a scale index $k \in [k_0, k_\lambda]$, we can build an array $Slots(U,k)$ (similar to the $Slots(v,k)$ array for a single vertex v) from $\{Slots(u_j,k)\}_{j \in [1,m]}$, provided all these arrays are generated using the same hash function. In sketching literature, this property is called linearity of sketches. (See (6; 17) for more details.) The *Slots* array $Slots(U,k)$ for set U enables us to sample an outgoing edge of U with weight in the range $(\varepsilon/n) \cdot (2^{k-1}, 2^k]$. The value of the element at index i of $Slots(U,k)$ is given by

$$Slots(U,k)[i] = Slots(u_1,k)[i] \bigoplus Slots(u_2,k)[i] \bigoplus \ldots \bigoplus Slots(u_m,k)[i].$$

Recall that $\bigoplus$ stands for bitwise XOR and all the *Slots* arrays in $\{Slots(u_j,k)\}_{j\in[1,m]}$ are generated using the same hash function. A given edge $e=(x,y)$ with $\omega(x,y)\in(\varepsilon/n)\cdot(2^{k-1},2^k]$ will be added to the same index elements in both $Slots(x,k)$ and $Slots(y,k)$. This ensures that every element of $Slots(U,k)$ contains a bitwise XOR of *only* the outgoing edges of U. As with the *Slots* arrays for individual vertices, for some appropriate index i, the element $Slots(U,k)[i]$ will contain only one edge with at least a constant probability. If we fail to sample an outgoing edge for U, we use a different set of *Slots* arrays generated using a different hash function. Recall that for each vertex $v\in V$ and $k\in K$, we maintain $O(\log n)$ copies of $Slots(v,k)$, each generated using a different hash function.

After the first pass, we go offline to compute the nodes of graphs $\{\mathscr{G}_k\}_{k\in K}$. We compute the nodes of graphs $\mathscr{G}_k$ for $k=k_0,k_0+1,k_0+2,\ldots$, sequentially, in that order. In the following, we describe a procedure called *ComputeCC* which computes the nodes of $\mathscr{G}_k$ from the nodes of $\{\mathscr{G}_j\}_{j<k}$. For $k=k_0$, the procedure *ComputeCC* computes the nodes of $\mathscr{G}_{k_0}$ from the vertex set V of the input graph G. We make $O(\log n)$ iterations. Each iteration starts with a set CC consisting of all the connected components of $\mathscr{G}_k$ identified so far. Note that each connected component $C\in CC$ is a subset of the vertex set V. Initially, set CC contains all the nodes of $\{\mathscr{G}_j\}_{j<k}$. For $k=k_0$, CC contains singleton sets $\{v\}$, for every $v\in V$. In every iteration, for every component $C\in CC$, we find an outgoing edge with weight in the range $(\varepsilon/n)\cdot(2^{k-1},2^k]$, for $k>k_0$ and $(0,(\varepsilon/n)\cdot 2^{k_0}]$, for $k=k_0$ (if there exists such an edge). For this, we can compute (offline) and use the *Slots* array of every component, as described in the last paragraph. For a component $C_1\in CC$ with an outgoing edge in the appropriate weight range to another component $C_2\in CC$, we merge the two components, and add $C_1\cup C_2$ to CC, and remove C_1 and C_2 from CC. We stop when none of the connected components in CC have an outgoing edge with weight in the appropriate range. Note that each of these connected components corresponds to a node of $\mathscr{G}_k$. For computing the nodes of $\mathscr{G}_k$ from those of $\{\mathscr{G}_j\}_{j<k}$, we only need to consider edges $e=(x,y)$ with $\omega(x,y)\in(\varepsilon/n)\cdot(2^{k-1},2^k]$ such that $x\in X$ and $y\notin X$, for every node X. For every such node X, we merge X with the node Y containing y. Note that while merging the nodes X and Y, the edge (x,y) that causes the merge and its exact weight do not matter as long as such an edge exists. Therefore, we do not need to consider every edge incident on every

vertex of a node X as long as we can find an outgoing edge (if exists) with weight in the range $(\varepsilon/n)\cdot(2^{k-1},2^k]$. This can be easily done by considering only the *Slots* array $Slots(X,k)$, which can be computed offline from the *Slots* arrays of the vertices contained in X.

At the end of the execution of procedure *ComputeCC*, we assign centers to the newly formed nodes and add star edges to the set S as follows. For $k=k_0$, we assign an arbitrary vertex of each node as its center and add its star edges with weight given by equation (4.9) to the set S. We also add a pair (k_0,u^*) to the list $L(v)$ of every vertex v in the node centred at u^*. For $k>k_0$, let $U\in CC$ be a node of $\mathscr{G}_k$ and let U be formed by merging nodes $X_1,X_2,\ldots,X_t$ (each from $\{\mathscr{G}_j\}_{j<k}$). Let further X_1 be the largest node among $X_1,X_2,\ldots,X_t$, we assign the center x^* of X_1 as the center of new node U. We update the list $L(x)$ for every $x\in U\setminus X_1$ with a pair (k,x^*), and add a star edge (x^*,x) with weight given by equation (4.9) to the set S.

Having computed the nodes of graphs $\{\mathscr{G}_k\}_{k\in K}$, we turn to the stream again to compute our hopsets. We compute a separate hopset $\mathscr{H}_k$ for all $k\in K$ in parallel. For each $k\in K$, we run our hopset construction algorithm from Section 4.1.2. Initially, the vertices of $\mathscr{G}_k$ can be derived from the *Lists* data structure that we populated after the first pass. We identify each vertex of $\mathscr{G}_k$ by its center. Whenever some update to an edge (x,y) of weight $\omega(x,y)$ is read from the stream, we know that it is active in at most $O(\log n/\varepsilon)$ scales. For each scale $k\in K$ such that $(\varepsilon/n)\cdot 2^k\le\omega(x,y)\le 2^{k+1}$, we use the *Lists* data structure to find the centres of nodes containing x,y in $\mathscr{G}_k$, and execute the algorithm from Section 4.1.2 as if an edge between these centres (of weight given by (4.8)) was just read from the stream. Recall that the construction of a single-scale hopset involves performing approximate Bellman-Ford explorations of the input graph from a set of starting vertices. In our approximate Bellman-Ford exploration algorithm (Section 2.4), we use a procedure *GuessDistance* (Section 2.4.2) to estimate the distance of a vertex to the set of starting vertices. We need to slightly tweak this procedure to take care of the fact that the graph $\mathscr{G}_k$ is a multigraph. We can do this by maintaining a running sum of the CIS-based encoding (see Section 2.2.4 for more details) of the sampled edges instead of bitwise XOR of their binary names in the proce-

dure *GuessDistance*. Note that this change does not affect the space usage of the procedure *GuessDistance*.

Elkin and Neiman (23) provide a detailed stretch analysis of the reduction in the centralized model which also applies to their insertion-only implementation. (See Section 4 of (23).) In particular, they show that the ultimate hopset $\mathscr{H}$ produced by this reduction is a $(1+6\varepsilon, 6\beta+5)$-hopset of the input graph G. See Lemma 4.3 of (23). By Lemma 4.11, every hopset edge of our single-scale hopset $\mathscr{H}_k$ ($k \in K$) construction is stretched at most by a factor of $(1+\chi)$ compared to the insertion-only algorithm of (23). We set $\chi = 6\varepsilon$. It follows that the stretch of our dynamic streaming construction is $(1+6\varepsilon)^2$. Let $\varepsilon' > 0$ be the desired stretch of our overall construction. We set $6\varepsilon = \varepsilon'/4$. Therefore, the overall stretch of our ultimate hopset is

$$(1+\varepsilon'/4)^2 \leq 1+\varepsilon'.$$

Next we analyze the space usage of the reduction. The following lemma summarizes the space requirement of the first phase.

Lemma 4.18. *The first phase of our dynamic streaming reduction requires* $O(n \cdot \log^3 n \cdot \log \Lambda)$ *bits of memory.*

Proof. The memory usage of the first phase has two main components, the space required to maintain $XORSlots(v)$ data structures, and the space required to store the $O(\log n)$ hash functions of $O(\log n)$ bits each. The storage of $XORSlots(v)$ data structures involves storing for every vertex $v \in V$, $O(\log n)$ copies of $O(\log \Lambda)$ *Slots* arrays, each of size $O(\log^2 n)$ bits. The total space required is therefore $O(n \cdot \log^3 n \cdot \log \Lambda)$ bits. □

As an output of the first phase, we produce the *Lists* data structure which is used throughout the second phase of the reduction. The *Lists* data structure requires $O(n \cdot \log^2 n)$ bits of memory. By equation (4.11), the size of hopset $\mathscr{H}$ is $O(n^{1+1/\kappa} \cdot \log n)$ which implies that the space required to store the hopset edges is $O(n^{1+1/\kappa} \cdot \log^2 n)$ bits. In the second phase, we invoke our dynamic streaming algorithm from Section 4.1.2 to construct all the relevant hopsets in parallel. Lemmas 4.2 and 4.10 summarize the resource requirements of the two main steps of the algorithm from Section 4.1.2. Using the fact that every graph in $\{\mathscr{G}_k\}_{k \in K}$ has aspect ratio $O(n/\varepsilon)$,

we can essentially replace $\log\Lambda$ by $\log(n/\varepsilon)$ in the space requirements in Lemmas 4.2 and 4.10. We get the following lemma summarizing the space requirement of the second phase of our reduction

Lemma 4.19. *The second phase of our dynamic streaming reduction requires* $O(\frac{\beta'}{\varepsilon'}\cdot\log^2 1/\varepsilon'\cdot n^{1+\rho}\cdot\log^5 n)$ *bits of space.*

The first phase of our reduction requires only one pass through the stream. Since we build all the relevant hopsets in parallel, we can drop the $\log\Lambda$ factor from the pass complexity of hopset construction given by Lemma 4.14.

Formally, we get the following equivalent of Theorem 4.15.

Theorem 4.20. *For any n-vertex graph $G(V,E,\omega)$ with aspect ratio Λ, $2\le\kappa\le(\log n)/4$, $1/\kappa\le\rho\le 1/2$ and $0<\varepsilon'<1$, our dynamic streaming algorithm computes a $(1+\varepsilon',\beta')$ hopset H with expected size $O(n^{1+1/\kappa}\cdot\log n)$ and the hopbound β' given by*

$$\beta' = O\left(\frac{(\log\kappa\rho+1/\rho)\log n}{\varepsilon'}\right)^{\log\kappa\rho+1/\rho} \tag{4.12}$$

whp.
It does so by making $O(\beta'\cdot(\log\kappa\rho+1/\rho))$ passes through the stream and using $O(n\cdot\log^3 n\cdot\log\Lambda)$ bits of space in the first pass and $O(\frac{\beta'}{\varepsilon'}\cdot\log^2 1/\varepsilon'\cdot n^{1+\rho}\cdot\log^5 n)$ bits of space (respectively $O(\frac{\beta'^2}{\varepsilon'}\cdot\log^2 1/\varepsilon'\cdot n^{1+\rho}\cdot\log^5 n)$ bits of space for path-reporting hopset) in each of the subsequent passes.

4.3 $(1+\varepsilon)$-Approximate Shortest Paths in Weighted Graphs

Consider the problem of computing $(1+\varepsilon)$-approximate shortest paths (henceforth $(1+\varepsilon)$-ASP) for all pairs in $S\times V$, for a subset S, $|S|=s$, of designated source vertices, in a weighted undirected n-vertex graph $G=(V,E,\omega)$ with aspect ratio Λ.

Let $\varepsilon,\rho>0$ be parameters, and assume that $s=O(n^\rho)$. Our dynamic streaming algorithm for this problem computes a path-reporting $(1+\varepsilon,\beta)$-hopset H of G with $\beta=O(\frac{\log n}{\varepsilon\rho})^{1/\rho}$ using the algorithm described in Section 4.2, with $\kappa=1/\rho$. By Theorem 4.20, $|H|=O(\log n\cdot n^{1+\rho})$, the space complexity of this computation is

$O(n \cdot \log^3 n \cdot \log \Lambda)$ for the first pass and $O(n^{1+\rho}) \cdot \log^{O(1)} n$ for subsequent passes, and the number of passes is $O(\beta) = poly(\log n)$.

Once the hopset H has been computed, we conduct $(1+\varepsilon)$-approximate Bellman-Ford explorations in $G \cup H$ to depth β from all the sources of S. (See the algorithm from Section 2.4.) By Theorem 2.22, this requires $O(\beta)$ passes of the stream, and space $O(|S| \cdot n \cdot poly(\log n, \log \Lambda))$, and results in $(1+\varepsilon)$-approximate distances $d^{(\beta)}_{G \cup H}(s,v)$, for all $(s,v) \in S \times V$. (Note that following every pass over G, we do an iteration of Bellman-Ford over the hopset H *offline*, as H is stored by the algorithm.) In addition, for every pair $(s,v) \in S \times V$, we also get the parent of v on the exploration rooted at source s. We compute the path $\pi_{G \cup H}(s,v)$ between s and v in graph $G \cup H$ from these parent pointers. As described in Section 4.1.4, the path-reporting property of our hopset H enables us to replace any hopset edge $e = (x,y) \in H$ on the path $\pi_{G \cup H}(s,v)$ with a corresponding path $\pi_G(x,y)$ in G. By definition of the hopset, we have

$$d_G(s,v) \leq d^{(\beta)}_{G \cup H}(s,v) \leq (1+\varepsilon) \cdot d_G(s,v),$$

and the estimates $\hat{d}(s,v)$ computed by our approximate Bellman-Ford algorithm satisfy

$$d^{(\beta)}_{G \cup H}(s,v) \leq \hat{d}(s,v) \leq (1+\varepsilon) \cdot d^{(\beta)}_{G \cup H}(s,v).$$

Thus, we have

$$d_G(s,v) \leq \hat{d}(s,v) \leq (1+\varepsilon)^2 \cdot d_G(s,v).$$

By rescaling $\varepsilon' = 3\varepsilon$, we obtain $(1+\varepsilon)$-approximate $S \times V$ paths, the total space complexity of the algorithm is $O(n^{1+\rho} \cdot poly(\log n, \log \Lambda))$, and the number of passes is $poly(\log n)$. We derive the following theorem:

Theorem 4.21. *For any parameters $\varepsilon, \rho > 0$, and any n-vertex undirected weighted graph $G = (V, E, \omega)$ with polynomial in n aspect ratio, and any set $S \subseteq V$ of n^ρ distinguished sources, $(1+\varepsilon)$-ASP for $S \times V$ can be computed in dynamic streaming setting in $\tilde{O}(n^{1+\rho})$ space and $\log^{\frac{1}{\rho}+O(1)} n = polylog(n)$ passes.*

Algorithm 4 Pseudocode for procedure *FindNewCandidate*

1: **Procedure FindNewCandidate**(v,h,I)
2: ▷ Initialization
3: $slots \leftarrow \emptyset$ ▷ An array with $\lambda = \lceil \log\ n \rceil$ elements indexed
4: from 1 to λ.

▷ Each element of slots is a tuple $(sCount, sNames)$. For a given index $1 \leq k \leq \lambda$, fields *sCount* and *sNames* of $slots[k]$ can be accessed as $slots[k].sCount$ and $slots[k].sNames$, respectively.

5:

▷ $slots[k].sCount$ counts the new update candidates seen by v with hash values in $[2^k]$. It is set to 0 initially.

▷ $slots[k].sNames$ is an encoding of the names of candidate sources seen by v with hash values in $[2^k]$. It is set to ϕ initially.

▷ Update Stage

6: **while** (there is some update $(e_t,\ eSign_t,\ eWeight_t)$ in the stream) **do**
7: **if** (e_t is incident on v and some $u \in V$) **then**
8: **for each** $(s, \hat{d}(u,s)) \in LCurrent_u$ **do**
9: **if** $((\hat{d}(u,s) + eWeight_t) \in I$ and
10: $\hat{d}(u,s) + eWeight_t < \hat{d}(v,s))$ **then**
11: $k \leftarrow \lceil \log h(s) \rceil$
12: **repeat** ▷ Update $slots[k]$ for all $\lceil \log h(s) \rceil \leq k \leq \lambda$
13: $slots[k].sCount \leftarrow slots[k].sCount + eSign_t$
14: $slots[k].sNames \leftarrow slots[k].sNames + \nu(s) \cdot eSign_t$
15: ▷ The function ν is described in Section 2.2.4.
16: ▷ The addition in line 14 is a vector addition.
17: $k = k + 1$
18: **until** $k > \lambda$

▷ Recovery Stage

19: **if** (*slots* vector is empty) **then**
20: return (ϕ, ϕ)
21: **else if** ($\exists$ index k s.t. $\frac{slots[k].sNames}{slots[k].sCount} = \nu(s)$ for some s in V) **then**
22: return $(s, slots[k].sCount)$
23: **else**
24: return $(\perp, \perp)$

5

New Sparse Recovery and ℓ_0-Sampling Algorithms

In this chapter, we show that our sampler *FindNewVisitor* or its weighted counterpart *FindNewCandidate* (See Algorithms 3 and 4) in the dynamic streaming setting can also be used to provide a general purpose 1-sparse recovery and ℓ_0-sampler in the strict turnstile model. (Recall that a dynamic streaming setting is called *strict turnstile model*, if ultimate values of all elements at the end of the stream are non-negative, even though individual updates may be both positive or negative.) We consider a vector $\vec{a} = (a_1, a_2, \ldots, a_n)$, which comes in the form of a stream of updates. Each update is of the form $\langle i, \Delta a_i \rangle$, and it means that one needs to add the quantity Δa_i to the i^{th} coordinate of the vector $\vec{a}$. As was mentioned above, we assume that for each i, the ultimate sum of all the update values Δa_i that refer to the i^{th} coordinate is non-negative.

We say that the vector $\vec{a}$ is 1-*sparse*, if it contains exactly one element in its support. The support of $\vec{a}$ denoted $supp(\vec{a})$ is the set of coordinates $a_i \neq 0$.

In the 1-*sparse recovery* problem, if the input vector $\vec{a}$ is 1-sparse, the algorithm needs to return the (only) coordinate i in the support of $\vec{a}$ and its ultimate value a_i. Otherwise, the algorithm returns $\perp$ (indicating a failure). Ganguly (33) devised an algorithm for this problem in the strict turnstile setting, which occupies space $O(\log M + \log n)$, where M is the maximum value of any coordinate a_j for any $j \in [n]$ during the stream. Cormode and Firmani (17) devised an algorithm with the same space complexity which applies for integer update values in general turnstile

model (in which ultimate negative multiplicities of the coordinates, also known as frequencies, are allowed). (See Section 5.1.) We show an alternative solution to that of Ganguly (33) with the same space complexity.

5.1 1-Sparse Recovery

The basic idea is to use CIS-based encodings ν described in Section 2.2.4. Throughout the execution of our algorithm, we maintain a sketch $\mathscr{L}$ which is a two-dimensional vector in $\mathbb{R}^2$ and a counter ctr. Initially, $\mathscr{L} = \vec{0}$ and $ctr = 0$. Every time we receive an update $\langle i, \Delta a_i \rangle$, we update $\mathscr{L}$ as $\mathscr{L} = \mathscr{L} + \nu(i) \cdot \Delta a_i$ and update ctr as $ctr = ctr + \Delta a_i$. At the end of the stream, if $ctr \neq 0$, we compute $\mathscr{L}' = \frac{\mathscr{L}}{ctr}$. (If $ctr = 0$, we return ϕ, indicating that the input vector is empty.) The algorithm then tests if $\mathscr{L}' \in \{\nu(1), \nu(2), \ldots, \nu(n)\}$, and if it is the case, i.e., $\mathscr{L}' = \nu(i)$ for some $i \in [n]$, then it returns (i, ctr), and $\perp$ otherwise.

For the analysis, observe that $\mathscr{L} = \sum_{i=1}^{n} \nu(i) \cdot a_i$ and $ctr = \sum_{i=1}^{n} a_i$. If $|supp(\vec{a})| = 1$, then let $\{i\} = supp(\vec{a})$. In this case, $\mathscr{L} = \nu(i) \cdot a_i$ and $ctr = a_i$ and thus $\mathscr{L}' = \frac{\mathscr{L}'}{ctr} = \nu(i)$. We can therefore retrieve i from $\nu(i)$. On the other hand, if $|supp(\vec{a})| = 0$, then the algorithm obviously returns $\perp$. Finally, by Lemma 2.12, if $|supp(\vec{a})| \geq 2$, then $\mathscr{L}' \notin \{\nu(1), \nu(2), \ldots, \nu(n)\}$, and in this case algorithm returns a message *too dense*.

In the context of our application of the above algorithm to computing near-additive spanners, one can just keep an encoding table which records $\nu(i)$ for every $i \in [n]$.

However, for a general-purpose 1-sparse recovery, one needs to be able to compute $\nu(i)$ (given an index $i \in [n]$) using $polylog(n)$ space. One also needs to compute i from $\nu(i)$ using small space. Recall that we define $R = \Theta(n^{3/2})$ and $\nu(1), \nu(2), \ldots, \nu(n)$, $n = \Theta(R^{2/3})$ are the n vertices of the convex hull of the set of integer points within a radius-R disc, centered at the origin, ordered clockwise. These vectors can be computed by Jarník's constriction (See (50; 16)). The latter can be computed in $O(\log^2 n)$ space, but the fastest log-space algorithms that we know for this task retrieve all vertices one after another and thus require time at least linear in n.

To speed up this computation, we next describe another encoding σ which maps $[n]$ into $\mathbb{Z}^5$. As a result, each encoding $\sigma(i)$ uses by constant factor more space than $\nu(i)$. On the other hand, we argue below that $\sigma(i)$ and $\sigma^{-1}(\mathscr{L})$ can be efficiently computed using log-space, for any $i \in [n]$ and any feasible vector $\mathscr{L} \in \mathbb{Z}^5$. (By a *feasible vector*, we mean here that $\mathscr{L}$ is in the range of the mapping defined by σ.)

Let $R = n$ and consider a 5-dimensional sphere $\mathbb{S}$, centered at origin. The sphere contains $\Theta(R^3)$ integer points, but we will use just R of them. Specifically, for any $i \in [n]$, let (p_i, q_i, r_i, s_i) be a fixed four-square representation of $R^2 - i^2$, i.e., $R^2 - i^2 = p_i^2 + q_i^2 + r_i^2 + s_i^2$, where $p_i, q_i, r_i, s_i \in \mathbb{N}$. Then we define $\sigma(i) = (p_i, q_i, r_i, s_i)$. (Such a representation exists for every natural number by Lagrange's four-square theorem, see, e.g., (48).)

There exist a number of efficient randomized (Las Vegas) algorithms (48; 49) for computing a four-square representation of a given integer. One of these algorithms is deterministic. It is known to require time polynomial in $O(\log n)$, assuming Heath-Brown's conjecture (35) that the least prime congruent to a (mod q), when $\gcd(a,q) = 1$, is at most $q \cdot (\log q)^2$. (See Section (48))

Another alternative is to use a randomized algorithm of Rabin and Shalit (49) which has been recently improved by Pollack and Treviño (48) and requires expected time $O(\log^2 n / \log\log n)$.

The problem with it is, however, that it may return different representations $\sigma(i)$, when invoked several times on the same number $R^2 - i^2$, for some $i \in [n]$. To resolve this issue, one may use Nisan's pseudorandom generator (46) to generate the random string used by all the invocations of Pollack and Enrique's algorithm (48) from a seed of polylogarithmic ($O(\log^2 n)$) length. The latter seed can be stored by our algorithm. This ensures consistent computations of four-square representations of different integers by our algorithm.The resulting random string (produced by Nisan's generator) is indistinguishable from a truly random one from the perspective of any $polylog(n)$-space bounded algorithm. Since both our algorithm and that of Pollack and Treviño (48) are $polylog(n)$-space bounded, this guarantees the correctness of the overall computation.

5.2 ℓ_0-Sampling

To demonstrate the utility of our new 1-sparse recovery algorithm, we point out that this routine directly gives rise to an s-sparse recovery algorithm, for an arbitrarily large s. (For example, see the description of the first pass of sub-phase j of interconnection step in Section 3.1.2.)

A vector $\vec{a}$ is said to be *s-sparse* if $|supp(\vec{a})| \leq s$. In the s-sparse recovery problem, the algorithm accepts as input a vector $\vec{a}$. If the vector $\vec{a}$ is not s-sparse or $\vec{a} = \vec{0}$, the algorithm needs to report $\perp$. Otherwise, with probability at least $\delta > 0$, for a parameter $\delta > 0$, the algorithm needs to return the original vector $\vec{a}$. A direct approach to s-sparse recovery via 1-sparse recovery is described in (33) and in Section 2.3.2 of (17). It produces an algorithm whose space is $O(s \log \frac{1}{\delta})$ times the space of the 1-sparse recovery algorithm. One can use our 1-sparse recovery algorithm instead of those of (33) or (17) in it.

Yet another application of our 1-sparse recovery algorithm is ℓ_0-samplers. An ℓ_0-sampler may return a $\perp$ (a failure) with probability at most δ. But if it succeeds, it returns a uniform (up to an additive error of n^{-c}, for a sufficiently large c) coordinate i and the corresponding value a_i in the support of the input vector $\vec{a}$. The scheme we describe next is close to Jowhari et al (39), and has a similar space complexity to it. It however uses 1-sparse recovery directly, while the scheme of (39) employs s-sparse recovery (which, in turn, invokes 1-sparse recovery). Like Jowhari et al (39), we first describe the algorithm assuming a truly random bit string of length $O(m \log n)$, where m is the length of the stream and n is the length of the input vector $\vec{a}$. We then replace it by string produced by Nisan's pseudorandom generator out of a short random seed. This seed is stored by the algorithm. (Its length is $O(\log^2 n)$ like in (39).)

The algorithm tries $\log n$ scales $j = 1, 2, \ldots, \log n$, and each scale j corresponds to a guess of $s = |supp(\vec{a})|$ being in the range $2^{j-1} \leq s \leq 2^j$. On scale j each coordinate i is consistently sampled with probability 2^{-j}, and a 1-sparse recovery algorithm attempts to recover the subsampled vector.

For a fixed coordinate i, and for j such that $2^{j-1} < s \leq 2^j$, the probability that only i will be sampled is $\frac{1}{2^j} \cdot (1 - \frac{1}{2^j})^{j-1} \geq \frac{1}{2s}(1 - \frac{1}{s})^{s-1} \geq \frac{e^{-1}}{2s}$.

Since the event of two fixed distinct coordinates to be discovered are disjoint, it follows that the probability of the sampler to recover *some* coordinate is at least $\frac{e^{-1}}{2}$. Conditioned on its success to retrieve an element, by symmetry, it follows that the probabilities of different coordinates in $supp(\vec{a})$ to be recovered are equal. Once the truly random source is replaced by the string produced by Nisan's pseudorandom number generator, the probabilities, however, will be skewed by an additive term of n^{-c}, for a sufficiently large constant $c > 0$.

Similarly to the argument in (39), no $polylog(n)$-space tester is able to distinguish between the truly random string and the one produced by Nisan's pseudorandom generator. Thus, in particular, they are indistinguishable for our ($polylog(n)$-space bounded) algorithm.

Viewed as a tester, our algorithm may be fed with a specific set of non-zero coordinates in the support of its input vector and any specific coordinate i in the support that the algorithm can test whether it is returned. (This tester is $polylog(n)$-space bounded.)

The overall space requirement of the algorithm in $O(\log n)$ times the space requirement of the 1-sparse recovery routine. The latter is $O(\log n)$ as well. In addition to this space of $O(\log^2 n)$, the algorithm also needs to remember the random seed of Nisan' generator which is of length $O(\log^2 n)$ as well.

The failure probability of the algorithm is, as was shown above $e^{-1}/2$. If we want to decrease it to δ, we can run $O(log 1/\delta$ copies of this algorithm in parallel, and pick an arbitrary copy in which the algorithm succeeded. (If there exists such a copy, otherwise the algorithm returns a failure.) The overall space of the resulting algorithm becomes $O(\log^2 n \log 1/\delta)$, To summarize:

Theorem 5.1. *Our algorithm provides an L_0-sampler with failure probability at most $\delta > 0$, for a parameter δ, and additive error n^{-c}, for an arbitrarily large constant c which affects the constant hidden in the O-notation of space. Its space requirement is $O(\log^2 n \cdot \log 1/\delta)$.*

Part II

A Graph Processing Problem in Query Model

6

An improved query strategy for finding a king

A tournament is an orientation of a complete graph. We say that a vertex x in a tournament $\vec{T}$ *controls* another vertex y if there exists a directed path of length at most two from x to y. A vertex is called a *king* if it controls every vertex of the tournament. It is well known that every tournament has a king. We follow Shen, Sheng, and Wu (67) in investigating the *query complexity* of finding a king, that is, the number of arcs in $\vec{T}$ one has to know in order to surely identify at least one vertex as a king.

The aforementioned authors showed that one always has to query at least $\Omega(n^{4/3})$ arcs and provided a strategy that queries at most $O(n^{3/2})$. While this upper bound has not yet been improved for the original problem, Biswas *et al*. (62) proved that with $O(n^{4/3})$ queries one can identify a *semi*-king, meaning a vertex which controls at least half of all vertices.

Our contribution is a novel strategy which improves upon the number of controlled vertices: using $O(n^{4/3}\operatorname{polylog} n)$ queries, we can identify a $(\frac{1}{2}+\frac{2}{17})$-king. To achieve this goal we use a novel structural result for tournaments.

6.1 Introduction

A *tournament* is a directed graph in which there is exactly one directed edge between every pair of vertices. Due to their usefulness in modelling many real world

scenarios such as game tournaments, voting strategies and many more, tournaments are a very well studied concept in structural as well as algorithmic graph theory. The early monograph of Moon (66) has been followed by extensive research on the topic. For example, Dey (63) studied the identification of the 'best subset of vertices' in a tournament motivated by the high cost of comparing a pair of drugs for a specific disease. Goyal *et al*. (64) studied the identification of vertices with specific in- or out-degrees.

In this work we investigate the *query complexity* of finding a *king* in a tournament graph, that is, a vertex from which we can reach every other vertex of the tournament via a directed path of length at most two. It is well known that every tournament has such a vertex.

The study of *query complexity* problems in tournaments has the following general shape: Initially, we are only given the vertex set of the tournament while the directions of its arcs are hidden from us. For each pair of vertices u, v we can, at unit cost, learn whether the arc uv or vu is in the tournament. Our goal is to use the fewest possible queries in order to reveal some combinatorial object in the tournament. The motivation for our work is found in Shen, Sheng, and Wu's work (67) on the query complexity of identifying a king. They showed that $\Omega(n^{4/3})$ queries are always necessary and provided an algorithm which reveals a king using $O(n^{3/2})$ queries. Ajtai *et al*. (60) independently proved the same upper bound in the context of imprecise comparison.

One of the enticing aspects of this setting is its game-theoretic nature: we can alternatively think of it as an adversarial game where one player, the *seeker*, wants to identify a combinatorial structure by querying arcs of the tournament while an adversary, the *obscurer*, tries to delay the seeker for as long as possible by choosing the orientation of queried arcs.

When reading Shen, Sheng, and Wu (67), one may be tempted to conjecture that a better analysis of their obscurer-strategy for finding a king can lead to a better lower bound. However, Biswas *et al*. (62) showed that against this strategy, the seeker can find a king with $O(n^{4/3})$ queries. They also showed that there exists a seeker strategy with $O(n^{4/3})$ queries for identifying a *semi-king*, that is, a vertex which controls at least half of all vertices. This result is optimal by Lemma 6, Biswas *et al*. (62). In fact, one needs to make $\Omega(t^{4/3})$ queries for identifying a

vertex which controls at least $t \leq n$ vertices against the obscurer-strategy of Shen, Sheng and Wu. (See Lemma 6 of Biswas *et al*. (62) and Ajtai *et al*. (60) for more details.) Therefore, if there exists an obscurer-strategy that proves a stronger than $\Omega(n^{4/3})$ lower bound for the king problem, then this strategy must rely on some factors which distinguish the king problem from the semi-king problem. In our eyes, this means that such a lower bound is much more difficult to find than one might think at first.

Proceeding from the above, it is tempting to try to improve the upper bound by using a variation of the seeker-strategy from Shen, Sheng, and Wu (67) and we can interpret the Biswas *et al*. (62)'s seeker-strategy for finding a semi-king as such an attempt. These strategies both rely on repeatedly selecting a set of vertices and then querying all the edges between them to find a maximum out-degree (MOD) vertex in this sub-tournament[1]. Balasubramanian, Raman and Srinivasaragavan (61) showed that identifying an MOD vertex in a tournament of size k requires $\Omega(k^2)$ queries in the worst case, which may explain the limits of the existing seeker strategies.

Our Result

In this work, we proceed along the line of research just described. On the one hand, we show that with $\tilde{O}(n^{4/3})$ queries[2], it is possible to identify a $(\frac{1}{2} + \frac{2}{17})$-king, which indicates that improving upon the $\Omega(n^{4/3})$ lower bound is probably even harder than indicated by the semi-king results. On the other hand, our technique does not rely on finding MOD vertices of sub-tournaments which circumvents the inherent high cost of this operation.

Technical Overview

Our result is based on the combinatorial structure of tournaments, which may be of independent interest. We believe that this work provides a novel toolkit which could lead towards resolving the query complexity of finding a king. Specifically, we design a seeker-strategy which consists of two main stages:

[1] The relationship between MOD vertices and kings is well-established: Landau (65), while studying the structure of animal societies, showed that every MOD vertex is a king, but non-MOD kings can exist as well.

[2] The big-$\tilde{O}$ notation hides constants and polylogarithmic factors

(i) The seeker queries the orientation of a set of edges defined by a so-called *template-graph*. These queries are *non-adaptive* in the sense that the queries do not change as a result of the answers provided by the obscurer.

(ii) The seeker analyses the answer to the queries of (i) in order to select queries that lead to the revelation of a $(\frac{1}{2}+\frac{2}{17})$-king.

The template-graph is an undirected graph over the tournament's vertices that has $\tilde{O}(n^{4/3})$ edges, with the property that every set of vertices of size around $n^{2/3}$ or more has edges to almost all the graph. In Section 6.3, we use the probabilistic method to prove that such a graph exists. Given the template-graph, the seeker's queries in the first stage are simply given by its edges, i.e. if there exists an edge uv in the template-graph, then the seeker asks the obscurer about the orientation of the edge uv in the tournament. The sparsity of the template-graph ensures that the seeker does not make too many queries and the connectivity of every sufficiently large set ensures that we do not miss any relevant information.

The second stage of the seeker-strategy is built on showing that when the obscurer chooses how the edges of the template graph are oriented they have a trade-off. The trade-off is either to reveal an *ultra-set* or not. We show that if the obscurer reveals an *ultra-set*, then the seeker can identify a $(\frac{1}{2}+\frac{2}{17})$-king with $\tilde{O}(n^{4/3})$ extra queries. If the obscurer does not do this, then the seeker can use this to find a partition of the vertex set of the tournament into sets of size $O(n^{2/3})$ each (which we refer to as *tiles*), so that the edges of the template-graph that are incident to the tiles satisfy a certain property. We obtain this combinatorial object by showing that if such a partition does not exist, then a simple set of queries already reveals a $(\frac{1}{2}+\frac{2}{17})$-king.

The tiles are analysed by the construction of what we refer to as the *free matrix* which contains a row for every tile and a column for every vertex of the tournament. An entry of the matrix indexed by a given tile-vertex pair is 1 if every edge between the vertex and a vertex in the tile is directed towards the tile, otherwise the entry is 0. We then use this free matrix to guide the seeker-strategy.

Given that the first part of the seeker-strategy is non-adaptive and against any adversary, this approach also reveals a combinatorial property of tournaments: for any fixed tournament and template graph with the same set of vertices, knowing

only the direction of the arcs of the tournament that correspond to the arcs of the template graph is sufficient for finding a set of vertices S of size $O(n^{2/3})$ such that querying all edges inside of S necessarily reveals a $(\frac{1}{2}+\frac{2}{17})$-king. We note that fraction $\frac{2}{17}$ is the result of balancing the various trade-offs in the seeker strategy.

The rest of the chapter is organised as follows. In Section 6.2, we provide necessary definitions and prove some basic lemmas about tournaments that are used in the rest of the chapter. Section 6.3 is dedicated to the formal definition and the proof of existence of template graphs. In Section 6.4, we describe our seeker-strategy and prove that it leads to the discovery of a $(\frac{1}{2}+\frac{2}{17})$-king. In Section 6.5, we give concluding remarks and open problems.

6.2 Preliminaries

Given an undirected graph G, a vertex $v \in V(G)$, and a vertex set $X \subseteq V(G)$ we define the *relative degree* $d(v,X) := |N(v) \cap X|$, where $N(v)$ is the neighbourhood of v in G.

For a vertex v in a directed graph $\vec{G}$, a vertex u in $\vec{G}$ is an out-neighbour of v, if the edge between u and v is oriented from v to u. For a directed graph $\vec{G}$, we denote the out-neighbourhood of a vertex $v \in \vec{G}$ by $N^+(v)$ and its out-degree by $d^+(v)$. For a vertex set $X \subseteq V(\vec{G})$, we let $d^+(X)$ be the number of arcs from a vertex in X to a vertex not in X.

Given additionally a vertex subset $X \subseteq V(G)$, we define the *relative out-degree* $d^+(v,X) := |N^+(v) \cap X|$.

The (closed) *second-out-neighbourhood* $N^{++}[v]$ of v is the set of vertices $u \in V(\vec{G})$ for which there exists a directed path from v to u of length at most two.

For simplicity we will adopt the following vocabulary for digraphs. We say that a vertex x *controls* a vertex y if $y \in N^{++}[x]$. We say that x *directly controls* a vertex y if $y \in N^+(x) \cup \{x\}$. We extend both of these terms to vertex sets U, for example, we will often write statements like 'x controls at least half of the vertices in U'.

A tournament is a digraph $\vec{T}$ obtained from a complete graph by replacing each edge with a directed arc. As done usually, we denote the subgraph induced by a vertex set $S \subseteq V(\vec{T})$, with $\vec{T}[S]$. Note that an induced subgraph of a tournament

is necessarily also a tournament. We will need the following basics facts about tournaments in the following.

Lemma 6.1. *Let $\vec{T}$ be a tournament with m vertices and $\alpha \in [0,1]$ such that αm is even. Then $\vec{T}$ has at least $(1-\alpha)m$ vertices of out-degree at least $\alpha m/2$.*

Proof. Let S initially be the vertices of $\vec{T}$ and proceed according to the following process: find a vertex of out-degree at least $\alpha m/2$ and remove it from S; and repeat until no such vertex exists in S. From here on we focus on set S after the vertex removal process ended.

Let $r = |S|$ be the size of the final set and consider the sub-tournament $\vec{T}[S]$. We know that by averaging considerations, every tournament of size r has at least one vertex of out-degree at least $r/2 - 1/2$. We also know that S does not contain any vertex of out-degree at least $\alpha m/2$. Hence, we conclude that $r \leq \alpha m$.

Consequently, our process discovered $m - r \geq (1-\alpha)m$ vertices of out-degree at least $\alpha m/2$ in $\vec{T}$. □

Lemma 6.2. *Let $\vec{G}$ be an orientation of a complete bipartite graph (V_0, V_1, E), where $|V_0| = |V_1| = m$, and m is divisible by 4 . Then, there exists $i \in \{0,1\}$, such that V_i has at least $m/2+1$ vertices v, where $d^+(v, V_{1-i}) \geq m/4$.*

Proof. Let S initially contain all the vertices in $V_0 \cup V_1$ and proceed according to the following process: We find a vertex of out-degree at least $m/4$ and remove it from S. Repeat until no such vertex exists and we are left with $S' \subseteq S$.

Every orientation of the complete bipartite graph $K_{t,t}$ must contain, by a simple averaging argument, a vertex of out-degree at least $t/2$. Therefore the induced subgraph $\vec{G}[S']$ must have at least one partite set of size strictly less than $m/2$ or we could continue the process. Consequently, our process discovered at least $m/2+1$ vertices of out-degree at least $m/4$ in that partite set. □

Lemma 6.3. *Let $\vec{T}$ be a tournament on $2m$ vertices, where m is divisible by 4. Let further sets S_0, S_1 be a partition of the vertices of $\vec{T}$ into sets of equal size. Then there exists a vertex v such that both $d^+(v, S_0) \geq m/4$ and $d^+(v, S_1) \geq m/4$.*

Proof. By Lemma 6.1, for both $i \in \{0,1\}$, there exist $m/2$ vertices v in S_i such that $|N^+(v) \cap S_i| \geq m/4$. By Lemma 6.2 for *one* of $i \in \{0,1\}$, there exist $m/2+1$

vertices v such that $|N^+(v) \cap S_{1-i}| \geq m/4$. Then by the pigeonhole principle, there exists $i \in \{0,1\}$ and a vertex $v \in S_i$, such that $|N^+(v) \cap S_i| \geq m/4$, for every $i \in \{0,1\}$, as claimed. □

6.3 Constructing the template-graph

Definition 6.4 (κ-template-graph)**.** *Let $\kappa \in (0,1)$ and G be an undirected graph over the vertex set $[n]$. The graph G is a* κ-template-graph*, if for every pair of disjoint sets $H_1, H_2 \subseteq [n]$ both of size at least $\kappa n^{2/3}$, there exists at least one edge between them, that is, $|E(H_1, H_2)| \geq 1$.*

For the remainder of this section, we fix $\kappa \in (0,1)$ and set $p = \frac{2\log n+2}{\kappa n^{2/3}}$. We next show that with strictly positive probability the Erdős–Renyi random graph $G(n,p)$ is a *κ-template-graph*, with $O(n^{4/3}\log n)$ edges, where the O notation hides a dependence on κ. By the probabilistic method, this implies that there actually exists such a graph.

All probabilities in the following are with respect to the probability space of this random graph.

Lemma 6.5. *Let $\kappa \in (0,1)$. With probability at least $3/4$, the graph $G(n,p)$, where p is defined as above, is a κ-template-graph.*

Proof. Note that if we prove the statement of the lemma for sets of size *exactly* (up to rounding errors) $\kappa n^{2/3}$, then the claim follows for all larger sets as well. To prove this, we next show, with the help of the union bound, that the probability that $G(n,p)$ has two disjoint subsets of vertices, each of size $\kappa n^{2/3}$, with no edge between them is strictly less than $1/4$.

Let H_1 and H_2 be any pair of disjoint subsets of $[n]$ of size $\kappa n^{2/3}$, then the total number of vertex pairs between them is $(\kappa n^{2/3})^2$. The probability that none of these pairs is an edge in the template-graph G is accordingly $(1-p)^{(\kappa n^{2/3})^2}$.

We apply the exponential bound $(1-p)^k \leq e^{-pk}$ for a k-round Bernoulli trial and obtain

$$\begin{aligned}(1-p)^{(\kappa n^{\frac{2}{3}})^2} &\leq e^{-p(\kappa n^{2/3})^2} = e^{-(2\log n+2)\kappa n^{2/3}} \\ &= n^{-2\kappa n^{2/3}} e^{-2\kappa n^{2/3}} \leq \frac{1}{4} n^{-2\kappa n^{2/3}},\end{aligned}$$

where the last inequality holds when n is large enough so that $\kappa n^{\frac{2}{3}} \geq 1$ and hence $e^{-2\kappa n^{2/3}} < \frac{1}{4}$. Since the total number of pairs of sets H_1, H_2 of size $\kappa n^{2/3}$ is bounded above by $n^{2\kappa n^{2/3}}$, the claim now follows from the union bound. □

Theorem 6.6. *Let $\kappa \in (0,1)$, there exists a κ-template-graph G with at most $O(n^{4/3}\log n/\kappa)$ edges.*

Proof. The expected number of edges of $G(n,p)$ for our choice of $p = (2\log n + 2)/(\kappa n^{2/3})$ is less than $m := (2\log n+2)n^{4/3}/\kappa$. Since every edge of the graph is selected independently, by the Chernoff bound the probability that the number of edges in $G(n,p)$ exceeds $2m$ is at most

$$e^{-p\binom{n}{2}/3} \leq e^{-\frac{(2\log n+2)}{\kappa n^{2/3}} \frac{n(n-1)}{6}} \leq e^{-(2\log n+2)n/6} \leq \frac{1}{4}$$

where the last inequality holds for $n \geq 4$.

Together with Lemma 6.5 this implies that with probability at least $1/2$, $G(n,p)$ is a κ-template-graph G with at most $O(n^{4/3}\log n/\kappa)$ edges. The claim follows by the probabilistic method. □

6.4 The seeker strategy

Having proved the existence of a κ-template-graph, we next examine the properties of an arbitrary orientation of such a graph. Given a κ-template-graph G_κ on a vertex set $[n]$, we use $\vec{G}_\kappa$ (*henceforth*) to refer to a directed graph obtained by replacing every edge of G_κ by a directed arc. Note that we assume nothing about $\vec{G}_\kappa$ and analyze as if its arcs were arbitrarily oriented by an adversary.

Definition 6.7 (η-weak, η-strong, η-ultra)**.** *For an oriented template-graph $\vec{G}_\kappa$ and any $\eta > 0$, a set $H \subseteq [n]$ is η-*weak *if $d^+(H) < (1/2+\eta)n$, is η-*strong *if*

$d^+(H) \geq (1/2+\eta)n$. *We call a set* η-ultra *if every subset* $H' \subset H$*, of size at least* $|H|/2$ *is* η*-strong.*

To understand why η-ultra sets are important, it is useful to think of the seeker as trying to force the obscurer to reveal enough information (in the form of query answers) so that the seeker can achieve their goal. This is done by first querying the orientation of all the edges of a template graph and nothing else. The observation below implies that if the orientation of the edges of the template graphs reveals an *eta*-ultra set, of size $\tilde{O}(n^{2/3})$, then the seeker can achieve its goal with an additional $\tilde{O}(n^{4/3})$ queries. Thus, the obscurer cannot reveal such an ultra-set. However, as we show further on, by doing this the obscurer reveals enough information for the seeker to achieve their goal.

Observation 6.8. *Let* $H \subseteq V(\vec{G}_\kappa)$ *be an* η*-ultra set. Then we can find a* $(1/2+\eta)$*-king using* $\leq |H|^2$ *additional queries.*

Proof. Query all $\leq |H|^2$ edges inside H. Let $v \in H$ be a vertex such that $d^+(v,H) \geq |H|/2$. Since H is η-ultra, the set $H' := N^+(v) \cap H$ is η-strong, meaning $d^+(H') \geq (1/2+\eta)n$. Therefore $|N^{++}[v]| \geq (1/2+\eta)n$ and v is a $(1/2+\eta)$-king. □

Definition 6.9 (Free set)**.** *Let* $W \subseteq V(\vec{G}_\kappa)$ *be an* η*-weak set. Then the* free set *of* W *is the vertex set* $F(W) := V(\vec{G}_\kappa) \setminus (N^+(W) \cup W)$*, that is, all vertices that lie neither in* W *nor in* $N^+(W)$*.*

Observation 6.10. *Let* W *be an* η*-weak set. Then* $|F(W)| > (\frac{1}{2}-\eta)n - |W|$.

By the properties of template-graphs, namely that each pair of large enough sets must have an edge between them, and by the definition of free sets it follows that all the arcs of $\vec{G}_\kappa$ between a sufficiently large set W and its free set $F(W)$ must point towards W. Let us formalize this intuition:

Definition 6.11 (α-covers)**.** *For* $\alpha \in [0,1]$ *we say that a set* S α-covers *a set* W *if* $|N^+(S) \cap W| \geq \alpha|W|$.

Lemma 6.12. *In the template graph, for every set* $W \subset [n]$ *of size* $n^{2/3}$ *and every subset* $S \subseteq F(W)$ *of size at least* $\kappa n^{2/3}$*, it holds that* S $(1-\kappa)$*-covers* W.

Proof. Consider a set $S \subseteq F(W)$ of size $\kappa n^{2/3}$. Then, by Lemma 6.5, there is at least one edge s_1w_1 between some $s_1 \in S$ and $w_1 \in W$. Remove w_1 from W and apply the argument to the remainder. In this way, we construct a sequence $w_1, \ldots, w_t$ such that each w_i has at least one neighbour in S.

The application of Lemma 6.5 is possible until the remainder of W has size less than $\kappa n^{2/3}$, hence the process works for at least $t = n^{2/3} - \kappa n^{2/3} = (1-\kappa)n^{2/3}$ steps. Now simply note that each edge sw_i for $s \in S$ must be oriented from s to w_i since S is a subset of $F(W)$. It follows that $|N^+(S) \cap W| \geq (1-\kappa)|W|$, as claimed. □

In the previous lemma lies the inherent usefulness of free sets. If, for some set W of size $n^{2/3}$, we find a vertex v that has at least $\kappa n^{2/3}$ out-neighbours in the free set $F(W)$, then v controls almost all of W. As observed above, η-weak sets have necessarily large free sets which makes them 'easy targets' for our strategy.

We now show that in case no η-ultra set exists (in which case we already win as per Observation 6.8), we can instead partition most of the vertices of $V(\vec{G}_\kappa)$ into weak sets.

Definition 6.13. *An* η-weak tiling *of* $\vec{G}_\kappa$ *is a vertex partition* $W_1, \ldots, W_m, R$ *where* $|W_i| = n^{2/3}$, $|R| < 2n^{2/3}$ *and every set* W_i *is* η*-weak. We call the sets* W_i *the* tiles *and* R *the* remainder.

By definition, the number of tiles m in an η-weak tiling is at least $n^{1/3} - 2$.

Lemma 6.14. *Fix* $\eta > 0$. *For large enough* n, $\vec{G}_\kappa$ *either contains an* η*-ultra set of size* $2n^{2/3}$ *or an* η*-weak tiling.*

Proof. We construct the tiling iteratively. Assume we have constructed $W_1, \ldots, W_j$ so far. Let $R := V(\vec{G}_\kappa) \setminus \bigcup_{i \leq j} W_i$ be all the vertices of $\vec{G}_\kappa$ which are not yet part of the tiling. If $|R| < 2n^{2/3}$ we are done, so assume otherwise. Let $H \subseteq R$ be an arbitrary vertex set of size $2n^{2/3}$. If H is η-ultra, then by Observation 6.8, we are done. Otherwise there exists an η-weak set $W_{j+1} \subseteq H$, $|W_{j+1}| = |H|/2$. Add this set to the tiling and repeat the construction. At the end of this procedure, we will either find an η-ultra set or an η-weak tiling. □

Our goal is now to find a vertex whose out-neighbourhood has large intersections with many free sets. To organise this search, we define the following auxiliary structure:

Definition 6.15 (Free matrix)**.** *Let $W_1,\dots,W_m,R$ be an η-weak tiling of $\vec{G}_\kappa$ and let $\mathscr{W}=\{W_1,\dots,W_m\}$. The* free matrix *$M$ of the tiling $\mathscr{W},R$ is a binary matrix with m rows indexed by $\mathscr{W}$ and n columns indexed by $[n]$. The entry at position $(W_i,v)\in\mathscr{W}\times V$ is 1 if $v\in F(W_i)$ and 0 otherwise.*

We will use the following notation in the rest of this section. Given a free matrix M of an η-weak tiling $\mathscr{W},R$ let $M[\mathscr{W}',U]$ denote a sub-matrix of M induced by a subset $\mathscr{W}'\subseteq\mathscr{W}$ of the tile set and a subset $U\subseteq[n]$ of the vertex set. For example, a column of M corresponding to a vertex $v\in[n]$ can be written as $M[\mathscr{W},\{v\}]$ in this notation. Analogously a row of M corresponding to a tile $W_i\in\mathscr{W}$ can be written as $M[\{W_i\},[n]]$. Given a sub-matrix M' of the free matrix M, we call the number of 1's in M' the *weight* of M' and denote it by $\sum M'$.

The following is a direct consequence of the construction of the free matrix and Observation 6.10.

Observation 6.16. *Every row of the free matrix M has a weight of at least $(\frac{1}{2}-\eta-n^{-1/3})n$.*

Definition 6.17 (Good Sub-Matrix)**.** *A sub-matrix $M[\mathscr{W},U]$, for some $U\subset[n]$, is* η-good *if, each one of its rows has weight at least $(\frac{1}{2}-\eta-2n^{-1/3}\log^{1/2}n)|U|$.*

We next show that a good sub-matrix with $2n^{2/3}$ columns exists, by using the probabilistic method. Specifically, we show that if we randomly pick $2n^{2/3}$ columns from the matrix $M[\mathscr{W},[n]]$ then with strictly positive probability the matrix that includes exactly these columns is good.

Lemma 6.18. *Let $\eta\in(0,\frac{1}{2})$. For large enough n the free matrix M has an η-good sub-matrix with $2n^{2/3}$ columns.*

Proof. Select $K\subset[n]$ of size $2n^{2/3}$ uniformly at random. Let $M'=M[\mathscr{W},K]$.

We set $p=1/2-\eta-n^{-1/3}$ and $t=n^{-1/3}\log^{1/2}n$. By Observation 6.16, every row of M has weight at least pn. By the Hypergeometric tail bound the probability

that a specific row of M' has weight less than $(p-t)n$ is at most $e^{-2t^2 2n^{2/3}} \leq 1/n$, where the inequality follows from our choice of t. Then by the union bound the probability that *every* row of M' has weight at least $(p-t)n$ is strictly positive.

Note that by our choice of p and t, we get that $(p-t)n$, for large enough n, is at least as large as $(1/2-\eta-2n^{-1/3}\log^{1/2} n)n$, therefore a good sub-matrix M' exists with strictly positive probability. By the probabilistic methods the claim therefore holds. □

Next we show that the only way that the adversary does not provide us with a $(1/2+\delta)$-king once we have identified a δ-good sub-matrix is if the distribution of 1's in that matrix is very restricted. We will use this additional structure to find a $(1/2+\delta)$-king in the sequel. For simplicity, we first query all the edges between the vertices associated with the columns of the δ-good sub-matrix, but note that this is not strictly necessary: we can instead inspect all potential partitions (with properties as stated in the lemma) and only if no such partition exists query said edges which then surely identifies a $(1/2+\delta)$-king. With this change the lemma is consistent with the structural claim from the introduction.

Lemma 6.19. *Let $M' = M[\mathscr{W}, V]$ be a δ-good sub-matrix of M with $2n^{2/3}$ columns. Let further $\kappa+\delta \leq 1/2$. If we query each one of the $O(n^{4/3})$ edges in V, then either we find a $(\frac{1}{2}+\delta)$-king, or we find partitions $V_1 \uplus V_2 = V$ and $\mathscr{W}_1 \uplus \mathscr{W}_2 = \mathscr{W}$ with the following properties:*

- $|V_1| = |V_2| = n^{2/3}$
- $|\mathscr{W}_1| \geq (\frac{1}{2}-\delta-\kappa)n^{1/3}-2$ *and* $|\mathscr{W}_2| < (\frac{1}{2}+\delta+\kappa)n^{1/3}$
- *Every row in $M'[\mathscr{W}_1, V_1]$ has weight at most $\kappa n^{2/3}$*
- *Every row in $M'[\mathscr{W}_2, V_1]$ has weight at least $\kappa n^{2/3}$*

Proof. We query all the edges in $V \times V$ and select a vertex $y \in V$ such that $d^+(y,V) \geq n^{2/3}$. Let V_1 be an arbitrary subset of $N^+(y) \cap V$ of size $n^{2/3}$ and let $V_2 = V \setminus V_1$. Partition the rows of M' into $\mathscr{W}_1 \cup \mathscr{W}_2$ so that $\mathscr{W}_1$ contains all rows with weight less than $\kappa n^{2/3}$ in the sub-matrix $M'[\mathscr{W}, V_1]$.

We claim that if $|\mathscr{W}_1| < (\frac{1}{2}-\delta-\kappa)n^{1/3}-2$ then y is a $(\frac{1}{2}+\delta)$-king. By construction, every row in $\mathscr{W}_2$ has weight at least $\kappa n^{2/3}$ in $M'[\mathscr{W}, V_1]$ and if the condition

of the claim holds then $|\mathscr{W}_2| \geq (\frac{1}{2}+\delta+\kappa)n^{1/3}$. By Lemma 6.12, the set V_1 must $(1-\kappa)$-cover every tile in $\mathscr{W}_2$. It follows that

$$\begin{aligned}
|N^{++}[y]| &\geq (1-\kappa)\Big|\bigcup \mathscr{W}_2\Big| \geq (1-\kappa)\left(\Big(\frac{1}{2}+\delta+\kappa\Big)n^{1/3}\right)n^{2/3} \\
&= (1-\kappa)\Big(\frac{1}{2}+\delta+\kappa\Big)n = \Big(\frac{1}{2}+\delta+\frac{\kappa}{2}-\kappa\delta-\kappa^2\Big)n \\
&= \left(\frac{1}{2}+\delta+\kappa\Big(\frac{1}{2}-\delta-\kappa\Big)\right)n \\
&\geq \Big(\frac{1}{2}+\delta\Big)n
\end{aligned}$$

where the last inequality holds since $\delta+\kappa \leq 1/2$. □

Lemma 6.19 implies the following about good sub-matrices.

Lemma 6.20. *Let $M' = M[\mathscr{W}, V]$ be a δ-good sub-matrix with $|V| = 2n^{2/3}$ and $\mathscr{W}_1 \uplus \mathscr{W}_2$, $V_1 \uplus V_2$ be partitions as in Lemma 6.19. Then every row in $M'[\mathscr{W}_1, V_2]$ has weight at least $(1-2\delta-2\kappa)n^{2/3}$.*

Proof. Since M' is a δ-good sub-matrix, by Definition 6.17, every row in $M'[\mathscr{W}_1, V]$ has weight at least $(1/2-\delta-2n^{-1/3}\log^{1/2}n)2n^{2/3}$. By Lemma 6.19, every row in $M'[\mathscr{W}_1, V_1]$ has weight at most $\kappa n^{2/3}$. Therefore, the weight of every row in $M'[\mathscr{W}_1, V_2]$ is

$$\begin{aligned}
&\geq (1/2-\delta-2n^{-1/3}\log^{1/2}n)2n^{2/3} - \kappa n^{2/3} \\
&= (1-2\delta-4\frac{\log^{1/2}n}{n^{1/3}}-\kappa)n^{2/3} \\
&\geq (1-2\delta-2\kappa)n^{2/3}
\end{aligned}$$

where we assume that n is large enough so that $4\frac{\log^{1/2}n}{n^{1/3}} \leq \kappa$. □

Our final technical lemma lets us, for a given set of rows of M, identify a set of columns with high enough weight when restricted to those rows.

Lemma 6.21. *Let $U \subset [n]$ be of size $2n^{2/3}$ and $\mathscr{W}' \subset \mathscr{W}$. Then there exists a set $V' \subset [n] \setminus U$, of size $n^{2/3}$ such that every column in $M[\mathscr{W}', V']$ has weight at least $(1/2-\delta-3n^{-1/3})|\mathscr{W}'|$.*

Proof. Let $\bar{U} := [n] \setminus U$ and let V' be an arbitrary subset of $n^{2/3}$ columns in $M[\mathscr{W}', \bar{U}]$ with the largest column-weight. Let t be the smallest weight among

these columns when restricted to $M[\mathscr{W}', V']$. We bound the weight of $M[\mathscr{W}', \bar{U}]$ first from below and then from above, then we use these bounds to show that $t > (\frac{1}{2} - \delta - 3n^{-1/3}) \cdot |\mathscr{W}'|$, which implies that V' is the claimed set.

For the lower bound on the weight of $M[\mathscr{W}', \bar{U}]$, we use the simple fact that

$$\sum M[\mathscr{W}', \bar{U}] = \sum M[\mathscr{W}', [n]] - \sum M[\mathscr{W}', U]. \tag{6.1}$$

By Observation 6.16, every row of the matrix M has weight least $(\frac{1}{2} - \delta - n^{-1/3}) \cdot n$. It follows that $\sum M[\mathscr{W}', [n]]$ is at least $(\frac{1}{2} - \delta - n^{-1/3}) \cdot n \cdot |\mathscr{W}'|$. For the second term, we have the trivial bound $\sum M[\mathscr{W}', U] \leq |U| \cdot |\mathscr{W}'| = 2n^{2/3} \cdot |\mathscr{W}'|$. Plugging these values into (6.1) we obtain

$$\begin{aligned} \sum M[\mathscr{W}', \bar{U}] &\geq (\frac{1}{2} - \delta - n^{-1/3}) \cdot n \cdot |\mathscr{W}'| - 2n^{2/3} \cdot |\mathscr{W}'| \\ &= (\frac{1}{2} - \delta - 3n^{-1/3}) \cdot n \cdot |\mathscr{W}'|. \end{aligned} \tag{6.2}$$

For the upper bound on the total weight of $M[\mathscr{W}', \bar{U}]$ we use that

$$\sum M[\mathscr{W}', \bar{U}] = \sum M[\mathscr{W}', V'] + \sum M[\mathscr{W}', \bar{U} \setminus V']. \tag{6.3}$$

We use the trivial bound $\sum M[\mathscr{W}', V'] \leq |V'| \cdot |\mathscr{W}'| = n^{2/3} \cdot |\mathscr{W}'|$ for the first term. By definition of the value t, we have that every column in $M[\mathscr{W}', \bar{U} \setminus V']$ has weight at most t. Accordingly, $\sum M[\mathscr{W}', \bar{U} \setminus V'] \leq t \cdot |\bar{U} \setminus V'| = t \cdot (n - 3n^{2/3})$. Plugging in these values into (6.3) we obtain

$$\sum M[\mathscr{W}', \bar{U}] \leq n^{2/3} \cdot |\mathscr{W}'| + t \cdot (n - 3n^{2/3}). \tag{6.4}$$

Finally, (6.2) and (6.4) taken together give us that

$$t \cdot (n - 3n^{2/3}) + n^{2/3} \cdot |\mathscr{W}'| \geq (\frac{1}{2} - \delta - 2n^{-1/3}) \cdot n \cdot |\mathscr{W}'|.$$

Consequently, $t > (\frac{1}{2} - \delta - 3n^{-1/3}) \cdot |\mathscr{W}'|$ and we conclude that V' has the claimed property. □

We are finally ready to prove our seeker-strategy for finding a $1/2 + \delta$-king using $\tilde{O}(n^{4/3})$ queries. For readability, we will state our main result in terms of concrete and simple values for κ and δ, however, note that smaller values of κ allow δ to be slightly larger than the stated bound of $\frac{2}{17}$.

Theorem 6.22. *Fix $\delta = \frac{2}{17}$, let $\kappa = \frac{1}{4000}$. For larger enough n, there exists a seeker strategy for finding a $(1/2+\delta)$-king using $\tilde{O}(n^{4/3})$ edge queries.*

Proof. We construct the template-graph G_κ and query all $\tilde{O}(n^{4/3})$ of its edges to obtain $\vec{G}_\kappa$.

By Lemma 6.14, we either obtain a δ-ultra set of size $2 \cdot n^{2/3}$ or a δ-weak tiling of $\vec{G}_\kappa$. If we find the former, by Observation 6.8 we can find a $(\frac{1}{2}+\delta)$-king using $O(n^{4/3})$ additional queries. Therefore assume that we obtained a δ-weak tiling $\mathscr{W}, R$ of $\vec{G}_\kappa$.

Let M be the free matrix of $\mathscr{W}, R$. By Lemma 6.18, M has a δ-good sub-matrix $M[\mathscr{W}, V]$ with $|V| = 2n^{2/3}$. We query all $O(n^{4/3})$ edges in $V \times V$ and by Lemma 6.19 either identify a $(\frac{1}{2}+\delta)$-king, or obtain partitions $V_1 \uplus V_2 = V$, $\mathscr{W}_1 \uplus \mathscr{W}_2$ with properties as listed in Lemma 6.19. Importantly, by Lemma 6.20, every row in the sub-matrix $M[\mathscr{W}_1, V_2]$ has weight at least $(1-2\delta-2\kappa)n^{2/3}$.

We now apply Lemma 6.21 with $\mathscr{W}' = \mathscr{W}_2$ and find a set of columns $V_3 \subseteq [n] \setminus V$ of size $n^{2/3}$ such that every column in $M[\mathscr{W}_2, V_3]$ has weight at least $(\frac{1}{2} - \delta - 3n^{-1/3})|\mathscr{W}_2|$. We now query all edges in $V_3 \times V_3$ and $V_2 \times V_3$, since $|V_2| = |V_3| = n^{2/3}$ this amounts to $O(n^{4/3})$ additional queries.

Since $\vec{G}[V_2 \cup V_3]$ is completely revealed, it is a tournament of size $2n^{2/3}$ and we apply Lemma 6.3 using the bipartition (V_2, V_3) to find a vertex $v \in V_2 \cup V_3$ such that $d^+(v, V_2)$ and $d^+(v, V_3)$ are both at least $n^{2/3}/4$. We claim that v is a $(\frac{1}{2}+\delta)$-king. Let in the following $V_2' = N^+(v) \cap V_2$ and $V_3' = N^+(v) \cap V_3$. We first prove the following two claims about these two sets:

Claim 6.23. *Every row in $M[\mathscr{W}_1, V_2']$ has weight at least $\kappa n^{2/3}$.*

Proof. [Proof of the claim] According to Lemma 6.20, every row in $M[\mathscr{W}_1, V_2]$ has weight at least $(1-2\delta-2\kappa)n^{2/3}$. Since $|V_2'| = |V_2|/4 = n^{2/3}/4$, we have that each row in $M[\mathscr{W}_1, V_2']$ has weight at least

$$(1-2\delta-2\kappa)n^{2/3} - \frac{3}{4}n^{2/3}$$

which is larger than $\kappa n^{2/3}$ for $\delta \leq \frac{1}{8} - \frac{3\kappa}{2}$ which holds true for our choices of δ and κ. □

Claim 6.24. *At least* $(\frac{1}{2}-\delta-4\kappa-3n^{-1/3})\frac{|\mathscr{W}_2|}{1-4\kappa}$ *rows in* $M[\mathscr{W}_2,V_3']$ *have weight at least* $\kappa n^{2/3}$.

Proof. [Proof of the claim] Recall that by choice of V_3, every column in $M[\mathscr{W}_2,V_3]$ and therefore also $M[\mathscr{W}_2,V_3']$ has weight at least $(\frac{1}{2}-\delta-3n^{-1/3})|\mathscr{W}_2|$. Accordingly,

$$\begin{aligned}\sum M[\mathscr{W}_2,V_3'] &\geq (\frac{1}{2}-\delta-3n^{-1/3})|\mathscr{W}_2|\cdot|V_3'| \\ &\geq (\frac{1}{2}-\delta-3n^{-1/3})|\mathscr{W}_2|\cdot\frac{1}{4}n^{2/3}.\end{aligned} \tag{6.5}$$

Let t denote the number of rows in $M[\mathscr{W}_2,V_3']$ with weight at least $\kappa n^{2/3}$. Our goal is to find a lower bound for t. Since t is minimized if every row that has weigth at least $\kappa n^{2/3}$ has in fact the maximum possible weight $|V_3'|=n^{2/3}/4$, we can lower-bound t using

$$\frac{t}{4}n^{2/3}+(|\mathscr{W}_2|-t)\kappa n^{2/3}\geq\sum M[\mathscr{W}_2,V_3'].$$

Combining this inequality with (6.5), we obtain

$$\begin{aligned}& t\frac{n^{2/3}}{4}+(|\mathscr{W}_2|-t)\kappa n^{2/3}\geq(\frac{1}{2}-\delta-3n^{-1/3})|\mathscr{W}_2|\cdot\frac{1}{4}n^{2/3} \\ \iff\ & t(1-4\kappa)\geq(\frac{1}{2}-\delta-4\kappa-3n^{-1/3})|\mathscr{W}_2| \\ \iff\ & t\geq(\frac{1}{2}-\delta-4\kappa-3n^{-1/3})\frac{|\mathscr{W}_2|}{1-4\kappa}\end{aligned}$$

□

Now note that for every tile $W\in\mathscr{W}$ for which the row $M[\{W\},V_2'\cup V_3']$ has weight at least $\kappa n^{2/3}$ we have that $d^+(v,F(W))\geq\kappa n^{2/3}$, therefore by Lemma 6.12 the set $N^+(v)\cap F(W)$ $(1-\kappa)$-covers W. In other words, v controls at least $(1-\kappa)n^{2/3}$ vertices in W.

Our goal is now to lower-bound the total number of such tiles, hence let s denote the total number of rows in $M[\mathscr{W},V_2'\cup V_3']$ with weight at least $\kappa n^{2/3}$. By the previous two observations and by plugging in the concrete values of $\delta=\frac{2}{17}$ and $\kappa=\frac{1}{4000}$, we have that

$$\begin{aligned}s&\geq|\mathscr{W}_1|+(\frac{1}{2}-\delta-4\kappa-3n^{-1/3})\frac{|\mathscr{W}_2|}{1-4\kappa} \\ &=|\mathscr{W}_1|+(\frac{6483}{17000}-3n^{-1/3})|\mathscr{W}_2|\frac{1000}{999}\end{aligned}$$

Again we are aiming to prove a lower-bound, thus we assume that $\mathscr{W}_1$ is as small as possible. By Lemma 6.19, this means that

$$|\mathscr{W}_1| = (\frac{1}{2} - \delta - \kappa)n^{1/3} - 2 = \frac{25983}{68000}n^{1/3} - 2 \quad \text{and}$$
$$|\mathscr{W}_2| = (\frac{1}{2} + \delta + \kappa)n^{1/3} = \frac{42017}{68000}n^{1/3}.$$

Plugging in the sizes of $\mathscr{W}_1, \mathscr{W}_2$ we obtain

$$\begin{aligned} s &\geq |\mathscr{W}_1| + (\frac{6483}{17000} - 3n^{-1/3})|\mathscr{W}_2|\frac{1000}{999} \\ &\geq \frac{25983}{68000}n^{1/3} + (\frac{6483}{17000} - 3n^{-1/3})\frac{42017}{67932}n^{1/3} - 2 \\ &\geq \frac{475777}{769896}n^{1/3} - 4. \end{aligned}$$

Since v controls a $(1-\kappa) = \frac{3999}{4000}$ fraction of each tile counted by s and each tile has a size of $n^{2/3}$, we finally have the following lower bound on the second out-neighbourhood of v:

$$\begin{aligned} |N^{++}[v]| \geq \frac{3999}{4000}sn^{2/3} &\geq \frac{3999}{4000} \cdot \frac{475777}{769896}n - 4n^{2/3} \\ &= 0.61782\ldots n - 4n^{2/3}. \end{aligned}$$

This value lies, for large enough n, above our target value of $(\frac{1}{2}+\delta)n = 0.61764\ldots n$.

□

6.5 Conclusion

We have shown how the usage of a *template-graph* helped us devise a seeker strategy that reveals a $(\frac{1}{2} + \frac{2}{17})$-king in a tournament using $\tilde{O}(n^{4/3})$ queries, shedding light on a long-standing open problem. Our approach begins with a non-adaptive querying strategy based on what we called a *template graph*, which then helps to guide the seeker to identify a small set of queries which necessarily lead to the discovery of a $(\frac{1}{2} + \frac{2}{17})$-king.

Naturally, we ask whether it is possible to find an improved strategy which reveals a $(\frac{1}{2}+\delta)$-king with δ substantially larger than $\frac{2}{17}$ using a similar amount of queries.

Part III

A Graph Processing Problem in Distributed Computing Model

7

Distributed Expansion Testing with Local Test Criteria

Given an n-vertex graph G and an expansion parameter α, the problem of testing whether G has expansion at least α or is far from having expansion at least $\Omega(\alpha^2)$ has been extensively studied (see (84; 74; 85; 88)) in the classic property testing model.

In the last few years, the same problem has also been addressed in non-sequential models of computing such as MPC and distributed CONGEST. However, all the algorithms in these models like their classic counterparts rely on some global information pertaining to the entire graph as a test criteria. We propose a new distributed algorithm with a local test criteria. The only distributed algorithm for the problem (78) tests conductance of the underlying network. Unlike the algorithm of (78), our algorithm does not rely on the wasteful construction of a spanning tree and information accumulation at its root. This makes our algorithm time and communication efficient and avoids a central point of failure.

7.1 Introduction

Many distributed algorithms are known to perform better provided that the underlying distributed network satisfies a certain property. For example, the randomised leader election algorithm by (86) works better on graphs that are good expanders. Similarly, (90) presents fast distributed coloring algorithms for triangle-

free graphs. Therefore, it makes sense to check if the graph of the underlying network satisfies a certain property or not. However, it may be hard to efficiently *verify* certain global graph properties in the CONGEST model of distributed computing. In this model, each vertex of the input graph acts as a processing unit and works in conjunction with other vertices to solve a computational problem. The computation proceeds in synchronous rounds, in each of which every vertex can send an $O(\log n)$-bits message to each of its neighbours, do some local computations and receive messages from its neighbours. (See (89) for more details.)

Distributed decision problems are tasks in which the vertices of the underlying network have to collectively decide whether the network satisfies a global property $\mathscr{P}$ or not. If the network indeed satisfies the property, then all vertices must accept and, if not, then at least one vertex in the network must reject. For many global properties, lower bounds on the number of rounds of computation of the form $\tilde{\Omega}(\sqrt{n}+D)$ are known for distributed decision, where n is the number of vertices and D is the diameter of the network. (See (75) for more details.) It makes sense to relax the decision question and settle for an approximate answer in these scenarios as is done in the field of property testing (see (80; 83)) in the sequential setting.

A property testing algorithm in the sequential setting arrives at an approximate decision about a certain property of the input by *querying* only a small portion of it. Specifically, an ε-tester for a graph property $\mathscr{P}$ is a randomised algorithm that can distinguish between graphs that satisfy $\mathscr{P}$ and the graphs that are ε-far from satisfying $\mathscr{P}$ with high constant probability. An m-edge graph G is considered ε-far from satisfying $\mathscr{P}$ if one has to modify (add or delete) more than $\varepsilon \cdot m$ edges of G for it to satisfy $\mathscr{P}$. Two-sided error testers may err on all graphs, while one-sided error testers have to present a witness when rejecting a graph. The cost of the algorithm is measured in the number of queries made. (See (80; 83; 82; 81) for a detailed exposition of the subject.)

7.1.1 Distributed Property Testing

A *distributed property testing* problem is a relaxed variant of the corresponding distributed decision problem, where if the input network satisfies a property, then, with high constant probability, all the vertices must accept and, if the input net-

work is ε-far from satisfying the property, then at least one vertex must reject. The definition of "farness" in distributed setting remains the same as in the classical setting. The complexity measure is the number of rounds required to test the property. Distributed property testing has been an active area of research recently. The work of (72) was the first to present a distributed algorithm (for finding near-cliques) with a property testing flavour. Later Censor-Hillel, Fischer, Schwartzman and Vasudev (86) did a more detailed study of distributed property testing. There has been further study on the topic (see (77) and (79)) in the specific context of subgraph freeness.

7.1.2 Expansion Testing

We address the problem of testing the expansion of a graph in the distributed CONGEST model of computing. A distributed expansion tester can be a useful pre-processing step for some distributed algorithms (such as (86)) which perform better on expander graphs. Focusing on vertex expansion, we call a graph $G = (V,E)$ an α-*expander* if every $U \subseteq V$ such that $|U| \leq |V|/2$ has at least $\alpha|U|$ neighbours. Here a vertex $v \in V \setminus U$ is a neighbour of U if it has at least one edge incident to some $u \in U$. Expansion testing for bounded degree graphs has been extensively studied in the classic property testing model. For a constant d, A graph $G = (V,E)$ is called a *bounded degree graph* with degree bound d if every $v \in V$ has degree at most d. In the classic setting, this problem was first studied by (84) and later by (74). Their work was followed by the state of the art results by (85) and (88). Both these papers present $\tilde{O}(\sqrt{n}/\alpha^2)$-query testers for distinguishing between graphs that have expansion at least α and graphs that are ε-far from having expansion at least $\Omega(\alpha^2)$. In the last few years, the same problem has also been addressed in non-sequential models of computing such as MPC by (87) and distributed CONGEST by (78). The distributed algorithm by Fichtenberger and Vasudev (78) tests the conductance of the input network in the unbounded degree graph model.

A typical algorithm for the problem in the sequential, as well as non-sequential, models can be thought of as running in two *phases*. In the first phase, the algorithm performs a certain number of short ($O(\log n)$-length) random walks from a

randomly chosen starting vertex. The walks should mix well on a graph with high expansion and should take longer to mix on a graph which is far from having high expansion (at least from some fraction of starting vertices). In the second phase, the algorithm then checks whether those walks mixed well or not. For that, the algorithm gleans some information from every vertex in the graph and computes some aggregate function. Specifically in the classic and MPC settings, the algorithms count the total number of pairwise collisions between the endpoints of the walks. The only known distributed algorithm for the problem by (78) precedes the first phase by building a rooted BFS spanning tree of the input graph.[1] This spanning tree is used for collecting information from the endpoints of the random walks in the second phase. Specifically, their algorithm estimates the discrepancy of the endpoint probability distribution from the stationary distribution by going over the discrepancy on each endpoint individually. If the discrepancy is above a certain threshold, the algorithm rejects the graph. This process of building a spanning tree and collecting information at the root to decide if the property holds or not takes a global and centralized view of the testing process.

The following natural question arises in the context of the second phase:

Question 7.1. *Is it possible to execute the second phase without computing a global aggregate function?*

In the classic setting, one strives for testers that make a sublinear (in n) number of queries which translates to running a sublinear number of walks. With only a sublinear ($O(\sqrt{n})$) number of walks, one hardly expects to see any useful information by itself on any individual vertex or in a small constant neighbourhood around it to know if the walks mixed well or not. Therefore, one has to rely on an aggregate function such as the total number of pairwise collisions between the endpoints of the walks.

In the non-sequential settings such as distributed CONGEST, one can utilize the parallelism to run a superlinear number of short walks while keeping the run time proportional to the length of the walks. This inspires us to stick to Question 7.1 in

[1] If the construction of BFS tree takes longer than $O(\log n)$ rounds the algorithm rejects without proceeding to the first phase since all good expanders have small diameter. However, a bad expander such as a dumbbell graph may also have small diameter, so their algorithm still need to proceed with the test after the successful construction of the spanning tree.

distributed setting and investigate what information one should store at each vertex during phase 1 and how it should be processed *locally* in phase 2 to allow each node to decide locally whether it is part of a good expander or not. This leads us to answer the following question in affirmative for bounded-degree graphs.

Question 7.2. *Can we test the expansion in distributed CONGEST model without collecting global information at a central point and come up with a local test criteria at each node?*

We believe that our analysis for bounded-degree graphs can be extended to the unbounded case by modifying the definition of our walk matrix appropriately and resolve Question 7.2 for the general case.

The advantage of a local test criteria is that it lets us do away with the wasteful construction of a spanning tree and information accumulation at the root. Note that we do not need to do any pre-processing for the construction of the spanning tree before phase 1. After phase 1, each node can compute its output bit without any information exchange. This saves us $O(\log n)$ rounds of communication and $O(n)$ messages in phase 2. All these savings make our algorithm more time and communication efficient and, perhaps more importantly, avoids a central point of failure. Moreover, our algorithm can tolerate any number of edge deletions during phase 2. Any implementation of parallel random walks in dynamic networks (with edge deletions) during phase 1 opens up the possibility of a fully-fault-tolerant tester for dynamic networks.

Finally, we believe that our approach to distributed property testing for expansion with emphasis on a local test criteria gives a new perspective for distributed property testing in general. The question of local verification of global properties has been well studied in the area of distributed verification and self-stabilization. The authors of (68) and (71) suggested that one can check certain global predicates on a distributed network by using local means: i.e., by having each vertex check the state of all its neighbours. It makes sense to look at the distributed property testing through a similar lens.

7.1.3 Our Results

We study the problem of testing the expansion of undirected graphs in the CONGEST model. We present a two-sided error distributed test algorithm with a local test criteria in the CONGEST model for testing vertex expansion.

Theorem 7.3. *For an input graph $G = (V,E)$ with degree bound d, and parameters $0 < \alpha < 1$ and $\varepsilon > 0$, there exists an $O(\log n/\alpha^2)$-round distributed algorithm with a local test criteria for testing with high constant probability whether G is an α-expander or is ε-far from being an $\Omega(\alpha^2)$-expander.*

A related notion of expansion considers the number of edges coming out of a set of vertices, instead of the number of neighbours. A graph is said to be an *α-edge-expander* if, for every $U \subseteq V$ that satisfies $|U| \leq |V|/2$, we have $E(U, V \setminus U) \geq \alpha|U|$. It follows that every α-edge-expander of bounded degree d is also an (α/d)-expander. The following corollary follows from the definition of edge expansion for bounded-degree graphs.

Corollary 7.4. *Let $\beta \in (0,1)$ and consider the algorithm of Theorem 7.3 with $\alpha = \beta/d$. Then, this algorithm distinguishes with high constant probability between β-edge-expanders and graphs that are from being ε-far an $\Omega(\beta^2/d^2)$-edge-expander.*

7.1.4 Technical Overview

In this section, we give a general overview of our algorithm and the concepts used in arriving at a local test criteria. Like all the previous algorithms for expansion testing, our algorithm proceeds in two phases. In the first phase, we perform a certain number of random walks from a randomly selected starting vertex. To boost the success probability of the process, we repeat this process in parallel from a constant number of randomly selected starting vertices. The main technical challenge in running random walks in parallel from different starting vertices is the congestion on the edges. As done by (78), we overcome this problem by not sending the entire trace of the walk from its current endpoint to the next. For each starting point q and for all the walks going from u to v, we simply send the ID of q and the number of walks destined for v to v. At the end of this process, for each starting point q, we simply store at each vertex v, the number of walks that ended at v. In

phase 2, each vertex $v \in V$ looks at the information stored at v at the end of phase 1 to check if the number of walks received from any starting vertex is than a certain threshold. If so, it outputs *Reject* and, otherwise, it outputs *Accept*.

To show that the number of walks a vertex v receives is sufficient to decide whether v is part of a good expander or not, we proceed as follows. A technical lemma from (74) implies that if a graph is ε-far from being an α-expander, then there exists a large enough set $S \in V$ of sufficiently low conductance (see Definition 7.6) in G. It follows intuitively that it is likely that a short random walk starting from a randomly selected starting vertex in S should not go very far and end in S. Not only that, we show that there exist a lot of vertices $v \in S$ such that short walks starting from v end in a large enough region T (subset of S) around v. We make this notion precise by using spectral graph theory to show that the low-conductance set S (as described above) contains a lot of *sticky* vertices. We call a vertex $v \in S$ sticky if there exists a set $T \subseteq S$ such that $v \in T$ and short random walks starting from v end in T with a *sufficiently* high probability. We define $\text{trap}(v, T, \ell)$ as the probability that an ℓ length walk starting from $v \in T \subseteq S$ ends in T. We show that a sufficiently large fraction of all vertices are sticky: that is, each has a high trap probability with respect to a set T, where T is a subset of a low-conductance set S in the graph. Thus, if we run sufficiently many walks from one of these vertices, then a lot of them will end in a subset T of S and some vertex in T will see a lot more walks than any vertex in a good expander should. To ensure that we pick one of the sticky vertices as a starting vertex, we sample each vertex to be a starting vertex with appropriate probability.

7.1.5 Organisation

The rest of the chapter is organised as follows. In Section 7.2, we provide necessary definitions and state some basic lemmas that are used in rest of the chapter. In Section 7.3, we provide the pseudocode and a detailed description of our testing algorithm. Section 7.3.1 is dedicated to the proof of our main theorem.

7.2 Preliminaries

Let $G = (V,E)$ be an unweighted, undirected graph on n vertices representing a distributed network. Each vertex $v \in V$ is a processing unit and can communicate with its *neighbours*, $N(v)$, using their identifiers. For a given vertex $v \in V$, $\deg(v)$ denotes the *degree* of v. In a *bounded-degree graph* with degree bound d, for all $v \in V$, $\deg(v) \leq d$.

A graph of bounded-degree d is said to be ε-*far* from satisfying a graph property $\mathscr{P}$ (or, in other words, being a member of a certain family of graphs closed under graph isomorphism), if one needs to add and/or delete more than εdn edges to G in order to turn it into a graph satisfying $\mathscr{P}$.

For sets $A, B \subseteq V$, we denote by $E(A,B)$ the number of edges that have one endpoint in A and the other endpoint in B.

Definition 7.5. *We say that a graph $G = (V,E)$ is an α-vertex-expander (or α-expander, for short) if, for every $U \subseteq V$ that satisfies $|U| \leq |V|/2$, we have $N(U) \geq \alpha|U|$.*

A cut in a graph is a partition of the vertices of the graph into two disjoint subsets. Given a graph $G = (V,E)$ any subset $S \subseteq V$ defines a cut denoted by $(S,\overline{S})$, where $\overline{S} = V \setminus S$.

Definition 7.6. *Given a cut $(S,\overline{S})$ in G, the conductance of $(S,\overline{S})$ is defined as*

$$\frac{E(S,\overline{S})}{\min\{vol(S), vol(\overline{S})\}},$$

where $vol(A) = \sum_{v \in A} deg(v)$. Alternatively, we also refer to the conductance of a cut $(S,\overline{S})$ as the conductance of set *S. The conductance of a graph is the minimum conductance of any cut in the graph.*

The main technical tool in our analysis will be random walks on the input graph G. We denote a random walk by its transition matrix M. For a pair of vertices $u, v \in V$, we let M_{uv} denote the probability of going from from u to v in one step of M. In the standard definition of a random walk, M_{uv} is defined as $1/\deg(u)$.

We will use a slightly modified version of the standard random walk called a *lazy random walk* in our work. In a *lazy random walk*, we transform the input graph

G with degree bound d to be $2d$-*regular* by adding $2d - \deg(v)$ self loops to each $v \in V$. Note that this change reflects only in the transition probabilities and does not involve any actual physical changes to the input network. Let M be a lazy random walk on G, the transition probabilities for M are defined as follows, For a pair $v, w \in V$ such that $v \neq w$, $M_{vw} = \frac{1}{2d}$, if $(v,w) \in E$ and 0, otherwise. Furthermore, for $v \in V$, we define $M_{vv} = 1 - \frac{\deg(v)}{2d}$.

On a $2d$-regular, connected, graph, a lazy random walk M can be viewed as a reversible, aperiodic Markov chain $\mathscr{M}_G$ with state space V and transition matrix M. Moreover, the stationary distribution of such a chain is uniform over the state space.

Definition 7.7. *Let $\mathscr{M}$ be a reversible, aperiodic Markov chain on a finite state space V with stationary distribution π. Furthermore, let $\pi(S) = \sum_{v \in S} \pi(v)$. The Cheeger constant or conductance $\phi(\mathscr{M})$ of the chain is defined as*

$$\phi(\mathscr{M}) = \min_{S \subset V : \pi(S) \leq 1/2} \frac{\sum_{x \in S, y \in V \setminus S} \pi(x) \mathscr{M}(x,y)}{\pi(S)}.$$

The definition of transition probabilities of M and the the fact that the stationary distribution π of our lazy random walk is uniform together imply that the Cheeger constant $\phi(M)$ (henceforth, ϕ_*) of the walk $\mathscr{M}$ is

$$\phi_* = \min_{U \in V, |U| \leq |V|/2} \frac{N(U)}{2d \cdot |U|}.$$

For an α-vertex-expander with bounded degree d, we get that

$$\phi_* \geq \frac{\alpha}{2d}. \tag{7.1}$$

We use the asymptotic notation $O_d()$ to indicate the dependency of a quantity on the parameters other than the degree bound d.

7.3 A Distributed Algorithm for Expansion Testing

Given an expansion parameter α, a distance parameter ε and an input graph $G = (V,E)$, with n vertices and degree bound d, our distributed expansion tester algorithm tests whether G is an α-vertex expander or is ε-far from being an $\Omega(\alpha^2)$-vertex expander. A key technical lemma from (74) implies that, if G is ε-far from

being an $\Omega(\alpha^2)$-expander, then there exists a low-conductance cut $(S, \overline{S})$ such that $\varepsilon n/12 \leq |S| \leq n/2$. We build on this lemma to show, using spectral methods (see Lemma 7.14 and Corollary 7.15), that there exist a constant fraction of *sticky* vertices in S. Recall that a vertex x in a low-conductance set S is sticky if there exists a large enough subset $T \subset S$ such that $x \in T$ and a short random walk starting from x *ends* in T with a sufficiently high probability. Intuitively, random walks starting from sticky vertices tend to stick to a small region around them. This leads to some vertex in the graph receive more than their *fair share* of number of walks. On the other, hand if a graph is a good expander, then the random walks from anywhere in the graph mix very quickly. This ensures that all the vertices in the graph receive roughly the same number of walks.

In our algorithm, we uniformly sample a set Q of $\Omega(1/\varepsilon)$ *source* vertices in the input graph and run random walks from each of them separately. Since a constant fraction of vertices of our large low-conductance set S in a bad expander are sticky, some vertex in Q will be sticky with sufficiently large probability. We exploit the fact that some vertex near a sticky vertex will see more number of walks than a pre-defined threshold and we use the number of walks received by each vertex from a specific source as a test criteria.

We implement sampling of the set Q by having each vertex v sample itself by flipping a biased coin with probability $48/(\varepsilon n)$. It follows from Chernoff bound that the probability of Q having more than $96/\varepsilon$ vertices is at most $e^{-16/\varepsilon} \leq e^{-16}$. Then, we perform K random walks of length ℓ starting from each of the chosen vertices in Q. The exact values of these parameters are specified later in the sequel. The entire algorithm is summarized in Algorithm 5. At any point before the last step of random walks, each vertex $v \in V$ contains a set W of tuples $(q, count, i)$, where *count* is the number of walks of length i originating from source q currently stationed at v. All these walks are advanced by one step (for ℓ times) by invoking Algorithm 6. At the end of the last step of the walks, Algorithm 6 outputs a set of tuples C_v. Each tuple in C_v is of the form $(q, count)$, where *count* is the total number of ℓ-step walks starting at q that ended at v. Then, in Algorithm 5, processor at vertex v goes over every tuple $(q, count)$ in C_v (see Lines 11 to 15 of Algorithm 5), and if the *count* value of any of them is above a pre-defined threshold τ, it outputs

Reject. If none of the tuples have their *count* value above threshold, it outputs *Accept*. The exact value of τ is specified later in the sequel.

The implementation of (one step) of a separate set of random walks from each of the starting points in Q is described in Algorithm 6. when advancing the walks originating at a source $q \in Q$ by one step, we do not send the full trace of every random walk. Instead, for every source $q \in Q$, every vertex $v \in V$ only sends a tuple (q,k,i) to its neighbour w indicating that k random walks originating at q have chosen w as their destination in their ith step. Since the size of Q is constant with high enough probability, we will not have to send more than a constant number of such tuples on each edge. Moreover, each tuple can be encoded using $O(\log n)$ bits (given the values of parameters ℓ and K specified in the sequel). Hence, we only communicate $O(\log n)$ bits per edge in any round with high probability. To make sure we do not congest any edge, we check the length of every message (see lines 13 to 15 of Algorithm 6) before we send it. If a message appears too large to send, we simply output *Reject* on the host vertex and abort the algorithm. Note that the number of tuples we ever have to send along any edge is upper bounded by $|Q|$ and $|Q| \leq 96\varepsilon$, with probability at least $1 - e^{-16}$. Therefore, we may rarely abort the algorithm before completing the execution of the random walks. If that happens, then the probability of accepting an α-vertex expander is slightly reduced. Hence the following observation follows:

Observation 7.8. *Algorithm 5 rejects an α-expander due to congestion with probability at most e^{-16}.*

For some $0 < \mu \leq 1$, we set the required parameters of Algorithm 5 as follows:

- the number of walks $K = n^{1+\mu}$,
- the length of each walk $\ell = \frac{16d^2}{\alpha^2} \log n$
- the rejection threshold $\tau = n^{\mu} + n^{2\mu/3} + 2$.

Note that the choice of the parameter μ affects the number walks started by each source vertex. Additionally, the smaller value of μ reduces the size of the class of non-expander graphs that will rejected by the algorithm with high probability. Therefore, one can slightly reduce the number of walks at the cost of a slight sacrifice in the precision.

Algorithm 5 Distributed algorithm running at vertex v for testing graph expansion.

Algorithm DISTRIBUTED-GRAPH-EXPANSION-TEST($G, \varepsilon, \alpha, \ell, K$)
▷ The algorithm performs K random walks of length ℓ from a set Q of $\Theta(1/\varepsilon)$ starting vertices, where every starting vertex is sampled randomly from V.
ℓ : The length of each random walk
K : The number of walks
W_v : Set of tuples $(q, count, i)$ ▷ where *count* is the number of walks originating at source q currently stationed at v
C_v : Set of tuples $(q, count)$ ▷ where *count* is the total number of ℓ step walks starting at q that ended at v
τ : maximum number of ℓ-length walks v should see from a given source on an α-expander.
Flip a biased coin with probability $p = 48/\varepsilon n$ to decide whether to start K lazy random walks.
If chosen, initialise W_v as $W_v \leftarrow \{(v, K, 0)\}$.
Call Algorithm 6 for ℓ synchronous rounds.
while there is some tuple $(q, count)$ in C_v **do**
  if $count > \tau$ **then** ▷ Received too many walks from q.
    Output *Reject* and stop all operations.
  else
    Remove $(q, count)$ from C_v
Output *Accept*

7.3.1 Analysis of the Algorithm

In this section, we show that Algorithm 5 accepts every α-expander with bounded degree d, with probability at least $2/3$, and rejects every graph with bounded degree d that is ε-far from being an α^*-expander, where $\alpha^* = \Theta(\alpha^2/d^2)$, with probability at least $2/3$.

The main idea behind our algorithm is that, in a bad expander, a random walk would converge to the stationary distribution more slowly and would initially get trapped within sets of vertices with small conductance. We use the following combinatorial lemma from (74) that shows the existence of a large low-conductance set in a graph which is far from being a good expander.

Lemma 7.9 (Corollary 4.6 of (74))**.** *Let $G = (V, E)$ be a bounded-degree graph, with $|V| = n$ and degree bound d. There exists a constant $C = C(d) > 0$ such that,*

Algorithm 6 Algorithm for moving random walks stationed at v by one step.

Algorithm MOVE-WALKS-AT-v
  W_v : Set of tuples $(q, count, i)$ ▷ where *count* is the number of walks originating at source q currently stationed at v just after their ith step..
  D_v : Set of tuples $(q, count, dest)$ ▷ where *count* is the number of walks starting at q that are to be forwarded to *dest*.
  C_v : Set of tuples $(q, count)$ ▷ where *count* is the total number of walks starting at q that have v as their final destination or endpoint.
  $D_v \leftarrow \emptyset$.
  while there is some tuple (q,k,i) in W_v **do**
    if $i \neq L$ **then** ▷ If not the last step, process the next set of destinations.
      Draw the next set of destinations for the k walks and update the set D_v.
      Remove (q,k,i) from W_v
  ▷ If last step of the walks, update how many ended at v.
      Update C_v to reflect the k walks that ended in v.
  ▷ Prepare the messages to be sent
  while there is some tuple $(q, count, dest)$ in D_v **do**
    Add tuple $(q, count, i+1\rangle$ to the message to be sent to *dest*
  ▷ Check each message for length
  For each message M to be sent
  if the number of tuples in $M > 96/\varepsilon$ **then**
    Output *Reject* and stop all operations.
  Send all the messages to their respective destinations.
  ▷ Process the messages received
  For each source s from which a total of s_{count} walks are received,
add tuple $(s, s_{count}, i+1)$ to W_v

for $0 < \varepsilon < 1$ and $\alpha^* \leq 1/10$, if G is ε-far from any α^*-expander, then there is a subset $A \subset V$, such that $\varepsilon n/12 \leq |A| \leq (1+\varepsilon)n/2$, such that $|N(A)| \leq C\alpha^*|A|$.

Lemma 7.9 implies the following corollary.

Corollary 7.10. *For $0 < \varepsilon < 1$ and $\alpha^* \leq 1/10$, if G is ε-far from any α^*-expander, then there exists a set $B \subset V$ with $\varepsilon n/12 \leq |B| \leq n/2$ and a constant $C = C(d,\varepsilon)$ such that $E(B, V \setminus B) \leq Cd\alpha^*|B|$, i.e., the conductance of cut $(B, \overline{B})$ is at most $C\alpha^*$.*

Proof. By Lemma 7.9, there exists a subset $A \subset V$, with $\varepsilon n/12 \leq |A| \leq (1+\varepsilon)n/2$, such that we have $|N_G(A)| \leq C\alpha^*|A|$. Furthermore, the degree bound d of G implies

that

$$E(A, V \setminus A) = E(V \setminus A, A) \leq dN_G(A) \leq dC\alpha^*|A|.$$

If $|A| \leq n/2$, then the claim follows simply by taking $B = A$. If $|A| > n/2$, we take $B = V \setminus A$, so that $|A| \leq (1+\varepsilon)n/2$ and $|B| \geq (1-\varepsilon)n/2$. It follows that

$$E(B, V \setminus B) \leq Cd\alpha^*|A| = Cd\alpha^* \frac{|A|}{|B|}|B| \leq C\frac{1+\varepsilon}{1-\varepsilon}d\alpha^*|B|,$$

where the last inequality follows by bounding $|A|$ from above and $|B|$ from below. □

Next we provide a lower bound for the probability of an ℓ-step random walk starting from a vertex chosen uniformly at random from a subset T of a low conductance set S finishing at some vertex in T.

Definition 7.11. *For a set $T \subseteq V$, and a vertex $u \in T$, let* $\mathrm{trap}(u, T, \ell)$ *(henceforth* trap probability*) denote the probability of an ℓ-step random walk starting from $u \in T$ finishing at some vertex in T. When the starting vertex is chosen uniformly at random from T, we denote by* $\mathrm{trap}(T, \ell)$ *the average trap probability over set T:*

$$\mathrm{trap}(T, \ell) = \frac{1}{|T|} \sum_{u \in T} \mathrm{trap}(u, T, \ell).$$

Given a set S (with $|S| = s$) of conductance at most δ and $T \subseteq S$ (with $|T| = t$), we establish a relationship between the average trap probability $\mathrm{trap}(T, \ell)$ and conductance bound δ of S in the next two lemmas. We first consider the case $T = S$ in Lemma 7.12, and then obtain a bound when T is a large subset of S in Lemma 7.14.

Lemma 7.12. *Consider a set $S \subseteq V$, such that $|S| = s$ and the cut $(S, \overline{S})$ has conductance at most δ. Then, for any integer $\ell > 0$, the following holds*

$$\mathrm{trap}(S, \ell) \geq \frac{s}{n} + \left(\frac{7}{8} - \frac{s}{n}\right)(1 - 8\delta)^\ell.$$

Proof. Let $\mathbb{1}_S$ denote the n-dimensional indicator vector of set S. Denote by M the transition matrix of a lazy random walk on G. The endpoint probability distribution $\vec{p}_S^{\,\ell}$ of an ℓ-step lazy random walk on G staring from a vertex chosen uniformly at random from S is given by

$$\vec{p}_S^{\,\ell} = \frac{1}{s}\mathbb{1}_S^{\mathsf{T}} M^\ell.$$

The trap probability $\text{trap}(S,\ell)$ of an ℓ-step lazy random walk starting from a random vertex in S can be expressed as the inner product of vectors $\vec{p}_S^{\,\ell}$ and $\mathbb{1}_S$:

$$\text{trap}(S,\ell) = \frac{1}{s}\mathbb{1}_S^{\mathsf{T}} M^{\ell}\mathbb{1}_S.$$

Let $1 = \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n > 0$ be the eigenvalues of M and let $\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n$ be the corresponding orthogonal unit eigenvectors. We express $\mathbb{1}_S$ in the orthonormal basis defined by the eigenvectors of M as $\mathbb{1}_S = \sum_i \alpha_i \vec{e}_i$. It follows that

$$\sum_i \alpha_i^2 = \langle \mathbb{1}_S, \mathbb{1}_S \rangle = s.$$

Next, we will consider the normalized Laplacian of the input graph G, which is given by $L = I - M$, where I is the identity matrix. (See Chapter 1 of (73) for more details.) Taking the quadratic form of L for vector $\mathbb{1}_S$, we get

$$\mathbb{1}_S^{\mathsf{T}} L \mathbb{1}_S = \mathbb{1}_S^{\mathsf{T}} I \mathbb{1}_S - \mathbb{1}_S^{\mathsf{T}} M \mathbb{1}_S = \langle \mathbb{1}_S, \mathbb{1}_S \rangle - \sum_i \alpha_i^2 \lambda_i = s - \sum_i \alpha_i^2 \lambda_i. \tag{7.2}$$

By definition of L in terms of M (see Chapter 1 of (73)), it follows that

$$\mathbb{1}_S^{\mathsf{T}} L \mathbb{1}_S = \sum_{i<j} M_{ij} \cdot \left(\mathbb{1}_S(i) - \mathbb{1}_S(j)\right)^2. \tag{7.3}$$

Here $\mathbb{1}_S(i)$ and $\mathbb{1}_S(j)$ denote the ith and jth entries, respectively of vector $\mathbb{1}_S$. Note that the term $\left(\mathbb{1}_S(i) - \mathbb{1}_S(j)\right)^2$ in (7.3) is non-zero only for pairs i, j corresponding to the cut edges $(i, j) \in E(S, \overline{S})$. Since the conductance of the cut $(S, \overline{S})$ is less than δ and the volume of the set S in G with degree bound d is at most ds, it follows that

$$E(S, \overline{S}) \leq d\delta s < 2d\delta s.$$

Also, $M_{ij} = 1/2d$ for every edge $(i, j) \in E$. It follows that

$$\mathbb{1}_S^{\mathsf{T}} L \mathbb{1}_S \leq \frac{1}{2d} \cdot 2d\delta s = \delta s. \tag{7.4}$$

Combining Eq.s (7.2) and (7.4) above, we get that

$$\sum_i \alpha_i^2 \lambda_i \geq s - \delta s. \tag{7.5}$$

Recall that $1 = \lambda_1 \geq \lambda_2 \geq \ldots, \lambda_n$ are the eigenvalues of the random walk matrix M. We call the quantity $\sum_i \alpha_i^2$ the *coefficient sum of the eigenvalue set*. We also

call an eigenvalue λ_i (and the corresponding eigenvector $\vec{e}_i$) *heavy* if $\lambda_i \geq 1-8\delta$. We denote by H the index set of the heavy eigenvalues and let $\overline{H}$ be the index set of the rest. Since $\sum_i \alpha_i^2 \lambda_i \geq s - \delta s$ is large for a set with small conductance, we expect many of the coefficients α_i^2 corresponding to heavy eigenvalues to be large. This would slow down the convergence of the random walk and make the trap probability for our low-conductance set S large. The following claim establishes a lower bound on the contribution of the index set H to the coefficient sum.

Claim 7.13. *For $\{\alpha_i\}_i, H$, and s as defined above,*

$$\sum_{i \in H} \alpha_i^2 \geq \frac{7}{8} s.$$

Proof. Let x denote the coefficient sum of the set H of heavy eigenvalues: that is,

$$x := \sum_{i \in H} \alpha_i^2.$$

The following expression follows by the definition of the set H and its coefficient sum x:

$$\sum_i \alpha_i^2 \lambda_i = \sum_{i \in H} \alpha_i^2 \lambda_i + \sum_{i \in \overline{H}} \alpha_i^2 \lambda_i \leq x + (\sum_i \alpha_i^2 - x)(1-8\delta).$$

The second inequality above follows by upper bounding every λ_i with $i \in H$ by 1 and every λ_i with $i \in \overline{H}$ by $1-8\delta$. Recall that $\sum_i \alpha_i^2 = \langle \mathbb{1}_S, \mathbb{1}_S \rangle = s$ and we just proved that $\sum_i \alpha_i^2 \lambda_i \geq s - \delta s$ in (7.5). It follows that

$$s - \delta s \leq \sum_i \alpha_i^2 \lambda_i \leq x + (s-x)(1-8\delta).$$

Rearranging the inequality above, we get that $x \geq 7s/8$. □

Next, we use Claim 7.13 to get a lower bound on the average trap probability of set S in terms of the conductance of the cut $(S, \overline{S})$.

$$\text{trap}(S,\ell) = \frac{1}{s} \mathbb{1}_S^\mathsf{T} M^\ell \mathbb{1}_S = \frac{1}{s} (\sum_i \alpha_i \vec{e}_i)^\mathsf{T} M^\ell (\sum_i \alpha_i \vec{e}_i) = \frac{1}{s} (\sum_i \alpha_i \vec{e}_i)^\mathsf{T} (\sum_i \alpha_i \lambda_i^\ell \vec{e}_i) = \frac{1}{s} \sum_i \alpha_i^2 \lambda_i^\ell.$$

Furthermore, focusing on the contribution of the index set H to the trap probability,

$$\begin{aligned}\text{trap}(S,\ell) &= \frac{1}{s}\sum_i \alpha_i^2\lambda_i^\ell \\ &\geq \frac{1}{s}\sum_{i\in H} \alpha_i^2\lambda_i^\ell \\ &= \frac{1}{s}\left(\alpha_1^2\lambda_1 + \sum_{i\in H\setminus\{1\}} \alpha_i^2\lambda_i^\ell\right) \\ &\geq \frac{1}{s}\left(\alpha_1^2 + \left(\frac{7s}{8} - \alpha_1^2\right)(1-8\delta)^\ell\right). \end{aligned} \tag{7.6}$$

The last inequality follows by the definition of a heavy eigenvalue and by Claim 7.13 (we have that $\sum_{i\in H}\alpha_i\beta_i \geq 7s/8$). By definition, $\lambda_1 = 1$ and $\vec{e}_1 = [1/\sqrt{n}, 1/\sqrt{n}, \ldots, 1/\sqrt{n}]^\mathsf{T}$. It follows that

$$\alpha_1 = \langle \mathbb{1}_S, \vec{e}_1\rangle = s/\sqrt{n}.$$

Plugging in the values of α_1 in (7.6), we get

$$\begin{aligned}\text{trap}(S,\ell) &\geq \frac{1}{s}\left(\frac{s^2}{n} + \left(\frac{7s}{8} - \frac{s^2}{n}\right)(1-8\delta)^\ell\right) \\ &= \frac{s}{n} + \left(\frac{7}{8} - \frac{s}{n}\right)(1-8\delta)^\ell.\end{aligned}$$

Therefore, we can conclude that there exists a vertex $v \in S$ such that

$$\text{trap}(v,S,\ell) \geq \frac{s}{n} + \left(\frac{7}{8} - \frac{s}{n}\right)(1-8\delta)^\ell.$$

□

Next, we prove that every large enough subset $T \subset S$ has high trap probability. More specifically, we prove a lower bound on the probability of an ℓ-step random walk starting from a vertex chosen uniformly at random from T finishing at some vertex in T.

Lemma 7.14. *Consider sets $T \subseteq S \subseteq V$, $|S| = s$, $|T| = t$, such that the cut $(S,\overline{S})$ has conductance at most δ and that $t = (1-\eta)s$ for some $0 < \eta < 7/8$, then for any integer $\ell > 0$, the following holds*

$$\text{trap}(T,\ell) \geq \frac{t}{n} + \left(\frac{7}{8}\left(1 - \sqrt{\frac{8\eta}{7}}\right)^2 - \frac{t}{n}\right)(1-8\delta)^\ell.$$

Proof. Let $\mathbb{1}_S$ and $\mathbb{1}_T$ denote the n-dimensional indicator vectors of sets S and T, respectively. As in Lemma 7.12, we express $\mathbb{1}_S$ and $\mathbb{1}_T$ in the orthonormal basis defined by the eigenvectors of the random walk matrix M as $\mathbb{1}_S = \sum_i \alpha_i \vec{e}_i$ and $\mathbb{1}_T = \sum_i \beta_i \vec{e}_i$.

Since the conductance of the cut $(S, \overline{S})$ is at most δ, Claim 7.13 from Lemma 7.12 holds. We have that

$$\sum_i \alpha_i^2 \geq \sum_{i \in H} \alpha_i^2 \geq \frac{7}{8}s.$$

By the definition of $\mathbb{1}_S$ and $\mathbb{1}_T$, we have

$$\|\mathbb{1}_S - \mathbb{1}_T\|_2^2 = s - t = s - (1-\eta)s = \eta s.$$

Furthermore, the following follows from the expression of $\mathbb{1}_S$ and $\mathbb{1}_T$ in terms of the eigenvectors of the random walk matrix M

$$\|\mathbb{1}_S - \mathbb{1}_T\|_2^2 = \sum_i (\alpha_i - \beta_i)^2 = \eta s$$

Applying the triangle inequality, $\|\vec{a} - \vec{b}\| \geq \|\vec{a}\| - \|\vec{b}\|$ and upper bounding $\sqrt{\sum_{i \in H}(\alpha_i - \beta_i)^2}$ by $\sqrt{\sum_i (\alpha_i - \beta_i)^2} = \sqrt{\eta s}$, we get that

$$\begin{aligned}\sum_{i \in H} \beta_i^2 &\geq \left(\sqrt{\sum_{i \in H} \alpha_i^2} - \sqrt{\sum_{i \in H} (\alpha_i - \beta_i)^2}\right)^2 \\ &\geq \left(\sqrt{\frac{7}{8}s} - \sqrt{\eta s}\right)^2 = \frac{7}{8}s\left(1 - \sqrt{\frac{8\eta}{7}}\right)^2.\end{aligned}$$

Reasoning as in Lemma 7.12 and applying $\lambda_i \geq (1 - 8\delta)$, for all $i \in H$, we can bound the average trap probability over set T as

$$\begin{aligned}\text{trap}(T, \ell) \geq& \frac{1}{t}\sum_{i \in H} \beta_i^2 \lambda_i^\ell = \frac{1}{t}\left(\beta_1^2 \lambda_1 + \sum_{i \in H \setminus \{1\}} \beta_i^2 \lambda_i^\ell\right) \\ \geq& \frac{1}{t}\left(\frac{t^2}{n} + \left(\frac{7s}{8}\left(1 - \sqrt{\frac{8\eta}{7}}\right)^2 - \frac{t^2}{n}\right).(1 - 8\delta)^\ell\right) \qquad (7.7) \\ =& \frac{t}{n} + \left(\frac{7}{8} \cdot \frac{s}{t}\left(1 - \sqrt{\frac{8\eta}{7}}\right)^2 - \frac{t}{n}\right)(1 - 8\delta)^\ell.\end{aligned}$$

The second-to-last inequality follows from the definition of the first eigenvalue, eigenvector pair of the random walk matrix. By definition, $\lambda_1 = 1$ and $\vec{e}_1 =$

$[1/\sqrt{n}, 1/\sqrt{n}, \ldots, 1/\sqrt{n}]^{\mathsf{T}}$. Therefore, we have that $\beta_1 = \langle \mathbb{1}_T, \vec{e}_1 \rangle = t$. Since $T \subseteq S$, it follows that $s/t \geq 1$, the claim of the lemma follows by substituting s/t by its lower bound 1 in (7.7). □

Lemma 7.14 implies that any sufficiently large subset T of a low conductance set $S \subseteq V$ contains at least one vertex $v \in T$ such that a random walk starting from v ends in T with sufficiently high probability, i.e., v is sticky. More precisely, there exists a sticky vertex $v \in T$ such that

$$\text{trap}(v, T, \ell) \geq \frac{t}{n} + \left(\frac{7}{8} \left(1 - \sqrt{\frac{8\eta}{7}} \right)^2 - \frac{t}{n} \right) (1 - 8\delta)^{\ell}. \tag{7.8}$$

This leads to the following corollary as a result:

Corollary 7.15. *Consider a set $S \subset V$, $|S| = s$, such that the cut $(S, \bar{S})$ has conductance at most δ. Given any $0 < \eta \leq 7/8$ and integer $\ell > 0$, there exist at least ηs sticky vertices $v \in S$ for which there exists $T \subseteq S$ of size $t = (1-\eta)s$ such that*

$$\text{trap}(v, T, \ell) \geq \frac{t}{n} + \left(\frac{7}{8} \left(1 - \sqrt{\frac{8\eta}{7}} \right)^2 - \frac{t}{n} \right) (1 - 8\delta)^{\ell}.$$

Proof. Let $P \subseteq S$ denote the set of all the vertices for which eq (7.8) holds for some set T. One can extract P using the following iterative procedure. To begin with, we pick an arbitrary subset T of size $(1-\eta)s$ from S. By Lemma 7.14, there exists a vertex v in T with the desired trap probability. We remove v from S and add it to P. Let R be the set of remaining vertices of S. We then extract another subset T of size $(1-\eta)s$ from R and this process continues until we do not have sufficient number of vertices left in R. It is easy to see that $|R| \leq (1-\eta)s$ when this process ends. This implies that we have at least ηs vertices in P. □

Finally, we need the following classical relation between the conductance or Cheeger constant of a Markov chain and its second largest eigenvalue.

Theorem 7.16 ((70; 69; 76))**.** *Let P be a reversible lazy chain (i.e., for all x, $P(x,x) \geq 1/2$) with Cheeger constant ϕ_*. Let λ_2 be the second largest eigenvalue of P. Then,*

$$\frac{\phi_*^2}{2} \leq 1 - \lambda_2 \leq 2\phi_*.$$

We can now state our main theorem.

Theorem 7.17. *For an input graph $G = (V,E)$ with degree bound d, and parameters $0 < \alpha < 1$ and $\varepsilon > 0$, there exists a constant $C = C(d,\varepsilon) > 0$ such that* DISTRIBUTED-GRAPH-EXPANSION-TEST *in Algorithm 5*

- *outputs Accept, with high probability at least $2/3$, on every vertex of G if G is an α- vertex expander.*
- *outputs Reject, with probability at least $2/3$, on at least one vertex of G if G is ε-far from any $(C\alpha^2)$-expander.*

The algorithm uses $O_d(\log n)$ communication rounds.

Proof. Let us start by showing that, with high probability, the algorithm outputs *accept* on every vertex if G is an α-expander. By Observation 7.8, we may reject G and abort the algorithm due to congestion with probability at most e^{-16}. For now, let us assume that this event did not occur. Denote by λ_2 the second largest eigenvalue of the lazy random walk M on G. It is well known (see, e.g., (91)) that, for a pair $u,v \in V$,

$$\left| M^\ell(u,v) - \frac{1}{n} \right| \leq \lambda_2^\ell \leq e^{-\ell(1-\lambda_2)}.$$

Furthermore, it follows from Theorem 7.16 that

$$|M^\ell(u,v) - \frac{1}{n}| \leq e^{-\ell\phi_*^2/2} \leq e^{-\frac{\ell\alpha^2}{8d^2}},$$

where the second inequality above follows from the fact that for an α-expander, $\phi_* \geq \alpha/2d$ (see (7.1) in Section 7.2). Thus, for $\ell = (16d^2/\alpha^2)\log n$, any starting vertex $u \in V$ and a fixed vertex $v \in V$, in an α-expander, we have that

$$\frac{1}{n} - \frac{1}{n^2} \leq M^\ell(u,v) \leq \frac{1}{n} + \frac{1}{n^2}.$$

We fix $K = n^{1+\mu}$, for some $0 < \mu < 1$, as the number of the random walks from each starting vertex u. Let $X_{u,v}$ denote the number of random walks starting from u that ended in v. It follows that

$$\mathbb{E}X_{u,v} = K \cdot M^\ell(u,v) \leq n^\mu + n^{\mu-1}.$$

We also fix the rejection threshold τ in the algorithm to be $n^{\mu} + n^{2\mu/3} + 2$. The random variable $X_{u,v}$ is the sum of K independent Bernoulli trials with success probability $M^{\ell}(u,v)$. Applying multiplicative Chernoff bounds, we get

$$\begin{aligned}\Pr[X_{u,v} > (1+n^{-\mu/3}) \cdot \mathbb{E}[X_{u,v}]] &< \exp(-n^{-2\mu/3} \cdot (n^{\mu} + n^{\mu-1})/3) \\ &= \exp(-n^{\mu/3}(1+n^{-1})/3) \leq \frac{1}{4n},\end{aligned}$$

for large enough n. Thus, each vertex y receives at most

$$\begin{aligned}(1+n^{-\mu/3}) \cdot \mathbb{E}[X_{u,v}] &\leq (1+n^{-\mu/3}) \cdot (n^{\mu} + n^{\mu-1}) \\ &\leq n^{\mu} + n^{\mu-1} + n^{2\mu/3} + n^{2\mu/3-1} \\ &< n^{\mu} + n^{2\mu/3} + 2,\end{aligned}$$

with probability at least $1 - (1/4n)$, where last inequality follows from that $\mu \leq 1$. Taking union bound over all $y \in V$ and all starting vertices u, we get that, conditioned on the event that we did not abort due to congestion, with high probability, our algorithm outputs *Accept* on every vertex of G for every starting point if G is an α-expander. Further union bounding over the events that we rejected due to congestion or due to receiving too many walks at some endpoint, the claim follows.

Next, we show that our algorithm rejects, with high probability, any graph G that is ε-far from being an α^*-expander, for $\alpha^* = \Theta_d(\alpha^2)$. By Corollary 7.10, there exists a set $S \subset V$, with $s = |S| \geq \varepsilon n/12$, such that the conductance of S is less than $C\alpha^*$, for some $C > 0$. Applying Corollary 7.15 with $\eta = \frac{7}{8} \cdot \frac{1}{9^2}$ on S as above, we get that there exists a set P containing at least $7s/648$ vertices v for which there exists a set $T \subseteq S$, $|T| = t = \frac{641}{648}s$, such that

$$\text{trap}(v,T,\ell) \geq \frac{t}{n} + \left(\frac{56}{81} - \frac{t}{n}\right) \cdot (1 - 8C\alpha^*)^{\ell} \geq \frac{t}{n} + (\frac{56}{81} - \frac{t}{n}) \cdot e^{-16C\alpha^*\ell},$$

where the last inequality follows from that $1 - x > e^{-2x}$, for $0 \leq x < 1/2$, provided that $8C\alpha^* \leq 1/2$. For $\ell = (16d^2/\alpha^2)\log n$ and $t \leq s \leq n/2$, we get that, for every $v \in P$,

$$\text{trap}(v,T,\ell) \geq \frac{t}{n} + \frac{31}{162} \cdot \left(\frac{1}{n}\right)^{256C\alpha^* \frac{d^2}{\alpha^2}}.$$

For $\alpha^* \leq \alpha^2\mu/(1024 \cdot Cd^2)$, it holds that

$$\text{trap}(v,T,\ell) \geq \frac{t}{n} + \frac{31}{162} \cdot n^{-\mu/4}.$$

Let us assume that a vertex $u \in P \subset S$ is picked as the starting vertex of $K = n^{1+\mu}$ random walks in G. By Corollary 7.15 and above exposition, a set T with $t = (1-\eta)s$ with $\text{trap}(v,T,\ell) \geq \frac{t}{n} + \frac{31}{162} \cdot n^{-\mu/4}$ will exist for every $v \in P$. Furthermore, let $Y_{u,T}$ be the number of walks that ended in the set T (corresponding to u as in Corollary 7.15) after ℓ steps. It follows that

$$\mathbb{E}Y_{u,T} \geq n^{1+\mu} \cdot \left(\frac{t}{n} + \frac{31}{162} n^{-\mu/4}\right) \geq t \cdot n^{\mu} + \frac{31}{162} \cdot n^{1+3\mu/4}.$$

Note that the random variable $Y_{u,T}$ is the sum of K Bernoulli trials with success probability at least $(t/n) + (31/162) \cdot n^{-\mu/4})$. By an application of Chernoff bounds and using $t \leq n/2$, we get

$$\begin{aligned}
\Pr\left[Y_{u,T} < \left(1 - 3\sqrt{n^{-\mu}t^{-1}}\right)\mathbb{E}[Y_{u,T}]\right] &< \exp\left(-4n^{-\mu}t^{-1}\mathbb{E}[Y_{u,T}]\right) \\
&\leq \exp\left(-4n^{-\mu}t^{-1} \cdot \left(t \cdot n^{\mu} + \frac{31}{162} \cdot n^{1+3\mu/4}\right)\right) \\
&\leq \exp\left(-4(1 + \frac{31}{81} n^{-\mu/4})\right) \\
&< \frac{1}{10}.
\end{aligned}$$

Again using $t \leq n/2$, it follows that, on average, a vertex in T receives at least

$$\frac{(1 - 3\sqrt{n^{-\mu}t^{-1}}) \cdot \mathbb{E}[Y_{u,T}]}{t} \geq (1 - 3\sqrt{n^{-\mu}t^{-1}})(n^{\mu} + \frac{31}{81} n^{3\mu/4}) \geq n^{\mu} + \frac{31}{81} n^{3\mu/4} - \frac{5n^{\mu/2}}{\sqrt{t}},$$

with probability at least $9/10$. In that case, the average number of walks received by vertices in T is at least $n^{\mu} + \frac{31}{81} n^{3\mu/4} - O(1)$, which is above the threshold $\tau = n^{\mu} + n^{2\mu/3} + 2$ for large enough n. Thus, some vertex in S will receive more than t walks and it will output reject.

Let $\mathscr{E}$ be the event that none of the vertices in P is sampled to be one of the starting points in Q. Since each vertex $u \in V$ is sampled with probability $48/(\varepsilon n)$ and $|P'| \geq \varepsilon n/24$, it follows that

$$Pr[\mathscr{E}] \leq \left(1 - \frac{48}{\varepsilon n}\right)^{\frac{1}{24} \cdot \varepsilon n} \leq e^{-2} = 0.14$$

Taking a union bound over the probability of the event $\mathscr{E}$ and the probability of set T around a starting vertex $v \in P$ not receiving enough walks, we get that with probability at most $0.1 + 0.14 = 0.24$, no vertex will output *Reject*. Thus, our

distributed algorithm will output *Reject* with probability at least $2/3$, on at least one vertex of G.

Finally, the upper bound on the number of communication rounds follows from the length $\ell = \frac{16d^2}{\alpha^2} \log n$ of each random walk.

□

Bibliography

[1] Michael Elkin and Chhaya Trehan. Brief Announcement:(1+ε)-Approximate Shortest Paths in Dynamic Streams, In *proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC 2022.

[2] Michael Elkin and Chhaya Trehan. (1+ε)-Approximate Shortest Paths in Dynamic Streams. In *proceedings of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM 2022.

[3] Oded Lachish, Felix Reidl, and Chhaya Trehan. When You Come at the King You Best Not Miss. In *proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS 2022.

[4] Tugkan Batu and Chhaya Trehan. A Distributed Conductance Tester Without Global Information Collection. *CoRR*, 2023. *CoRR*, abs/2305.14178, 2013.

[5] Amir Abboud and Greg Bodwin. The 4/3 additive spanner exponent is tight. In *proceedings of the ACM Symposium on Theory of Computing*, STOC 2016.

[6] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2012.

[7] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: Sparsification, spanners, and subgraphs. In *Proceedings of the ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS 2012.

[8] Ingo Althofer, Gautam Das, David Dobkin, and Deborah A Joseph. Generating sparse spanners for weighted graphs. In *proceedings of Scandinavian Workshop on Algorithm Theory*, SWAT 1990.

[9] Khanh Do Ba, Piotr Indyk, Eric Price, and David P. Woodruff. Lower bounds for sparse recovery. In *proceedings of the ACM-SIAM Symposium on Discrete Algorithms* SODA 2010.

[10] Antal Balog and Imre Bárány. On the convex hull of the integer points in a disc. In *proceedings of the Symposium on Computational Geometry*, SoCG 1991.

[11] Surender Baswana. Streaming algorithm for graph spanners - single pass and constant processing time per edge. *Information Processing Letters*, 2008.

[12] Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *proceedings of the International Symposium on Distributed Computing*, DISC 2017.

[13] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 1979.

[14] Yi-Jun Chang, Martin Farach-Colton, Tsan-sheng Hsu, and Meng-Tsung Tsai. Streaming complexity of spanning tree computation. In *proceedings of the International Symposium on Theoretical Aspects of Computer Science*, STACS 2020.

[15] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *proceedings of the Symposium on Theory of Computing*, STOC 1994.

[16] Don Coppersmith and Michael Elkin. Sparse sourcewise and pairwise distance preservers. *SIAM Journal on Discrete Mathematics*, 2006.

[17] Graham Cormode and D. Firmani. A unifying framework for ℓ-0-sampling algorithms. *Distributed and Parallel Databases*, 2013.

[18] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Transactions on Algorithms*, 2011.

[19] Michael Elkin. Distributed exact shortest paths in sublinear time. In *proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017.

[20] Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Improved weighted additive spanners. *Distributed Computing*, 2023.

[21] Michael Elkin and Shaked Matar. Near-additive spanners in low polynomial deterministic congest time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC 2019.

[22] Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. *ACM Transactions on Algorithms*, 2018.

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

[24] Michael Elkin and Ofer Neiman. Centralized and parallel multi-source shortest paths via hopsets and fast matrix multiplication. In *proceedings of International Symposium on Theoretical Aspects of Computer Science*, STACS 2022.

[25] Michael Elkin and Ofer Neiman. Near-additive spanners and near-exact hopsets, A unified view. *Bulletin of EATCS*, 2020.

[26] Michael Elkin and David Peleg. $(1+\varepsilon,\beta)$-spanner constructions for general graphs. In *proceedings of the ACM Symposium on Theory of Computing*, STOC 2001.

[27] Michael Elkin and Shay Solomon. Fast constructions of light-weight spanners for general graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2013.

[28] Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\varepsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 2006.

[29] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *proceedings of the International Colloquium on Automata, Languages and Programming*, ICALP 2004.

[30] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 2009.

[31] Manuel Fernandez, David P. Woodruff, and Taisuke Yasuda. Graph spanners in the message-passing model. In *proceedings of the ACM conference on Innovations in Theoretical Computer Science Conference, ITCS 2020.*

[32] Arnold Filtser, Michael Kapralov, and Navid Nouri. Graph Spanners by Sketching in Dynamic Streams and the Simultaneous Communication Model. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2021.*

[33] S. Ganguly. Counting distinct items over update streams. Theoretical Computer Science*, 2007.*

[34] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. In proceeding of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2013.*

[35] D. R. Heath-Brown. Almost-primes in arithmetic progressions and short intervals. Mathematical Proceedings of the Cambridge Philosophical Society*,1978.*

[36] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In proceedings of the International Colloquium on Automata, Languages, and Programming*, ICALP 2015.*

[37] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source

shortest paths. In Proceedings of the ACM Symposium on Theory of Computing*, STOC 201616.*

[38] Piotr Indyk, Eric Price, and David P. Woodruff. On the power of adaptivity in sparse recovery. In IEEE Symposium on Foundations of Computer Science*, FOCS 2011.*

[39] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems *, PODS 2011*

[40] Michael Kapralov and David Woodruff. Spanners and sparsifiers in dynamic streams. In Proceedings of the Symposium on Principles of Distributed Computing*, PODC 2014.*

[41] Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC 2015.*

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

[43] Felix Lazebnik and Vasiliy A. Ustimenko. Some algebraic constructions of dense graphs of large girth and of large size. In Expanding Graphs, Proceedings of a DIMACS Workshop*, volume 10 of* DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS/AMS, 1992.*

[44] Andrew McGregor. Graph stream algorithms: A survey. SIGMOD Record*, 2014.*

[45] Morteza Monemizadeh and David P. Woodruff. 1-pass relative-error l_p-sampling with applications. In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2010.*

[46] Noam Nisan. Pseudorandom generators for space-bounded computation. Combinatorica*, 1992.*

[47] David Peleg and Alejandro A. Schaffer. Graph spanners. Journal of Graph Theory*, 1989.*

[48] P. Pollack and Enrique Treviño. Finding the four squares in lagrange's theorem. Integers*, 2018.*

[49] Michael O. Rabin and Jeffery O. Shallit. Randomized algorithms in number theory. Communications on Pure and Applied Mathematics*, 1986.*

[50] Vojuch Jarnik. Mathematische Zeitschrift. Uber die gitterpunkte auf konvexen kurven, 1926.

[51] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In proceedings of the International Workshop on Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques *, RAMDOM/APPROX, 2013.*

[52] Ashish Goel, Michael Kapralov, and Ian Post. Single pass sparsification in the streaming model with edge deletions. CoRR*, abs/1203.4900, 2012.*

[53] Jonathan A. Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. In proceedings of the International Symposium on Theoretical Aspects of Computer Science*, STACS 2011.*

[54] Sepehr Assadi, Sanjeev Khanna, and Yang Li. Tight bounds for single-pass streaming complexity of the set cover problem. In proceedings of the ACM Symposium on Theory of Computing*, STOC 2016.*

[55] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2017.*

[56] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2016.*

[57] Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via Sampling with Applications to Finding Matchings

and Related Problems in Dynamic Graph Streams. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2016.*

[58] Jelani Nelson and Huacheng Yu. Optimal lower bounds for distributed and streaming spanning forest computation. In proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA 2019.*

[59] Kook Jin Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. In proceedings of the International Colloquium on Automata, Languages and Programming*, ICALP 2009.*

[60] Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. Sorting and selection with imprecise comparisons. In proceedings of International Colloquium on Automata, Languages and Programming *, ICALP 2009 .*

[61] Ramachandran Balasubramanian, Venkatesh Raman, and G Srinivasaragavan. Finding scores in tournaments. Journal of Algorithms*,1997.*

[62] Arindam Biswas, Varunkumar Jayapaul, Venkatesh Raman, and Srinivasa Rao Satti. The complexity of finding (approximate sized) distance-d dominating set in tournaments. In proceedings of International Workshop on Frontiers in Algorithmics*, FAW 2017.*

[63] Palash Dey. Query complexity of tournament solutions. In Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI 2017.*

[64] Dishant Goyal, Varunkumar Jayapaul, and Venkatesh Raman. Elusiveness of finding degrees. Discrete Applied Mathematics*, 2020.*

[65] HG Landau. On dominance relations and the structure of animal societies: III the condition for a score structure. The bulletin of mathematical biophysics*,1953.*

[66] John W Moon. Topics on tournaments in graph theory. *Courier Dover Publications, 2015.*

[67] Jian Shen, Li Sheng, and Jie Wu. Searching for sorted sequences of kings in tournaments. SIAM Journal of Computing*, 2003.*

[68] Yehuda Afek, Shay Kutten, and Moti Yung. The local detection paradigm and its application to self-stabilization. Theoretical Computer Science*, 1997.*

[69] Noga Alon. Eigenvalues and expanders. Combinatorica*, 1986*

[70] Noga Alon and V. D. Milman. Eigenvalues, expanders and superconcentrators (extended abstract). In proceedings of Symposium on Foundations of Computer Science *, IEEE, FOCS 1984.*

[71] Baruch Awerbuch and George Varghese. Distributed program checking: A paradigm for building self-stabilizing distributed protocols (extended abstract). In proceedings of Symposium on Foundations of Computer Science *, IEEE, FOCS 1991.*

[72] Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. Distributed Computing*, 2009.*

[73] Fan R. K. Chung. Spectral Graph Theory. *Providence, RI, 1997.*

[74] Artur Czumaj and Christian Sohler. Testing Expansion in Bounded-Degree Graphs. Combinatorics, Probability & Computing*, 2010.*

[75] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In Proceedings of the Annual Symposium on Theory of Computing*, STOC 2011.*

[76] Jozef Dodziuk. Difference equations, isoperimetric inequality and transience of certain random walks. Transactions of the American Mathematical Society*, 1984.*

[77] Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three Notes on Distributed Property Testing. In International Symposium on Distributed Computing*, DISC 2017.*

[78] Hendrik Fichtenberger and Yadu Vasudev. A two-sided error distributed property tester for conductance. In International Symposium on Mathematical Foundations of Computer Science *, MFCS 2018.*

[79] Orr Fischer, Tzlil Gonen, and Rotem Oshman. Distributed Property Testing for Subgraph-Freeness Revisited. CoRR*, abs/1705.04033, 2017.*

[80] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In Proceedings of Conference on Foundations of Computer Science*, SFCS 1996.*

[81] Oded Goldreich. Introduction to Testing Graph Properties*, pages 105–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.*

[82] Oded Goldreich. Introduction to Property Testing. *Cambridge University Press, 2017.*

[83] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. In Proceedings of the ACM Symposium on Theory of Computing*, STOC 1997.*

[84] Oded Goldreich and Dana Ron. On testing expansion in bounded-degree graphs. Electronic Colloquium on Computational Complexity*, 2000.*

[85] Satyen Kale and C. Seshadhri. An expansion tester for bounded degree graphs. SIAM Journal of Computing*, 2011.*

[86] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. Distributed Computing and Networking*, 2013.*

[87] Jakub Łkacki, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Walking randomly, massively, and efficiently. In Proceedings of ACM Symposium on Theory of Computing*, STOC 2020.*

[88] Asaf Nachmias and Asaf Shapira. Testing the expansion of a graph. Information and Computation*, 2010.*

[89] David Peleg. Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.*

[90] Seth Pettie and Hsin-Hao Su. Distributed coloring algorithms for triangle-free graphs. In proceedings of International Colloquium on Automata, Languages, and Programming*, ICALP 2013.*

[91] Alistair Sinclair. Algorithms for Random Generation and Counting: A Markov Chain Approach. *Birkhauser Verlag, CHE, 1993.*