Abstract

Massive Data Streams in Graph Theory and Computational Geometry

Jian Zhang

2005

Streaming is an important paradigm for handling data sets that are too large to fit in main memory. In the streaming computational model, algorithms are restricted to using much less space than they would need to store the input. The massive data set is accessed in a sequential fashion and, therefore, can be viewed as a stream of data elements. The order of the data elements in the stream is not controlled by the algorithm. There are three important resources considered in the streaming model: the size of the workspace, the number of passes that the algorithm makes over the stream, and the time to process each data element in the stream.

In this thesis, we study computational-geometry problems and graph problems in the streaming model. We design algorithms for computing diameter in the streaming and the sliding-window models and prove some corresponding lower bounds. The sliding-window model is a variation of the streaming model in which only more recent data elements in the stream are considered. Previously, work in the streaming model has focused on algorithms that use polylog space. For graph problems, this restriction is too strong. We investigate the computational power of the model with different space restrictions. For a graph of n vertices, our lower bounds show that not many interesting computations can be done when the space restriction is $o(n)$. We also show that, by using $n \cdot \text{polylog}(n)$ space, one can design algorithms that provide approximations to some basic problems, *e.g.*, the all-pairs, shortest-path distance problem. We present two main algorithms that approximate these distances. We also exhibit tradeoffs between the accuracy of the results and the resources used by the algorithms.

Massive Data Streams in Graph Theory and Computational Geometry

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Jian Zhang

Dissertation Director: Joan Feigenbaum

Dec. 2005

Copyright © 2005 by Jian Zhang

All rights reserved.

Contents

Table of Contents i

List of Figures iii

Acknowledgments iv

1 Introduction 1

2 Preliminaries 12

2.1 Computational Model 12

2.1.1 Massive Data Streams 12

2.1.2 Streaming Computational Model 14

2.2 Algorithm Design Approaches and Analysis Tools 16

2.2.1 Synopses and Sketches 17

2.2.2 Probability Bounds 19

2.3 Streaming and Communication Complexity 20

2.4 Streaming Annotation 23

3 Massive Data Streams in Computational Geometry 28

3.1 Introduction 28

3.2 Sector-Based Diameter Approximation in the Streaming Model 31

3.3 Maintaining the Diameter in the Sliding-Window Model 35

3.4 Lower Bounds 49

3.5 Closest Pair and K-Promised Convex Hull 54

3.6 Conclusions 56

4 Massive Data Streams in Graph Theory 58

4.1 Massive Graphs and Streaming 58

4.2 The Shortest-Path-Distance Problem and Graph Spanners 62

4.3 Distance Approximation in Multiple Passes 68

4.3.1 Graph Spanners and Distributed Computing 68

4.3.2 Streaming Spanner Construction 71

4.3.3 Fast Distributed Spanner Construction 72

4.3.4 Analysis of Time and Message Complexity 81

4.3.5 Parallel Implementation 85

4.3.6 Adaptation to the Streaming Model 86

4.3.7 Conclusion 90

4.4 Distance Approximation in One Pass 92

4.4.1 Cluster Structures for Distance Approximation 93

4.4.2 One-Pass Spanner Construction 97

4.4.3 Analysis 99

5 Conclusion and Future Work 106

5.1 Conclusion 106

5.2 Open Problems 107

5.3 Future Work: From Theory to Applications 108

Bibliography 115

List of Figures

3.1 Two examples of sectors 32

3.2 Rounding points in each interval 36

3.3 Clusters built for the first window 40

3.4 A point may be moved in each rounding, but all the displacements are in the same direction. 44

3.5 Reduction from DISJ to diameter 50

3.6 Convex hull using only one point on the circles 56

4.1 The distance between x and y is 6 if there is no common neighborhood of x and y. Otherwise (if the dash-dotted edge exists), the distance is 2. 61

4.2 Example of clusters in covers. 79

Acknowledgments

Above all the people who have helped me to make it through the graduate school, I would like to thank my adviser, Joan Feigenbaum. Her insightful suggestions and constant encouragement were essential for this dissertation to become possible. Moreover, she is a mentor to me in many ways. She taught me not only how to conduct research in computer science, but also how to further develop my scholarly career.

I am grateful to my co-adviser Sampath Kannan. I benefited greatly from many discussions with him. His ideas and advice helped me to find solutions to research problems as well as the path to become a researcher. I am also thankful to S. Muthu Muthukrishnan and Jennifer Rexford for their helpful advice.

While working on this dissertation, I was very fortunate to collaborate with such brilliant people as Michael Elkin, Andrew McGregor, and Sid Suri. It has been a great pleasure to work with them and they are my friends as well as my collaborators. I am thankful for many enjoyable hours spent brainstorming with them on difficult theorems.

I am also thankful to my friends and fellow students in the department, who have made my time at Yale colorful.

Support for my PhD study was provided primarily by the National Science Foundation through grants 0105337 and 0331548. Travel support was provided by the Office of Naval Research through grant N00014-01-1-0795.

Finally, I must thank my parents. Their love helped me to get through many difficulties. Without their support, none of this would be possible.

Chapter 1

Introduction

In recent years, massive data sets have emerged in many applications. In particular, applications that monitor the operations of large systems produce huge amounts of data. For example, sale records are collected every day in supermarkets. Phone companies keep track of calling records. Some network routers, such as the Cisco router with the NetFlow feature [22], are capable of producing network traffic reports continuously.

New challenges arise when computing with massive data sets. First, one may need to redefine what is an efficient computation. The time complexity of a computation is the amount of time required to complete that computation. It is a function $f(n)$ of the input's size n. Traditionally, an algorithm is considered to be efficient when $f(n)$ is a polynomial of n. In practice, this works fine with fairly large input sizes. However, with extremely large input sizes, an algorithm with polynomial time complexity whose polynomial contains terms of high orders becomes prohibitively inefficient. It is reasonable to expect that computer performance will be improved greatly in the future. It is also reasonable to expect that the "massive" data sets in the future will be much more "massive" than the ones we encounter now, especially

given the trends in the expansion of information systems and the development of sophisticated automatic data-collecting tools. The computation challenge may remain. In fact, the massive data sets in data-warehouse mining and Web searching we see today were not encountered a decade ago. Another example is provided by the Internet. The Internet has grown so fast that many problems that were not foreseen in the original design have arisen. Measurements of different aspects of the Internet are considered invaluable for troubleshooting the system and for improving the design of the system. In this case, the amount of measurement data is enormous. The networking community has seen an emerging problem: although a large amount of data is collected, it is of little use in providing needed information. The reason is simply that the current data analysis methods, particularly those that incorporate certain levels of sophistication, do not work well with massive data sets because of their high time complexity. One is left with trivial analyses, which apparently does not reveal much useful information. There are increasing demands for near-linear or sublinear data-analysis algorithms.

The second challenge comes from storage devices used for massive data sets. Inexpensive devices can provide large storage capacity. However, such a large capacity comes with a price: random accesses on these devices are very slow. The storage devices in a computer system are diversifying in two directions: one that is fast but has relatively small capacity and the other that is slow but has extremely large capacity. The memory hierarchy of a computer reflects such a difference. At the top of the hierarchy is the main memory (RAM), which is the primary storage device in a computer. The main memory is very fast but its capacity is rather small. Below the main memory are the secondary storage devices, such as hard disks. The secondary storage devices have medium capacity and a medium access speed. At the bottom of the hierarchy are massive storage devices such as CDs and tapes, which

have tremendous capacity but an extremely slow access speed. (See, *e.g.*, [73] for a detailed description.)

The differences between storage devices are not considered in the traditional computational model. In such a model, a computer consists of two major components, the control and the memory space. The traditional model treats all storage devices indifferently and abstracts (the union of) them as a single memory space. All the memory cells in this space are viewed equally—that is, all the basic operations on this memory space, such as access to a random location, take an equal (constant) amount of time. Because of the diversification of the memory, computational complexities derived from the traditional model in many cases do not reflect the complexity of the real computation on a real machine.

To consider the difference between memory types, the external memory model [77] was proposed. This model is similar to the traditional computational model except that memory is now divided into two types: internal (main memory) and external (disks). Despite the fact that access to the external memory is slow, the computation can still use the whole memory space (external + internal). That is, the computation can make unlimited random access to the external memory as well as to the internal memory. The fast but small internal memory is often used as a cache for data on the external memory, and localities in these data are often exploited to minimize the number of accesses to the external memory.

The external memory in this model often refers to the secondary storage devices. When dealing with even larger data sets, one may need to go further down the memory hierarchy and use massive storage devices like tapes. At this point, unlimited random access to the data becomes prohibitively inefficient. Also, in some cases, it may be impossible to store the data at all. For example, in the Internet routing system, the routers forward packets at such a high speed that there may not be

enough time for them to dump the detailed information of the packets into slow storage devices.

Therefore, we need computational models that divide the memory space of the traditional model into two types: workspace and storage space. The workspace is the memory where the computation performs frequent random accesses. The storage space is the memory where the inputs to the computation are kept. The computation uses a comparably small workspace and needs only a few random accesses to the storage space. In the extreme, the computation accesses the storage space in a completely sequential fashion. Furthermore, we may also require that in these models, the computation be done in at most near-linear time.

Towards this goal, two types of models have been intensively studied: sampling and streaming. Both types of models use a workspace that is much smaller than the input size. The difference between them is that they allow different types of access to the storage space. In the sampling model, the computation is allowed to perform only a few random accesses to the inputs. With these accesses, it takes samples from the input and then computes on these samples. The main limitation of this model is that the computation does not see the whole input. Therefore, in some cases, it needs to take a rather large sample or can provide only rather relaxed approximate results. In the streaming model, the computation is allowed to access the storage space only in a sequential fashion. On the other hand, access to the whole storage space in such a fashion is granted. Therefore, the computation can see the whole input. Note that a non-adaptive sampling algorithm can be trivially transformed into a streaming algorithm. However, an adaptive sampling algorithm that takes advantage of the random access allowed by the sampling model cannot be directly adapted into the streaming model.

The streaming model was formally proposed in [52]. However, before [52], there

were studies [54, 43] that considered similar models, without using the term streaming. Since [52], there has been a large body of work on algorithms and complexity in this model [6, 39, 55, 50, 48, 68, 57]. The majority of the work on streaming algorithms considered problems that fall into a small number of categories, such as computing statistics, norms, and histograms. To further the understanding of the streaming model, in this dissertation we explore new directions and problems that are of significance for streaming computation in general. In particular, we study streaming problems in two main directions: computational-geometry problems and graph-theoretic problems.

Geometric data streams arise naturally in applications that involve monitoring or tracking entities carrying geographic information. For example, Korn *et al.* [60] discuss applications such as decision-support systems for wireless-telephony access. The customers using the service generate streams of data about their locations, and the cell-phone company may want to process and query the streams for various decision-support purposes. Data streams generated by sensor nets and other observatory facilities provide another example [27]. In these applications, the data streams consist of the records of geographically dispersed events, and computing extent measures (such as the diameter) is an important component of monitoring the spatial extent of these events.

In joint work with Joan Feigenbaum and Sampath Kannan, I studied the diameter problem in both the streaming model and one of its variations: the sliding-window model [28]. In the sliding-window model, the data stream is infinite but the computation is concerned only with recent data. A sliding window that includes the most recent data items is imposed on the stream, and the computation is done on data within this window. (We give a detailed introduction to the sliding-window model in Chapter 2.) We investigate the two-dimensional diameter problem in these two

models. Given a set of points P, the diameter is the maximum, over all pairs x, y in P, of the distance between x and y. There are efficient algorithms to compute the exact diameter [23, 71] or to approximate the diameter [2, 3, 12, 18] in the traditional computational models. However, at the time of our work, little was known on computing diameters in the streaming model.

In this work, we show that computing the exact diameter for a set of n points in the streaming model or maintaining it in the sliding-window model (with window width n) requires $\Omega(n)$ bits of space. However, when approximation is allowed, we present a simple ϵ-approximation algorithm in the streaming model that uses $O(1/\epsilon)$ space and processes each point in $O(\log(1/\epsilon))$ time. In the sliding-window model, we also present an ϵ-approximation algorithm to maintain the diameter in 2-d using $O(\frac{1}{\epsilon^{3/2}} \log^3 n(\log R + \log\log n + \log \frac{1}{\epsilon}))$ bits of space, where R is the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two points in the window. Both of our algorithms make only one-sided errors. That is, the output $\hat{D}$ of our algorithms ϵ-approximates the true diameter D in that $D \geq \hat{D} \geq (1-\epsilon)D$. We also give streaming-space lower bounds to the closest-pair problem and the k-promised convex hull problem.

Our work initiated the study of computational geometry in the streaming and sliding-window models. We also provided a useful approach for the design of geometry algorithms in the streaming model. That is, instead of building data structures for the points, the algorithm can divide the space into sectors/grids, therefore transforming the geometry problems into counting problems in these sectors/grids. This approach was further extended by [27, 19].

The second direction of study in this dissertation is graph-theoretic problems in the streaming model. Massive graphs arise naturally in many real-world scenarios. Two examples are the *call graph*, where nodes correspond to telephone numbers

and edges to calls between numbers that call each other during some time interval, and the *Web graph*, where nodes are web pages, and the edges are links between pages. Massive dense graphs also appear in applications such as structured data mining, where the relationships among the data items in the data set are modeled as graphs. Because many application problems can be modeled as graph problems, it is important to study massive graphs in the streaming model.

In joint work with Joan Feigenbaum, Sampath Kannan, Andrew McGregor, and Sid Suri, I studied graph problems in the streaming model, where a graph is presented as a stream of edges. In particular, we considered graph-distance problems such as the graph-diameter problem and the all-pairs, shortest-path-distance problem.

In this work, we examine graph problems with a more general view of the streaming model. The two salient features of the streaming model are that the workspace used by the computation is much smaller than the input size and that the input data are accessed in sequential order. These two features give rise to two complexity measures of the streaming model—namely, the amount of workspace and the number of passes through the input data. Most previous work on the streaming model focused on the problems that can be solved in one pass using polylogarithmic space. These studies explored a restricted streaming model. To fully understand the computational power of the model, it is necessary to consider problems that may require multiple passes and/or a larger workspace. Graph problems are of this type. A simple graph problem, such as determining whether the distance between two vertices x and y is 2, if computed in a constant number of passes, requires a workspace as much as the number of vertices in the input graph [16]. To better understand graph-theoretic problems in the streaming model, we need to consider a full spectrum of the workspace size and the number of passes.

Consider the spectrum of the workspace size among the algorithms that make

sequential (one-way) access to the input data. At one extreme of the spectrum, we have dynamic algorithms [36]. These algorithms can use memory large enough to hold the whole input. At the other extreme, we have the type of streaming algorithms that use only polylogarithmic space. It has been suggested by Muthukrishnan [68] that the middle ground, where, for a graph with n vertices, the algorithms use $n \cdot \text{polylog}(n)$ bits of space, is an interesting open area. This is the main area that we explore in this work. To our best knowledge, our work is the first study on massive graphs in this area of the streaming model. Note that our algorithms can store the vertex set, but not the edge set. (For a dense graph, this amount of space is sublinear in the input size.) In [1], the authors studied graph problems in a semi-external model, in which the vertex set of the graph can be stored in the main memory, but the edge set cannot. The difference between their model and ours is that their model is based on the external memory model. In their model, random access to the edges, although expensive, is still allowed. As argued in [61], this is a major drawback when computing on massive graphs. Our algorithm does not have this problem.

In the spectrum of the number of passes, we considered computations that allow multiple passes through the input stream. Clearly, too many passes will be prohibitively inefficient in this model. Therefore, we restrict our consideration only to computations that use a constant number of passes. Very few studies [30, 16] have explored this area of the streaming model.

We consider graph-distance computation with different combinations of the two parameters. Intuitively, computing graph distances is difficult in the streaming model because many such computations employ construction of BFS (breadth first search) trees or similar subroutines. In order to achieve accurate distance computation or approximation, the depth of the BFS trees is quite large. Clearly, BFS-tree construction is not suitable for streaming. The growth of one layer of the tree, in the

worst-case scenario, may need one pass through the stream. Hence, to construct a tree with depth more than a constant number requires too many passes through the stream. In this work, we show that despite the seemingly stringent restrictions of the streaming model, graph distances can be approximated using sublinear space in one or a constant number of passes.

For a graph with n vertices, we first provide a simple one-pass algorithm [37] that gives $(\log n/\log\log n)$-approximations for diameter and all-pairs, shortest-path distances in unweighted graphs. The algorithm uses $n \cdot \text{polylog}(n)$ space and constructs a $(\log n/\log\log n)$-spanner for the input graph. A k-spanner H of a graph G is an edge subgraph of G such that for any pair of vertices their shortest-path distance in H is at most k times their shortest-path distance in G. Therefore, the distances in the input graph can be approximated by the spanner.

The drawback of this algorithm is that, in the worst case, it needs $O(n)$ time to process each edge in the input stream. Such a per-edge processing time is prohibitive in the streaming model. In [38], we devise a better algorithm that improves the per-edge processing time. We show a randomized streaming algorithm that constructs a $(2t+1)$-spanner for an unweighted, undirected graph in one pass. With probability $1 - \frac{1}{n^{\Omega(1)}}$, the algorithm uses $O(t \cdot n^{1+1/t} \log^2 n)$ bits of space and processes each edge in the stream in $O(t^2 \cdot n^{1/t} \log n)$ time. Using this spanner, the diameter as well as the all-pairs distances in the graph can be $(2t+1)$-approximated. Note that for $t = \log n/\log\log n$, the algorithm uses $n \cdot \text{polylog}(n)$ bits of space and processes each edge in $\text{polylog}(n)$ time. This algorithm can be extended to construct $((1+\epsilon)\cdot(2t+1))$-spanners for weighted, undirected graphs. A complementary result is given in [38] that shows, with $O(n^{1+1/t})$ space, we cannot approximate the distance between u and v better than by a factor t even if we know the vertices u and v. Therefore, our algorithm is quite close to optimal.

To explore the area of the streaming model in which multiple passes through the stream are allowed, I studied, in a joint work with Michael Elkin, spanner constructions using multiple passes. We give a streaming algorithm for constructing $(1+\epsilon, \beta)$-spanners using only a *constant number of passes* through the input. The algorithm uses $O(n^{1+1/\kappa} \cdot \log n)$ bits of space, and $O(n^{\rho})$ processing time-per-edge, where $\kappa = 1, 2, \ldots, \quad 0 < \epsilon < 1$, and $0 < \rho < 1$ are control parameters. β is a function of ϵ and κ. For a graph $G(V, E)$ and two vertices $x, y \in V$, let $d_G(x, y)$ be the shortest-path distance between x and y in the graph G. A subgraph $H = (V, E')$ is a (ϵ, β)-*spanner* of G if, for any vertices $x, y \in V$, $d_G(x, y) \leq d_H(x, y) \leq (1+\epsilon) \cdot d_G(x, y) + \beta$. (Hence, H is also called an "additive spanner.") Again, our streaming construction of the additive spanner leads directly to a streaming algorithm with the same complexity parameters (number of passes and space) that approximates all-pairs, shortest-path distances. Note that for large distances, an additive spanner provides a better approximation. Hence, while this algorithm uses more passes (though still a constant number) than the above algorithm, it provides *far shorter* paths and distance estimates. This demonstrates a certain tradeoff between the accuracy of the approximation and the number of passes.

Our work initiates the study of graph problems, particularly the graph-distance problems in a more general streaming model. By considering algorithms using multiple passes and larger than polylogarithmic, but still sublinear, space, we extended the understanding of streaming computations. Our algorithms are the first streaming constructions of spanners. Because our one-pass algorithm provides novel spanner construction that avoids BFS-like explorations. it can also be applied to incremental computations

The joint work with Joan Feigenbaum and Sampath Kannan on computing diameter in the streaming and sliding-window models first appeared in [41]. The joint

work with Joan Feigenbaum, Sampath Kannan, Andrew McGregor, and Sid Suri on massive graphs first appeared in [37] and [38]. The joint work with Michael Elkin on approximating graph distances using additive spanners first appeared in [35].

Chapter 2

Preliminaries

2.1 Computational Model

2.1.1 Massive Data Streams

A *data stream* is a sequence of data elements $a_1, a_2, \ldots, a_n$ from a finite set M. The semantics of the data element σ_i may be different from application to application. This leads to different models of data streams. In the most common case, the input to the computation consists of a set, *e.g.*, a set of points, a set of numbers. With such an input, the data stream is simply a sequence of the elements in the input set. But the order in which the elements appear in the sequence can be arbitrary. For example, if the input to the computation is a set of points, an arbitrary sequence of these points then forms the input data stream. Sometimes the input itself is already a sequence. For example, many data streams are also time series where the i-th entry in the sequence is some measurement at time i. In this dissertation, when referring to streams, we mean a data stream like the above ones.

There is another type of data stream [68], in which an implicit array A of size n is involved. The data elements in the stream may take the form (i, k) where $i \in [n]$

and $k \in \mathbb{Z}$. The semantics of such an element state that $A[i] = A'[i] + k$, where $A[i]$ is the value of the i-th entry of A after seeing this data element and $A'[i]$ is the value of that entry before seeing the element. These semantics lead to models of data streams such as the cash-register model and the turnstile-model [68]. The difference between the two models is that, in the cash-register model $k \geq 0$, while in the turnstile model, k could be negative. Because the data elements in this type of streams stand for updates, one can view problems in these models as dynamic problems, where the input is a sequence of updates to an implicit set or array and the goal of the computation is to compute some function over the *current values* of the set or array.

Most streaming problems considered in this dissertation can be viewed as partial dynamic problems, *i.e.*, data elements are added to the current set, but not deleted. The only model we considered that has the full dynamic property is the sliding-window model. The sliding-window model was introduced in [28]. In this model, the data stream $a_1, a_2, \ldots$ may be infinite, and one is interested only in the n most recent data elements. Suppose a_i is the current data element. A sliding window of size n then consists of elements $\{a_{i-n+1}, a_{i-n+2}, \ldots, a_i\}$. When new elements arrive, old elements at the end of the window expire. If we view the data stream as an infinite string, the window is a substring of the stream.

In the turnstile model, to delete a data element, *i.e.*, to set the entry $A[i]$ to be zero, $(i, -A[i])$ is explicitly added in the stream. In the sliding-window model, the oldest data element is implicitly deleted when the window slides forward. Because a sliding-window algorithm is not given enough space to store all the data elements in the window, this implicit deletion raises a challenge for algorithm design.

2.1.2 Streaming Computational Model

Having defined data streams, we are now ready to give the definition of our streaming computational model. Besides the streaming model given in [52], there are several other streaming or streaming-like models [51, 30]. We try to capture the common features of these models and give a more general definition. The other streaming models can be viewed as restricted versions of our model.

A *streaming algorithm* is an algorithm that computes some function over a data stream and has the following properties:

1. The input to the streaming algorithm is a data stream as defined in the previous section. Note that in this dissertation, we mainly consider two models of streams. One is the data stream that is an arbitrary sequence of the input data set. The other is the sliding-window model. In both cases, we denote by n the length of the stream or the size of the window.

2. The streaming algorithm accesses the data elements in the stream in sequential order. The order of the data elements in the stream is not controlled by the algorithm.

3. The algorithm uses a workspace much smaller than the input size. It can perform unrestricted random access in the workspace. The amount of workspace required by the streaming algorithm is an important complexity measure of the algorithm.

4. As the input data stream by, the algorithm needs to process each data element quickly. The time needed by the algorithm to process each data element in the stream is another important complexity measure of the algorithm.

5. The algorithms are restricted to access the input stream in a sequential fashion. However, they may go through the stream in multiple, but a small number, of passes. (Clearly this does not apply to the sliding-window model.) The third important complexity measure of the algorithm is the number of passes required.

Note that these properties characterize the algorithm's behavior during the time when it goes through the input data stream. Before this time, the algorithm may perform certain pre-processing on the workspace (but not on the input stream). After going through the data stream, the algorithm may undertake some post-processing on the workspace. During these two periods, the algorithm needs access only to the workspace. Hence the processing is essentially computation in the traditional model. We do not impose any restriction on it except that the time complexity should be at most near-linear.

There are three important complexity measures in our streaming model: the workspace size, the per-element processing time, and the number of passes that the algorithm needs to go through the stream. In this dissertation, we consider streaming algorithms whose complexity measures fall into the following range: the workspace size is bounded by $O(n^\epsilon)$. The per-element processing time is $O(n^\delta)$, and polylog(n) is preferred. (Here ϵ and δ are two small constants.) The number of passes is $O(1)$ and a one-pass algorithm is preferred.

2.2 Algorithm Design Approaches and Analysis Tools

Previous work on streaming-algorithm design has focused on computing statistics over the stream. There are streaming algorithms for estimating the number of distinct elements in a stream [44] and for approximating frequency moments [6]. Work has been done on approximating L^p differences or L^p norms of data streams [39, 45, 55].

The statistics computed are normally a numerical value. The essence of many such algorithms lies in the construction of a random variable such that the numerical value can be approximated by this random variable. For example, the algorithm in [55] approximates the L^p difference and L^p norm of data streams. It uses a sum of a sequence of random variables. The random variables in the sequence follow a *stable* distribution. That is, they are random variables $X_1, X_2, \ldots, X_n$ with an i.i.d. (independent identical distribution) $\mathscr{D}$ called a p-stable distribution. A p-stable distribution has the following feature: For n real numbers $a_1, a_2, \ldots, a_n$, the random variable $\sum_i a_i X_i$ has the same distribution as the variable $(\sum_i |a_i|^p)^{1/p} X$ where X is also a random variable with distribution $\mathscr{D}$. Therefore, by constructing $\sum_i a_i X_i$, the algorithm can give an estimation on $(\sum_i |a_i|^p)^{1/p}$ that is the L^p norm of the numbers $a_1, a_2, \ldots, a_n$.

The construction of such random variables has been under intensive study [39, 45, 55]. However, the types of problems that can be solved by this approach are limited. There are also other approaches in the design of streaming algorithms. One of the important approaches is the synopsis/sketch approach.

2.2.1 Synopses and Sketches

The concept of synopsis data structures was introduced in [47].

Definition 2.1 (Synopsis Data Structure[47]). *An $f(n)$-synopsis data structure for a class Q of queries is a data structure for providing (exact or approximate) answers to queries from Q that uses $O(f(n))$ space for a data set of size n, where $f(n) = o(n^\epsilon)$ for some constant $\epsilon < 1$.*

The concept of sketches is similar to the synopsis data structures. In [15], the authors used sketches to compare the similarity between two documents. [39] provides a formal definition of the sketch. Let X be a set. A function $f : X^n \times X^n \to Z$ can be computed using sketches if there exists a set S, a (randomized) *sketch function* $h : X^n \to S$ and a (randomized) *reconstruction function* $\rho : S \times S \to Z$ such that, for all $x_1, x_2 \in X^n$, with high probability, $|\rho(h(x_1), h(x_2)) - f(x_1, x_2)| < \epsilon f(x_1, x_2)$.

We observe that, in both cases, the computations of the functions (the queries and the function $f : X^n \times X^n \to Z$) are performed in two stages. At the first stage, a sketch/synopsis data structure is constructed. At the second stage, the functions are evaluated using the sketch/synopsis data structure. This evaluation does not require access to the original input data. If the construction of the sketch/synopsis data structure can be done in a streaming fashion, such a construction leads directly to a streaming algorithm that computes the same function. We now tweak the sketch definition in [39] and give a definition more relevant to streaming computation. This definition is also a generalization of the concept in [68].

Definition 2.2 (Streaming Sketch). *Let X, S be two sets and $f : X^n \to Z$ be a function. Let $h : X^n \to S$ be a (randomized)* sketch function *and $\rho : S \to Z$ be a (randomized)* evaluation function*, such that, for all $x \in X^n$, with high probability,*

$|\rho(h(x)) - f(x)| < \epsilon f(x)$. *We call* $h(x) \in S$ *a sketch of* $x \in X^n$ *for the function* f *if the following holds:*

1. *There exists a (randomized) streaming algorithm that computes* h.

2. *The size of* $h(x)$ *is bounded by* $O(n^\epsilon)$ *for a small constant* $0 < \epsilon < 1$.

3. *The space complexity of computing* ρ *is at most linear to the size of* $h(x)$.

In the following chapters, we will use the term sketch instead of streaming sketch, as we always use the sketch concept in the streaming context.

The synopsis data structures have roots in database systems, where a summary of the data in the database is often kept. The summary can provide inaccurate but reasonable answers to queries. (Such answers are different from approximation because the errors are often not bounded.) It can also be used in query optimizations and other system-tuning tasks. Both the summary data structures in database systems and our sketches are small-space representations of the data. However, the sketch defined above differs from summary data structures in two aspects: first, when the function is evaluated using the sketch instead of the original data, the error is bounded even in the worst case. Second, our sketch is constructed in a streaming fashion. We emphasize the streaming construction here because in some cases, the target function f itself may not be easy to compute in the streaming model while there is a streaming algorithm for computing the sketch.

Clearly, a sketch will not retain all the information in the input data stream. For different functions, there will be different kinds of sketches. The simplest sketch is a random sample of the input data. For many problems, such a sample is not good enough. Several other types of sketches have been used in solving problems such as wavelet transformations [49] and histogram constructions [50, 48]. The exponential histogram [28] is a sketch for maintaining the sum of the data elements in a sliding

window. The geometry algorithms in [57] use a sketch that employs a set of shifted, nested square grids over the geometric space. One important problem in the study of the streaming model is to devise sketches for different functions. In this dissertation, we provide sketches for some geometric and graph problems.

2.2.2 Probability Bounds

In this section, we review two probability bounds that are often used in our proofs. We refer the readers to standard textbooks like [72] for a detailed introduction to probability theory.

The first bound we review is the Chernoff bound. For a sequence of independent random variables, it bounds the tail of the sum of the sequence. A commonly considered type of random variables is the indicator random variable. Let $\{x_i\}$ $i = 1, 2, \ldots, n$ be a sequence of such random variables. The random variable x_i takes the value 1 with probability p_i and the value 0 with probability $1 - p_i$. All the x_is are independent. Let $X = \sum_{i=1}^{n} x_i$ be the random variable representing the sum and let $\mu = E(X) = \sum p_i$ be the expectation of X. Let $0 < \delta \leq 1$ be a constant. The Chernoff bound bounds the probability of both the lower tail and the upper tail. For the lower tail, it states:

$$\mathbf{Pr}[X < (1-\delta)\mu] \leq exp(-\mu\delta^2/2) \tag{2.1}$$

For the upper tail, it states:

$$\mathbf{Pr}[X > (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu} \tag{2.2}$$

The second bound we often use is the union bound. It derives directly from the inclusion-exclusion principle and states that the probability of the union of a set of

events is at most the sum of the probabilities of the events. Let $\mathscr{E}_1, \mathscr{E}_2, \ldots, \mathscr{E}_n$ be an arbitrary set of events, the bound says:

$$\mathbf{Pr}[\cup_{i=1}^n \mathscr{E}_i] = \sum_i \mathbf{Pr}[\mathscr{E}_i] - \sum_{i<j} \mathbf{Pr}[\mathscr{E}_i \cap \mathscr{E}_j] + \cdots \leq \sum_i \mathbf{Pr}[\mathscr{E}_i] \tag{2.3}$$

An important feature of the union bound is that there is no other condition on the events for the bound to hold. In particular, when the events are not independent, the bound still holds. This is different from the above Chernoff bound and is quite useful in places where independence among random variables cannot be achieved.

2.3 Streaming and Communication Complexity

Streaming-space complexity is closely related to communication complexity. Communication complexity studies the amount of communications required among two or more parties to compute certain function. The book by Kushilevitz and Nisan [62] provides detailed descriptions on the models and excellent surveys of the problems that are studied in communication complexity. To help the reader understand the relationship between streaming and communication complexity, here we give a short introduction to the basic communication model and some of the problems that are frequently used in proving streaming-space lower bounds.

The communication model we describe here is the two-parties communication model proposed by Yao [80]. In this model, there are two players (two parties). We often name them Alice (player A) and Bob (player B). Let X, Y, and Z be arbitrary finite sets and $f : X \times Y \to Z$ be an arbitrary function. Alice and Bob want to evaluate the function f. Alice knows only $x \in X$ and Bob knows only $y \in Y$. A communication protocol $\mathcal{P}$ is used by Alice and Bob to compute the function f. The communication complexity of the protocol $\mathcal{P}$ is the number of bits that Alice and

Bob must exchange in order to evaluate the function f [62].

There is a straightforward way to transform a streaming algorithm with space complexity $S(n)$ into a communication protocol with communication complexity $S(n)$. If a streaming algorithm exists for a problem, Alice can simulate the streaming algorithm on her input and then transmit her memory contents to Bob. Bob can continue the streaming algorithm on his input. The amount of communication in this scheme is exactly the amount of memory used by the streaming algorithm. Thus communication-complexity lower bounds for a particular problem are also space lower bounds for the same problem in the streaming model. There are several problems that are well studied in the communication model. Their streaming-space lower bounds thus come automatically from their communication-complexity lower bounds. Furthermore, reductions are often made from these problems to show space lower bounds for other streaming problems. Among these well-studied communication problems, the two most often used problems in these reductions are the set-disjointness problem and the index problem.

Note that reductions used in the streaming context have more restrictions than the ones used in proving **NP**-completeness results. As polynomial-time reductions are required for proving **NP**-completeness results, here we need reductions that can be computed in a streaming fashion. That is, the reduction itself should be a streaming algorithm. Given an input data stream, it produces another stream [11].

The set-disjointness problem asks if two subsets of a set are disjoint, *i.e.*, if the intersection of the two subsets is empty. More formally, the set-disjointness problem is defined as the following:

Definition 2.3 (Set Disjointness). *Given a set U of size n and two subsets $x \subseteq U$ and $y \subseteq U$, the set-disjointness function $disj(x, y)$ is defined to have the value "1" when $x \cap y = \phi$ and "0" otherwise.*

In the communication model, the input to player Alice is x and the input to player Bob is y. The two players then start a communication and output the function value of $disj(x, y)$. In the streaming model, the input stream consists of the concatenation of the subset x with the subset y. In what follows, we will refer to the set-disjointness problem by "DISJ."

The index problem asks to tell the bit value of an entry in a bit string. More formally,

Definition 2.4 (Index). *Given a bit string S of length n and a number $i \leq n$, the index function $ind(S, i)$ is defined to be $S(i)$, i.e., the value of the i-th bits of S.*

In the communication model, we consider one-round communication protocols for the index problem. Alice has the string S and Bob has the number i. There is only one round of communication between Alice and Bob—that is, Alice sends a message to Bob and Bob must send back the one-bit answer. In the streaming model, the input stream consists of the string S concatenated by the number i. The difficulty of the index problem lies in that the bit string S goes before the index number i in the stream.

The following communication-complexity lower bounds hold for these problems:

Theorem 2.1. [59] *The communication complexity of DISJ is $\Theta(n)$.*

Theorem 2.2. [62] *The one-round communication complexity of the index problem is $\Theta(n)$.*

From our discussion about the relationship between streaming-space complexity and communication complexity, the above bounds translate directly into one-pass streaming-space lower bounds for the corresponding streaming problems.

We now show that, in the streaming model, the linear-space lower bound holds for DISJ even if a constant number of passes is allowed.

Theorem 2.3. *DISJ requires linear space in the streaming model, even if $O(1)$ passes are allowed.*

Proof. The lower bound for one-pass stream algorithms can be derived by showing that if we have a stream algorithm for DISJ that requires at most a certain amount of memory, then we are guaranteed to have a communication protocol requiring at most the same amount of communication.

Now, if we allow multiple passes, the above argument still works, except that the communication will be twice the amount of memory required by the stream algorithm multiplied by the number of passes. Assume Alice and Bob possess inputs x and y respectively. In the middle of each pass, Alice sends her memory contents to Bob, and, at the end of each pass, Bob sends back his memory contents: two communications per pass. Thus the total communication will be twice the amount of memory required by the streaming algorithm multiplied by the number of passes.

Given the linear communication complexity of DISJ, it is not possible to have a sublinear-space streaming algorithm that makes a constant number of passes. □

2.4 Streaming Annotation

We end this chapter by showing that streaming-space requirements can be reduced if extra information is added to the input stream. We use the set-disjointness problem as an example.

Intuitively, the entity that generates the data stream may provide this extra information and help to reduce the resource requirements of the streaming algorithm. We model this situation with a stream proof system, in which we have both a streaming verifier and a prover. The prover has unlimited computing power and adds a proof

(or annotation) to the stream. The verifier is a streaming algorithm that checks the proof.

The streaming verifier for a language L can be viewed as a probabilistic oracle machine M that obeys the streaming restriction and satisfies:

- *completeness*: For every stream $x \in L$, there exists a proof π_x:

$$Pr[M^{\pi_x}(x) = 1] \geq \frac{2}{3}$$

- *soundness*: For every stream $x \notin L$ and all proofs π:

$$Pr[M^{\pi}(x) = 1] \leq \frac{1}{3}$$

We measure the complexity of this oracle machine both by the space requirement of the verifier and by the size of the proof.

We now show a stream proof system for DISJ.

Recall that we are trying to compute the function $disj(x, y)$ of two subsets $x \subseteq U$ and $y \subseteq U$ such that the function has value "1" when $x \cap y = \phi$ and "0" otherwise. Once x and y are chosen, each element i in the universe U will belong to one of the following four categories:

C1 $i \in x$ and $i \in y$;

C2 $i \in x$ and $i \notin y$;

C3 $i \notin x$ and $i \in y$;

C4 $i \notin x$ and $i \notin y$;

Algorithm 2.1. Prover

The prover indicates, for each element in U, which of the above four categories it belongs to. This gives a bit vector p. It then appends p to the end of the input stream.

Because there are only four categories, a bit vector of size $O(|U|)$ can be constructed to show the categories of all of the elements in U.

The input to the verifier is a stream of the form xyp. Because p indicates the category for each of the elements in U, the verifier can reconstruct two subsets x' and y' from p. That is, $x' = \{i \in U | i \in \mathbf{C1} \text{ or } i \in \mathbf{C2} \text{ according to the proof } p\}$ and $y' = \{i \in U | i \in \mathbf{C1} \text{ or } i \in \mathbf{C3} \text{ according to the proof } p\}$. The verifier needs to check that $x = x'$ and $y = y'$. For this, it can use Lipton's set-fingerprinting technique [66].

Given a multi-set $s = \{a_1, \ldots a_n\}$, where each a_i is an m-bits number, the fingerprint of this multi-set is computed by evaluating the following polynomial at some random location r.

$$F_s(x) = \prod_{i=1}^{n} (a_i + x)$$

The additions and multiplications are carried out in the field $\mathbb{F}_q$ where q is a prime selected randomly from the interval $[(nm)^2,\ 2(nm)^2]$.

Theorem 2.4. [66] *The probability that two sets $s_1 = \{a_1 \ldots a_k\}$ and $s_2 = \{a_1 \ldots a_l\}$ are unequal but have the same fingerprints is at most*

$$O(\frac{\log n + \log m}{nm} + \frac{1}{nm^2}),$$

where all elements in s_1 and s_2 are m-bit numbers, and $n = \max(k, l)$.

Note that for a set $s = \{a_1 \ldots a_n\}$, the fingerprint can be computed in one pass,

Algorithm 2.2. Verifier

> *The stream verifier checks the following:*
>
> 1. *There is no element of U in category **C1**. This can be checked trivially in one pass through p.*
> 2. *$x = x'$ and $y = y'$. This can be checked by randomly choosing r in $\mathbb{F}_q$ and checking that $F_x(r) = F_{x'}(r)$ and $F_y(r) = F_{y'}(r)$.*
>
> *The verifier accepts if and only if (1) and (2) both hold.*

using $O(\log n)$ space.

Lemma 2.1. *The proof system given by Algorithm 2.1 and Algorithm 2.2 satisfies the "completeness" and "soundness" requirements.*

Proof. It is clear that, if $x \cap y = \phi$, a truthful proof will make the verifier accept. On the other hand, if $x \cap y \neq \phi$, the prover can cheat with a proof p claiming that elements in category **C1** are in other categories. However, in this case, the resulting x' or y' will not be equal to the x or y in the original input. Such a discrepancy can be caught with high probability, according to Theorem 2.4, by comparing fingerprints. □

Note that to determine the sets x' and y', the prover does not need to know the elements in category **C4**. Therefore, it is not necessary to include these elements in the proof. We did so because when the size of x and y is the same order of the universe U, excluding the elements in **C4** from the proof does not save much. On the other hand, if U is much larger than x and y, we can still have a linear size proof by dropping the elements in **C4**.

Theorem 2.5. *DISJ can be verified in $O(\log n)$ space if a linear-size annotation is provided.*

Proof. The proof has size $O(n)$, because it needs only to encode the category for each element in U. The only verification that requires workspace is the check that $x' = x$ and $y' = y$. Because the fingerprint technique needs only $O(\log n)$ space [66], the whole verifier needs only $O(\log n)$ space. □

Chapter 3

Massive Data Streams in Computational Geometry

3.1 Introduction

Massive data streams for computational-geometry problems arise naturally in applications that involve monitoring or tracking entities carrying geographic information. They are also encountered as problems in such areas as information retrieval and pattern recognition, modeled as computational-geometry problems by means of embedding. It is important to study stream algorithms for basic problems in computational geometry.

At the time of our work, aside from the stream-clustering algorithm in [51][1] and the study of the reverse-nearest-neighbor problem in [60], little was known about computational-geometry problems in the streaming or sliding-window models.

Note that a streaming or sliding-window version of a computational-geometry problem is a dynamic problem, *i.e.*, the set of geometric objects under considera-

[1]Note that the stream-clustering algorithm can be applied in any metric space, not just in the Euclidean space.

tion may change. There are additions and deletions to the current set of geometric objects as the computation proceeds. The problems in the streaming model can be viewed as incremental problems while the problems in the sliding-window model can be viewed as a particular type of dynamic problem. However, streaming/sliding-window computations are different from the settings for incremental/dynamic problems. In the streaming/sliding-window model, the algorithm has only limited space and cannot store all the geometric objects in its memory while algorithms for incremental/dynamic problems do not have such a space restriction. Many dynamic algorithms use space that is large enough to hold all the geometry objects. The focus of these algorithms is more on devising data structures that can provide efficient updates and query-answering than on minimizing the amount of space used. This is particularly the case with dynamic problems. Thus, most dynamic computational-geometry algorithms cannot be applied directly in the streaming/sliding-window model. The study of stream algorithms for basic problems in computational geometry is an interesting new research area.

In this work, we initiate the study of computational-geometry problems in the streaming and sliding-window models. In particular, we study the diameter problem in these models. Given a set of points P, the diameter is the maximum, over all pairs x, y in P, of the distance between x and y. In many applications, the data streams consist of the records of geographically dispersed events. Computing extent measures (such as the diameter) is an important component of monitoring the spatial extent of these events. Furthermore, diameter is a basic property of many geometry objects. The diameter information is often used in computations such as clustering. This motivates us to investigate the diameter problem as a step towards better understanding of massive data streams in computational geometry.

At a high-level, our algorithms employ the sketch approach, *i.e.*, they construct

a small space sketch for the input stream, from which the diameter of the whole set of points can be approximated. Note that this sketch is not necessarily a subset of the input points. It also differs from random sampling in that the construction of the sketch depends on the information derived from the part of the stream seen so far. Our algorithms show that for a set of points in two dimensional space, there is a sketch of size $O(1/\epsilon)$ that can be constructed in the streaming fashion. Using this sketch, the diameter of the whole set can be ϵ-approximated. (Denote by A the output of an algorithm and by T the value of the function that the algorithm wants to compute. We say A ϵ-approximates T if $1 > \epsilon > 0$ and $(1+\epsilon)T \geq A \geq (1-\epsilon)T$.) In the streaming construction of this sketch, our algorithm processes each point in $O(\log(1/\epsilon))$ time.

Agarwal *et al.* [2] presented an algorithm that maintains approximate extent measures of moving points. Among these extent measures is the diameter of the set of the points. In [2], this algorithm is not presented as a streaming algorithm. However, it can be simply adapted to compute the diameter in the streaming model. Our sketch is different from their data structure. We reduce the amount of time required to process each point while using slightly more space.

In the sliding-window model, for a set of points in two dimensional space, we show that we can construct a sketch of size $O(\frac{1}{\epsilon^{3/2}} \log^3 n(\log R + \log\log n + \log \frac{1}{\epsilon}))$ bits, where R is the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two points in the window. The diameter of the points in the window can be ϵ-approximated using this sketch. Note that the moving-points model and the sliding-window model are different and thus that the results for maintaining the diameter in the moving-points model in [2] are not directly comparable to our results in the sliding-window model.

The sketches trade the accuracy of the computation for the savings on the space.

The results computed using the sketches have bounded error. Our algorithms in this chapter make only one-sided error. That is, the output $\hat{D}$ of our algorithms ϵ-approximates the true diameter D in that $D \geq \hat{D} \geq (1-\epsilon)D$. On the other hand, we show that exact computation requires more space. Our lower bounds state that computing the exact diameter for a set of n points in the streaming model or maintaining it in the sliding-window model (with window width n) requires $\Omega(n)$ bits of space.

Since our work in [40, 41], more has been done on geometric problems in the streaming model. In [53], Hershberger *et al.* provide a streaming algorithm that ϵ-approximates the diameter using $O(\frac{1}{\sqrt{\epsilon}})$ space and $O(\log(\frac{1}{\epsilon}))$ time per point, thus improving the space upper bound given by our approximation. The heart of their algorithm is the computation of the convex hull of the adaptively sampled extrema. The same convex hull is also used in [53] to approximate other properties, such as enclosure and linear separation, of a stream of points. Using a technique similar to ours, Cormode *et al.* [27] devised the radial histogram to estimate the diameter, furthest neighbor, and convex hull of a stream of points. Furthermore, spatial joints and spatial aggregation such as reverse-nearest neighbors can be estimated using multiple radial histograms.

3.2 Sector-Based Diameter Approximation in the Streaming Model

In this section, we provide a streaming algorithm for approximating the diameter of a set of points in the plane.

A simple streaming adaptation of the algorithm of [2] gives an ϵ-approximation of the diameter using $O(1/\sqrt{\epsilon})$ space and time. Let l be a line and $p, q \in P$ be two

points that realize the diameter. Denote by $\pi_l(p), \pi_l(q)$ the projection of p, q on l. Clearly, if the angle θ between l and the line pq is smaller than $\sqrt{2\epsilon}$, $|\pi_l(p)\pi_l(q)| \geq |pq| \cos\theta \geq (1 - \frac{\theta^2}{2})|pq| \geq (1-\epsilon)|pq|$. By using a set of lines such that the angle between pq and one of the lines is smaller than $\sqrt{2\epsilon}$, the algorithm can approximate the diameter with bounded error. The adaptation can go through the input in one pass, project the points onto each line, and maintain the extreme points for the lines. Thus, it is a streaming algorithm.

However, the time taken per point by this algorithm is proportional to the number of lines used, which is $\Omega(1/\sqrt{\epsilon})$. We now present an almost equally simple algorithm that reduces the running time to $O(\log(1/\epsilon))$. Our basic idea is to divide the plane into sectors and compute the diameter of P using the information in each sector. Sectors are constructed by designating a point x_0 as the center and dividing the plane using an angle of θ. We show two sectors in Figure 3.1.

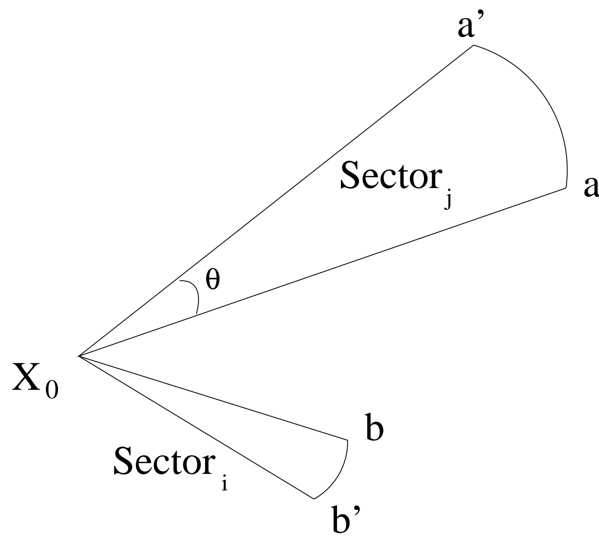

Figure 3.1: Two examples of sectors

The sectors have outer boundaries (the arcs aa' and bb' in the figure) that are determined by the distance between the center and the farthest point from the center

Algorithm 3.1. Streaming_Diameter

1. *Take the first point of the stream as the center, and divide the plane into sectors according to an angle* $\theta = \frac{\epsilon}{2(1-\epsilon)}$, *where* ϵ *is the error bound. Let* S *be the set of sectors.*
2. *While going through the stream, for each sector record the point in that sector that is the furthest from the center. Also keep track of the maximum distance,* R_c, *between the center and any other point in* P.
3. *Let* $|ab|$ *be the distance between points* a *and* b. *Define* $D^{ij}_{max} = \max |uv|$ *for* $u \in$ *boundary arc of sector* i *and* $v \in$ *boundary arc of sector* j, *and define* $D^{ij}_{min} = \min |uv|$ *for* $u \in$ *boundary arc of sector* i *and* $v \in$ *boundary arc of sector* j. *Output* $\max\{R_c, \max_{i,j \in S} D^{ij}_{min}\}$ *as the diameter of the point set* P.

in that sector. The algorithm records the farthest point for each sector while it goes through the input stream. The full description of the algorithm is given in Algorithm 3.1. The algorithm's space complexity is determined by the sector angle θ.

Claim 1. *The distance between any two points in sector* i *and sector* j *is no larger than* $\max\{R_c, D^{ij}_{max}\}$. *(Here* i *could be equal to* j.*)*

Proof. Consider sectors in Figure 3.1. Let u be a point in sector i and v be a point in sector j. Extend x_0u until it reaches the arc aa'. Denote the intersection point u'. Also extend x_0v until it reaches the arc bb'. Denote the intersection point v'. Then we have $|uv| \leq \max\{|x_0v|, |vu'|\} \leq \max\{R_c, |x_0u'|, |u'v'|\} \leq \max\{R_c, D^{ij}_{max}\}$. (In the two inequalities above we have [twice] used the fact that if a, b, c occur in that order on a line and d is some point, then $|db| \leq \max(|da|, |dc|)$. ☐

Claim 2. *With notation as in Figure 3.1 and in the description of the algorithm,* $D^{ij}_{max} \leq D^{ij}_{min} + length(aa') + length(bb') \leq D^{ij}_{min} + 2R_c \cdot \theta$.

Proof. Consider sectors in Figure 3.1. Let $|uv| = D^{ij}_{max}$ and $|u'v'| = D^{ij}_{min}$. Because $u, u' \in arc\ aa'$ and $v, v' \in arc\ bb'$, there is a path from u to v, namely $u \sim u' \sim v' \sim v$. Therefore $D^{ij}_{max} \leq |uu'| + D^{ij}_{min} + |vv'| \leq D^{ij}_{min} + 2R_c \cdot \theta$. □

Assume that the true diameter $diam_{true}$ is the distance between a point in sector i and another point in sector j. Let $diam$ be the diameter computed by our algorithm. We observe the following:

$$\max\{R_c,\, D^{ij}_{min}\} \leq \max\{R_c,\, \max_{m,n \in S} D^{mn}_{min}\} = diam \leq diam_{true} \leq \max\{R_c,\, D^{ij}_{max}\}$$

Depending on the relationship between R_c and D^{ij}_{min}, we consider two cases: In the case where $R_c \geq D^{ij}_{min}$, we want $R_c \geq (1-\epsilon)D^{ij}_{max}$ in order to bound the error. This leads to $\theta \leq \frac{\epsilon}{2(1-\epsilon)}$. In the case where $R_c < D^{ij}_{min}$, we want $D^{ij}_{min} \geq (1-\epsilon)D^{ij}_{max}$. Again, this leads to $\theta \leq \frac{\epsilon}{2(1-\epsilon)}$. We will have $O(\frac{1}{\epsilon})$ sectors. Given the center and another point, it takes $O(\log(1/\epsilon))$ time to identify the sector where the point is located. We then have the following theorem:

Theorem 3.1. *There is an algorithm that ϵ-approximates the 2-d diameter in the streaming model using storage for $O(\frac{1}{\epsilon})$ points. In order to process each point, it takes $O(\log(1/\epsilon))$ time.*

The above algorithm does not work in the sliding-window model. In the streaming model, the boundaries of sectors only expand. This nice property allows us to keep only the extreme points. However, in the sliding-window model, the diameter may decrease with different windows. One may need more information in order to report the diameter for each window.

3.3 Maintaining the Diameter in the Sliding-Window Model

In this section, we present a sliding-window algorithm that maintains the diameter of a set of points in the plane. Note that by applying the projection technique used in the diameter algorithm of [2], we can transform the problem of maintaining the diameter in two-dimensional Euclidean space to the problem of maintaining the diameter in one-dimensional space. Hence, we first consider the problem on a line.

In the sliding-window model, each point has an age indicating its location in the current window. We call the recently arrived points *new* points and the expiring points *old* points. We denote by $|ab|$ the distance between point a and point b. We also say that the distance $r = |ab|$ is *realized* by points a and b. We may further say that r is realized by a when it is not necessary to mention b or when b is clear in context. In particular, we denote by $diam_a$ the largest distance realized by a.

We first show that given a static window, the set of points in the window can be compressed. In our main algorithm, we employ a subroutine for this purpose. We call this subroutine the *rounding* subroutine. When invoked on a set of points, the rounding subroutine produces a space-efficient representation for the set. We call this representation a *cluster*. Once the cluster is constructed, the diameter of the set of points can be approximated by computing the diameter of this cluster. We now describe the rounding subroutine in detail. (Note that, for simplicity, we describe here the rounding subroutine invoked on one set of points. In our main algorithm that maintains the diameter in the sliding-window model, the rounding subroutine will be invoked on different subsets of points in the window, as well as on the clusters, which essentially are also sets of points.)

Given three points a, b, c and an approximation error of $\hat{\epsilon}$, if we treat point

c as a center (the coordinate zero), we can "round" point b to point a if $|ac| \leq |bc| \leq (1+\hat{\epsilon})|ac|$ and a, b lie on the same side of c. For a set of points, we can pick some point in the set as the center and round the other points in the same manner. Let c be the point in the set picked as the center. Let d be the minimum distance between c and any other point in the set. Consider the distance intervals $[c, t_0), [t_0, t_1), [t_1, t_2), \ldots, [t_{k-1}, t_k]$, such that $|ct_i| = (1+\hat{\epsilon})^i d$. The rounding subroutine rounds down (moves) each point in the interval $[t_i, t_{i+1})$ to the location t_i (Figure 3.2).

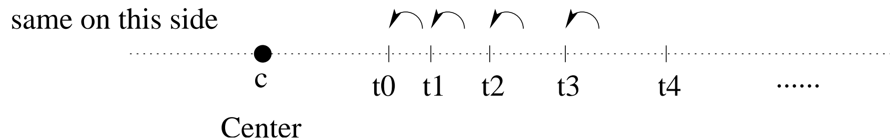

Figure 3.2: Rounding points in each interval

If multiple points are rounded to the same location, the subroutine keeps only the newest one. There is at most one point in each of these intervals. The cluster for this set of points consists of the center c and the points that are kept by the subroutine after rounding them. Note that the points in a cluster are different from the points in the original set. We say a point a in the cluster *represents* a point b in the original set if b is rounded to the location of a by the rounding subroutine, regardless of whether b is discarded. (We call the point a the *representative point* of the *original* point b.) Note that a point a in the cluster may represent several points in the original set that are rounded to its location. The location of the representative point in the cluster may not be the same as the location of the point in the original set.

The *volume* of a cluster is the number of points in the original set that are represented by the cluster. We say a cluster is at level ℓ if the volume of the cluster

is 2^ℓ. The *size* of a cluster is the number of points in the cluster. Let D be the diameter of a set of points. The size of the cluster constructed by invoking the rounding subroutine on this set is at most:

$$k \leq 2\log_{1+\hat{\epsilon}} \frac{D}{d} = 2\frac{\log D/d}{\log e \ln(1+\hat{\epsilon})} \leq \frac{4}{\hat{\epsilon}\log e} \log \frac{D}{d}$$

Hence, given a set of points with diameter D and an arbitary point c in the set, if the nonzero minimal distance between c and any other point in the set is d and if the approximation allows an error of $\hat{\epsilon}$, the rounding subroutine constructs a cluster. The diameter of the set of points can be approximated by the diameter of this cluster. The size of the cluster is upper bounded by $\frac{4}{\hat{\epsilon}\log e}\log\frac{D}{d}$. Also, we observe that there may be a displacement between an original point in the set and its representative point in the cluster. In this case, the representative point is always closer to the center than the original point. Thus, we say that the point in the original set is rounded (moved) "toward the center."

Claim 3. *When invoked on a set of points, the rounding subroutine constructs a cluster that can be used to approximate the diameter of the set of points. If there is a displacement between a point in the set and its representative point in the cluster, the point will only be rounded (moved) toward the center of the cluster.*

We now proceed to describe our main algorithm that maintains the diameter in the sliding-window model. Recall that when the window slides forward, there are new points added to the window as well as old points retiring from the window. A cluster constructed for a set of points in a static window will not be useful for approximating the current diameter if, after the construction of the cluster, some of the points in the set have been deleted and some new points have been added. We now analyze several problems that come from such additions and deletions and see

how they affect our main algorithm.

For a static set of points, by constructing the cluster we are able to represent a point, say b, by another point, say a, because there is some distance (for example $|bc|$) realized by b that promises a lower bound for any diameter that may be realized by b, and the displacement (if any) because of rounding is at most $\hat{\epsilon}|bc|$. In the sliding-window model, such a promise provided by the point c may be broken when c expires. We observe that in the construction of a cluster, the distance from a point to the center is used as the promised lower bound. To avoid the aforementioned problem, when constructing clusters, we always use the newest point in the set as the center.

Such clusters can successfully approximate the diameter if no additional points are added to the window. The situation changes when a new point p is added. Note that p is now the newest one in the current window. We cannot round p using the original center of the cluster, which is older than p. Doing so will cause the aforementioned problem. To maintain the property that the center of a cluster is the newest point in the set represented by the cluster, we need to make p a new center.

The following is a straightforward way to do this: Essentially, a cluster is also a set of points. Assume we have a cluster that represents all the points in the previous window. Now the window slides forward and a new point arrives. We can delete the expired point from the cluster and view the newly added point and the remaining points in the cluster as a new set of points. We take the newly added point as the center and invoke the rounding subroutine on this set. This then gives a new cluster.

However, this method is troublesome. Note that each time we do rounding, we may introduce a certain displacement between a point and its representative point in the cluster. The cluster approximates the diameter because such a displacement is bounded by ϵ of a lower bound on the diameter that can be realized by the

point. If we update the center whenever we see a new point, we need to perform the rounding process using the new center as well. Hence, each new point may cause some displacement to each point. Although the amount of a single displacement is bounded, the displacements will add up and cause large errors, *i.e.*, the displacement between the representative point and the original point becomes so large that the error in the diameter estimation using the representative points is no longer bounded.

Because of these difficulties, we use multiple clusters in our main algorithm. The multiple clusters serve two purposes: first, with multiple clusters, we can ensure that the center of each cluster is the newest point in the set of points represented by that cluster. More important, we use only a small number of clusters such that during the time a point is in the window, the point and its representative points are only subject to the rounding subroutine for a small number of times. Therefore, the total displacement caused by invocations of the rounding subroutine can be controlled.

In the main algorithm, we maintain multiple clusters with the following properties:

1. A cluster represents an interval of points in the window (a contiguous subsequence of points within a time interval of the window). The newest point in the interval is picked to be the center. The rounding subroutine is invoked on this interval of points, and the resulting cluster represents these points in the window interval.

2. The clusters are at levels $1, 2, \ldots, \lfloor \log n \rfloor$.

3. We allow at most two clusters at each level.

4. When the number of clusters at level i exceeds 2, the oldest two clusters (where the age of a cluster is determined by the age of its center) at that level are merged to form a cluster at level $i + 1$.

Imagine a tree built on the original input points in a window. The points are the leaves. Two consecutive points can form a node (a cluster) at level 1. Two consecutive level-1 clusters can merge to form a node (a cluster) of level 2. This can be repeated until we reach the top level. In this structure, the original input points represented by a cluster are the leaves of the subtree rooted at the node corresponding to that cluster. Note that at each level, we only keep at most two nodes (clusters). The clusters form a cover of the window—that is, each original input point in the window is represented by some cluster. The whole window can be represented by $O(\log n)$ clusters. Figure 3.3 shows an example of the clusters built on a window.

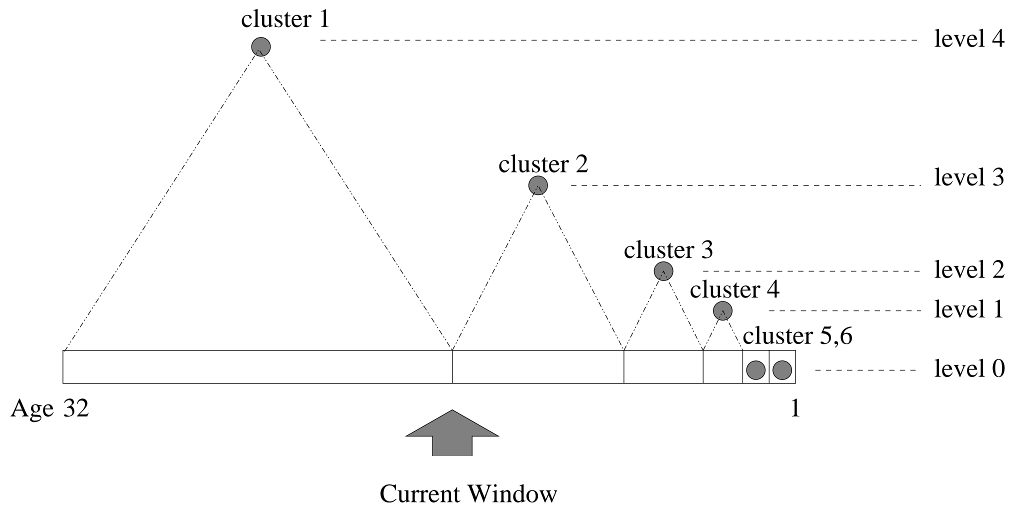

Figure 3.3: Clusters built for the first window

When the window slides forward, new points are added to the window, and new clusters are formed. To maintain the required number of clusters at each level, clusters are merged whenever there are too many clusters at some level. Once a cluster reaches the top level, it stays at that level. Points in this cluster will ultimately be aged out until the whole cluster is gone.

In order to merge clusters c_1 centered at Ctr_1 and c_2 centered at Ctr_2 to form cluster c_3, we go through the following steps (we can assume *w.l.o.g.* c_2 is newer than

c_1):

1. Use Ctr_2 as the center of newly formed cluster c_3.
2. Discard the points in c_1 that are located between the centers of c_1 and c_2.
3. After step (2), if any point p in c_1 satisfies $|pCtr_2| < |pCtr_1| \leq (1+\hat{\epsilon})|Ctr_1Ctr_2|$, discard p.
4. Let P_{merge} consist of the remaining points of c_1 and the points in c_2. Invoke the rounding subroutine on P_{merge} with Ctr_2 as the center. Note that the new value of d is the nonzero minimal distance from Ctr_2 to any other point in P_{merge}. It may be different from the one used in building the cluster c_2. The rounding subroutine may round down the points in cluster c_2 too.

From step (4), we know that the new value of d is the nonzero minimal distance between Ctr_2 and any other point in P_{merge}. Let p_{min} be this minimal distance point. If p_{min} belongs to cluster c_1, the distance $|Ctr_2p_{min}|$ may be much smaller than the distance between the point Ctr_2 and the original point(s) represented by p_{min}. This happens because when the points are rounded to form the cluster c_1, the rounding is based on the distance between these points and the center Ctr_1, not the point Ctr_2. Thus we can't lower bound the value of d for the new cluster c_3 by the minimal distance between its center and any other original point whose representative point is in the cluster. However, steps (2) and (3) assure that $|Ctr_2p_{min}|$ is at least $\hat{\epsilon} \cdot |Ctr_1Ctr_2|$. Otherwise, p_{min} will be discarded. We know that the two points Ctr_1 and Ctr_2 are at their original locations. Thus, d is bounded by $\hat{\epsilon}$ times the minimal distance between the cluster center and any other original points whose representative points are in the cluster. The lower bound for the whole window will then be the minimum over all the clusters.

In both the rounding subroutine and the merging process, we may discard points. We now show that if any of the discarded points has the potential of realizing the diameter, it is represented by some representative point in the resulting cluster.

Lemma 3.1. *In the rounding subroutine or in the merging process, if a point is discarded, either it will not realize any diameter or it is represented (by some representative point) in the cluster resulting from the rounding subroutine or the merging process.*

Proof. The case with the rounding subroutine is clear because a point b is discarded only if there is already a representative point a in the cluster for b and the age of a is smaller than that of b.

In the merging process, we discard two types of points. For two clusters c_1 and c_2 with centers Ctr_1 and Ctr_2, the first type of points we discard are the points in c_1 that are located between Ctr_1 and Ctr_2. Let p_0 be such a point. Note that for any distance realized by p_0 and some other points p, there is a point $p' \in \{Ctr_1, Ctr_2\}$ such that $|pp'| > |pp_0|$. Hence, the first type of points we discard will not realize any diameter. The second type of points we discard are the set of points $S = \{p : p \in c_1 \text{ and } |pCtr_2| < |pCtr_1| \leq (1 + \hat{\epsilon})|Ctr_1Ctr_2|\}$. For the points in S, Ctr_2 is their representative point in the new cluster resulting from the merging process. □

Define a *boundary point* in a cluster to be an extreme point. (Because here the points are on a line, the extreme point is the point having the largest [smallest] coordinate.) We keep track of the boundary points for each cluster as well as the boundary points for the whole window. Points may expire from the oldest cluster, and this may require updating the boundary points of this cluster. The whole process is summarized in Algorithm "sliding-window diameter."

Algorithm 3.2. Sliding-Window Diameter

> ***Update***: *When a new point arrives:*
>
> 1. *Check the age of the boundary points of the oldest cluster. If one of them has expired, remove it and update the boundary point.*
> 2. *Make the newly arrived point a cluster of size 1. Go through the clusters from the most recent to the oldest and merge clusters whenever necessary according to the rules stated above. Update the boundary points of the clusters resulting from merges.*
> 3. *Update the boundary points of the window if necessary.*
>
> ***Query Answer***: *Report the distance between the boundary points of the window as the window diameter.*

Call the time during which an original point is within some sliding window the *lifetime* of that point. Let's trace a point p through its lifetime. For simplicity, in what follows, instead of saying that the original point p is represented by some point in some cluster, we will just say that p is contained or included in that cluster. When clusters merge, if the new representative point of p is located at a different location, we will just say p has been rounded ("moved") to a new location and there is a displacement between the new and the old locations of p.

Let p_0 be the original location of the point p and Ctr_0 be the center of the first cluster that includes the point p. When this cluster and some other cluster merge, p could be rounded to a new location p_1. Let Ctr_1 be the center of the newly formed cluster. If we continue this process, before p expires or is no longer represented by any representative point in the window, we will have a sequence of p's locations $p_0, p_1, \ldots$ and corresponding sequence of centers $Ctr_0, Ctr_1, \ldots$. Assume that at a certain time t, p realizes the distance $diam_p$. Let $p_0, p_1, \ldots, p_t$ be the sequence of p's locations up to the time t and $Ctr_0, Ctr_1, \ldots Ctr_t$ the corresponding sequence of

centers. We have the following claim:

Figure 3.4: A point may be moved in each rounding, but all the displacements are in the same direction.

Claim 4. *If a point is rounded multiple times during its lifetime, all the displacements because of rounding are in the same direction (Figure 3.4). That is, for all the locations p_i and all the corresponding centers Ctr_i, $|p_0Ctr_i| \geq |p_iCtr_i|$. Furthermore, if a point realizes the diameter at a certain time, the distance between the original point and any of its cluster centers up to that time is at most the distance of this diameter. That is, for $i = 1, 2, \ldots, t$, $|p_0Ctr_i| \leq diam_p$.*

Proof. Suppose that the first time p is rounded, it is rounded to the right. If now p is rounded to the left for the first time on step i, then by Claim 3, Ctr_{i-1} lies to the right of p while Ctr_i lies to the left. Furthermore, p belonged to the cluster of Ctr_{i-1} before the merge. Hence, by our rules it would have been *discarded*, because it lies between the two centers, and it belongs to the older cluster. Furthermore, as shown in the proof of Lemma 3.1, such a point p will not realize the diameter and would no longer be represented by any point in the clusters in the window.

Note that $diam_p$ is the true diameter realized by the point p. p_0 is the original location of p and for $i = 1, 2, \ldots, t$, the center Ctr_i is also at its original location. Furthermore, these centers are no older than the point p in the window. Hence, for $i = 1, 2, \ldots, t$, the diameter realized by p is at least the distance $|p_0Ctr_i|$. □

We bound the error in the rounding process by showing that for all i, $|p_0p_i|$ is at most an ϵ fraction of the diameter realized by p.

Each time we round a point, we may introduce some displacement or error. Let $err_{i+1} = |p_i p_{i+1}|$ be the displacement introduced in the $i+1$th merging. We have the following lemma:

Lemma 3.2. *The total rounding error of a point p, before it expires or is no longer represented by any representative point in the window, is at most $\hat{\epsilon} \log n \cdot diam_p$.*

Proof. We examine the two cases where there may be a displacement between p_i and p_{i+1}. In the rounding case, we maintain $|p_i Ctr_{i+1}| \le (1+\hat{\epsilon})|p_{i+1} Ctr_{i+1}|$. Hence, $err_{i+1} = |p_i p_{i+1}| \le \hat{\epsilon}|p_{i+1} Ctr_{i+1}| \le \hat{\epsilon}|p_0 Ctr_{i+1}|$, where the last inequality is by Claim 4. The second case is step (3) of the merging process. If p_i satisfies the condition, it will be discarded. As stated in the proof of Lemma 3.1, in this case, Ctr_{i+1} becomes the new representative point of p. Hence, p_{i+1} is Ctr_{i+1} and $err_{i+1} = |p_i p_{i+1}| \le \hat{\epsilon}|Ctr_i Ctr_{i+1}| \le \hat{\epsilon}|p_i Ctr_i|$. By Claim 4, $err_{i+1} \le \hat{\epsilon}|p_0 Ctr_i|$.

A point may participate in at most $\log n$ merges. The total amount of displacement is then at most $\sum_i err_i \le \hat{\epsilon} \log n \cdot \max_i |p_0 Ctr_i|$. By Claim 4, we also have $diam_p \ge \max_i |p_0 Ctr_i|$. The lemma follows. ☐

To bound the error by $\frac{1}{2}\epsilon$, we make $\hat{\epsilon} \le \frac{\epsilon}{2 \log n}$. The number of points in a cluster after rounding will then be $O(\frac{1}{\epsilon} \log n \log \frac{D}{d})$. Note that for each cluster, d is bounded by $\hat{\epsilon}$ times the minimal distance between the center of the cluster and any other original point whose representative point is in the cluster. Denote by R the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two original points in that window. Then $\log \frac{D}{d} \le \log R + \log \frac{1}{\hat{\epsilon}} = O(\log R + \log\log n + \log \frac{1}{\epsilon})$. The number of points in a cluster can then be bounded by $O(\frac{1}{\epsilon} \log n(\log R + \log\log n + \log \frac{1}{\epsilon}))$.

Theorem 3.2. *There is an ϵ-approximation algorithm for maintaining diameter in one dimension in a sliding window of size n, using $O(\frac{1}{\epsilon} \log^3 n(\log R + \log\log n +$*

$\log\frac{1}{\epsilon}))$ *bits of space, where R is the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two points in that window. The algorithm answers the diameter query in $O(1)$ time. Each time the window slides forward, the algorithm needs a worst-case time of $O(\frac{1}{\epsilon}\log^2 n(\log R+\log\log n+\log\frac{1}{\epsilon}))$ to process the incoming point. With a slight modification, the algorithm can process incoming points with $O(\log n)$ amortized time using $O(\frac{1}{\epsilon}\log^2 n(\log n + \log\log R + \log\frac{1}{\epsilon})(\log R + \log\log n + \log\frac{1}{\epsilon}))$ bits of space.*

Proof. By Lemma 3.1, any point that may realize the diameter for some window has a representative point in one of the clusters we maintain. We now calculate the error in reporting the diameter using these representative points. Set $\hat{\epsilon} \leq \frac{\epsilon}{2\log n}$. By Lemma 3.2, for a point p and the diameter $diam_p$ realized by p, the displacement between the original location of p and the location of its representative points is at most $\hat{\epsilon}\log n \cdot diam_p \leq \frac{\epsilon}{2}diam_p$. Because our algorithm reports the diameter realized by the representative points, such a displacement causes error in our diameter approximation. Note that each of the two representative points that realize the reported diameter may introduce an error at most $\frac{\epsilon}{2}$ of the true diameter. Hence, the diameter reported by our algorithm is at least $(1-\epsilon)$ of the true diameter. Also note that the reported diameter is at most the true diameter. Otherwise, let p and p' be the two original points whose representative points $R(p)$ and $R(p')$ realize the diameter reported by our algorithm. Then $|pp'| < |R(p)R(p')|$. By Claim 3, the displacement between the location of p (or p') and the location of $R(p)$ (or $R(p')$) is caused by rounding (multiple times) p (or p') toward some center(s). Hence, there exists at least one center c such that $|R(p)c|$ or $|R(p')c|$ is larger than $|R(p)R(p')|$. This contradicts the fact that $|R(p)R(p')|$ is the diameter reported by our algorithm.

We now analyze the time and space requirement of our algorithm. For each cluster, we maintain the following information:

1. The exact location of the center and the exact location of the point closest to (but not located at) the center.

2. The age of all the points.

3. The relative positions of all the points other than the center.

The relative positions of all the points in a cluster can be encoded by a bit vector. We may need $\log n$ bits of space to record the age in the current window for each point. Thus, we need $O(\log n)$ bits for each cluster point except the center. There are at most $O(\frac{1}{\epsilon}\log n(\log R + \log\log n + \log\frac{1}{\epsilon}))$ points in each cluster. The space requirement for storing the information in items (2) and (3) for the whole cluster is then $O(\frac{1}{\epsilon}\log^2 n(\log R + \log\log n + \log\frac{1}{\epsilon}))$. Because we assumed that this space is much larger than the space required to store two points, we can neglect the latter (the space for information in item (1)). Given that there are $O(\log n)$ clusters, the total space requirement will be $O(\frac{1}{\epsilon}\log^3 n(\log R + \log\log n + \log\frac{1}{\epsilon}))$ to maintain the diameter.

In order to report the diameter at any time, we maintain the two boundary points for the window while we maintain the clusters. For each cluster, we need only to look at its boundary points, and thus the process of updating the sliding window's boundary points will only cost $O(\log n)$ time.

However, while updating the clusters, we may face a sequence of cascading merges. In the worst case, we may need to merge $O(\log n)$ clusters with $O(\frac{1}{\epsilon}\log n(\log R + \log\log n + \log\frac{1}{\epsilon}))$ points in each. This requires time $O(\frac{1}{\epsilon}\log^2 n(\log R + \log\log n + \log\frac{1}{\epsilon}))$.

If a bit vector is used to specify the relative locations of the points in a cluster, when we process the cluster during merging we may need to go through the zero entries in the vector. This could be a waste of time if the vector is sparse.

We can directly specify the relative location of a point instead. Because there are $O(\frac{1}{\epsilon}\log n(\log R + \log\log n + \log\frac{1}{\epsilon}))$ different locations, we need an additional $O(\log\frac{1}{\epsilon} + \log\log n + \log\log R)$ bits, besides the $O(\log n)$ bits stated above, for each point in a cluster. The space requirement for each point in a cluster will then be $O(\log n + \log\log R + \log\frac{1}{\epsilon})$. With this modification, when merging two clusters, we are free of overhead other than processing the points in the clusters. During a point's lifetime, it will take part in at most $\log n$ merges. Thus, a simple analysis can show that the amortized cost for updating is now only $O(\log n)$. □

To extend the algorithm to 2-d, we can apply the projection technique. We use a set of lines and project the points in the plane onto the lines. We guarantee that for any pair of points, they will project to a line with angle θ such that $1 - \cos\theta \le \frac{\epsilon}{2}$. This will require $O(\frac{1}{\sqrt{\epsilon}})$ lines. We then use our diameter-maintenance algorithm on lines to maintain the diameter in the 2-d case.

As pointed out by [19], one problem we need to take care of here is that the distance between two points in the two-dimensional space, when projected onto some line, can be made arbitrarily small. Therefore, the R value is not controlled. To avoid this, we keep the exact locations of the two newest points in the current window. Let the distance between them be d. Recall that in step 3 of the merging process, the distance between the two centers of the clusters to be merged, $|Ctr_1Ctr_2|$, is used in deciding whether a point should be discarded. Here, when we extend the algorithm to the two-dimensional space, we use d in place of $|Ctr_1Ctr_2|$ in the merging process. The second change we make to step 3 of the merging process is that we consider the points on both sides of the center Ctr_2 in terms of whether they should be discarded. Therefore, the condition for discarding a point p in the one-dimensional case, $|pCtr_2| < |pCtr_1| \le (1+\hat{\epsilon})|Ctr_1Ctr_2|$, becomes $|pCtr_2| \le \hat{\epsilon}d$ in the two-dimensional case. With this modification, the R value is preserved because d is

always a true value of the distance between some points in the window.

Theorem 3.3. *There is an ϵ-approximation algorithm for maintaining diameter in 2-d in a sliding window of size n using $O(\frac{1}{\epsilon^{3/2}} \log^3 n(\log R + \log\log n + \log \frac{1}{\epsilon}))$ bits of space, where R is the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two points in that window.*

3.4 Lower Bounds

It is well known that the set-disjointness problem has linear communication complexity [59] and thus a linear-space lower bound in the streaming model. (These lower bounds apply to randomized protocols and algorithms as well.) One can map the set elements to points on a circle such that the diameter of the circle will be realized if and only if the corresponding element is presented in both sets. This reduction gives the following theorem:

Theorem 3.4. *Any streaming algorithm that computes the exact diameter of n points, even if each point can be encoded using at most $O(\log n)$ bits, requires $\Omega(n)$ bits of space.*

Proof. We reduce the set-disjointness problem to a diameter problem. Recall that the set-disjointness problem is defined as follows: Given a set U of size n and two subsets $x \subseteq U$ and $y \subseteq U$, the function $disj(x, y)$ is defined to be "1" when $x \cap y = \phi$ and "0" otherwise. The corresponding language DISJ is the set $\{(x, y) | x \subseteq U,\ y \subseteq U,\ x \cap y = \phi\}$.

The set-disjointness problem has a linear communication-complexity lower bound. Because a streaming algorithm can be easily transferred into a one-round communication protocol, the linear communication-complexity lower bound gives a linear space lower bound for the set-disjointness problem in the streaming model.

Consider points on a circle in the plane. For a given point p_i, there is exactly one other point on the circle such that the distance between it and p_i is exactly equal to the diameter of the circle. Denote this antipodal point p_i'. The distance between p_i and all other points on the circle is smaller than the distance between p_i and p_i'. We map each element $i \in U$ onto one such antipodal pair. We further make the appearance of one point in the pair correspond to the appearance of the element i in subset x and the appearance of the other point correspond to the element i in y. We will have both points p_i and p_i' only if the element i is in both subsets x and y.

Given an instance (x, y) of DISJ, we construct an instance of the diameter problem according to the above principle. We give an example in Figure 3.5.

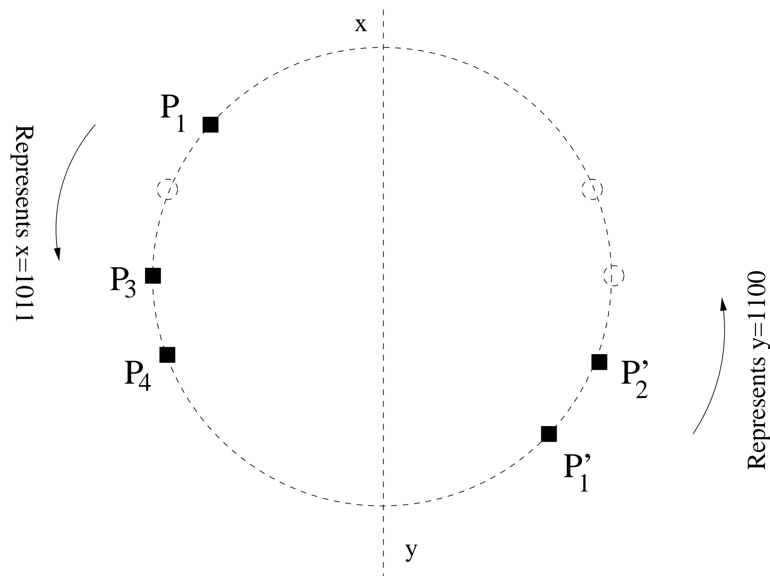

Figure 3.5: Reduction from DISJ to diameter

The solid squares in the figure are the points we put into the diameter instance. The DISJ instance in Figure 3.5 is x, y, where $x = 1011$ and $y = 1100$. The diameter instance contains p_1, p_3, p_4, because $x = 1011$, and p_1', p_2', because $y = 1100$. The dashed circles in the figure show the location for p_2, p_3', p_4'. Because $x_2 = 0$ and $y_3 = y_4 = 0$, these points are not presented in the stream.

In the example, element 1 is in both x and y. The diameter of the point set constructed is $|p_1 p_1'|$ and is exactly the diameter of the circle. On the other hand, if $x \cap y = \phi$, the diameter of the point set will be strictly smaller than the diameter of the circle. Thus, an exact algorithm for the diameter problem could be used to solve the set-disjointness problem. □

We remark that in the above construction, each point in the input stream may be encoded using at most $O(\log n)$ bits. Hence, Theorem 3.4 gives a space lower bound of $\Omega(n)$ bits for a stream of length $O(n \log n)$ bits.

For a positive number $\epsilon \leq \frac{1}{\pi^2}$, consider the same construction with a set of at most $\frac{1}{\sqrt{\epsilon}}$ points. Let x_ϵ and y_ϵ be the two sets represented by these points and let the diameter of the circle used in the construction be 1. If $x_\epsilon \cap y_\epsilon \neq \phi$, the diameter of the set of points is 1 while if $x_\epsilon \cap y_\epsilon = \phi$, the diameter would be at most $\cos(\pi\sqrt{\epsilon})$. Because $1 - \cos(x) \geq \frac{1}{2}x^2 - \frac{1}{24}x^4 \geq \frac{11}{24}x^2$, $1 - \cos(\pi\sqrt{\epsilon}) \geq \frac{11\pi^2}{24}\epsilon \geq 3\epsilon$, a streaming algorithm that can $(1 \pm \epsilon)$-approximate the diameter will be able to distinguish the two cases and thus solve the set-disjointness problem on the two sets x_ϵ and y_ϵ. Such a streaming algorithm requires $\Omega(\frac{1}{\sqrt{\epsilon}})$ bits of space.

In the sliding-window case, we have a similar bound even for points on a line. Obviously the lower bound holds for higher dimensions as well.

Theorem 3.5. *To maintain, in a sliding window of size n, the exact diameter of a set of points on a line, even if each point in the set can be encoded using $O(\log n)$ bits, requires $\Omega(n)$ bits of space.*

Proof. Consider a family $\mathscr{F}$ of point sequences of length $2n-2$. Let $\{a_1, a_2, \ldots, a_{2n-2}\}$ be a sequence in $\mathscr{F}$. Because a_i is a point in one dimension, we denote by a_i the point as well as the coordinate (a real number) of the point. The sequences in the family $\mathscr{F}$ have the following properties:

1. For $i = n, n+1, \ldots, 2n-2$, a_i is located at coordinate zero, *i.e.*, $a_i = 0$. Furthermore, $a_{n-1} = 1$.

2. $|a_1 a_n| \geq |a_2 a_{n+1}| \geq |a_3 a_{n+2}| \geq \ldots \geq |a_{n-1} a_{2n-2}|$, *i.e.*, $a_1 \geq a_2 \geq a_3 \geq \ldots \geq a_{n-1}$.

3. The coordinates of the points a_j, for $j = 1, 2, \ldots, n-2$, are picked from the set $\{2, 3, \ldots, n\}$.

For a window that ends at point a_s, the diameter is exactly the distance $|a_s a_{s+n-1}|$. Any two members of the family will have different diameters for a window that ends at a_s, for some $s \in \{1, 2, \ldots, n-2\}$, where the coordinates of a_s differ in the two sequences. Thus, an algorithm that maintains the diameter exactly has to distinguish any two sequences in $\mathscr{F}$.

We need to assign values for the $n-2$ coordinates $a_1 \geq a_2 \geq \ldots \geq a_{n-2}$. By Property (3), we have $n-1$ possible values to choose. (We may assign the same value to multiple coordinates.) The number of such assignments (and thus the number of sequences in $\mathscr{F}$) is $\binom{n-2+n-2}{n-2} \geq 2^{n/2}$. Hence, the algorithm needs $\log |\mathscr{F}| = \Omega(n)$ space. □

We remark that in this family $\mathscr{F}$, the nonzero minimal distance between any two points in a sequence is at least 1. For each sequence, the ratio of the diameter over the nonzero minimal distance between any two points is at most n. Hence, the lower bound holds without the requirement of an extremely large R.

If we change the form of the coordinates of a_j for $j = 1, 2, \ldots, n-2$ to $(1+\epsilon)^{3k}$ while respecting the Property (2) above, a similar family of point sequences can be constructed for ϵ-approximation algorithms. We have the following lower bounds for approximation from this modified family of point sequences.

Theorem 3.6. *Let R be the maximum, over all windows, of the ratio of the diameter to the minimum nonzero distance between any two points in that window. To ϵ-approximately maintain the diameter of points on a line in a sliding window of size n requires $\Omega(\frac{1}{\epsilon}\log R \log n)$ bits of space if $\log R \leq \frac{3\log e}{2}\epsilon \cdot n^{1-\delta}$, for some constant $\delta < 1$. The approximation requires $\Omega(n)$ bits of space if $\log R \geq \frac{3\log e}{2}\epsilon \cdot n$.*

Proof. Once again consider the family of point sequences in the proof of Theorem 3.5. We make the following change: The coordinates of the points a_j for $j = 1, 2, \ldots, n-2$, have the form $(1+\epsilon)^{3k}$, for some $k \in \{1, 2, \ldots, \lfloor \frac{1}{3}\log_{(1+\epsilon)} R \rfloor = m\}$. These coordinates are chosen so as to respect Property (2) in the proof of Theorem 3.5. Note that $\frac{2}{3\log e \cdot \epsilon}\log R \geq m \geq \frac{1}{3\log e \cdot \epsilon}\log R$, for ϵ sufficiently small, because $\epsilon/2 \leq \ln(1+\epsilon) \leq \epsilon$. Depending on the value of $\log R$, we consider two cases:

1. $\log R \leq \frac{3\log e}{2}\epsilon \cdot n^{1-\delta}$ for some constant $\delta < 1$. By a similar argument to the one given in the proof of Theorem 3.5, the space requirement will now be lower bounded by:

$$\log \binom{m-1+n-2}{m-1} = \Omega(m \log \frac{n}{m}) = \Omega(\frac{1}{\epsilon}\log R(\delta \log n))$$
$$= \Omega(\frac{1}{\epsilon}\log R \log n)$$

2. $\log R \geq \frac{3\log e}{2}\epsilon \cdot n$. In this case, $m \geq \frac{n}{2}$. We can always choose from $\frac{n}{2}$ different values for the coordinates of the points $a_1, \ldots, a_{n-2}$. (Same value can be chosen for multiple coordinates.) The space requirement will be lower bounded by

$$\log \binom{n/2-1+n-2}{n/2-1} = \Omega(\frac{n}{2} \cdot \log 2) = \Omega(n)$$

□

3.5 Closest Pair and K-Promised Convex Hull

In this section, we consider two problems that are related to the diameter problem. The first problem we consider is the closest-pair problem. The diameter problem finds the maximum of the pairwise distance. The closest-pair problem finds the minimum.

Definition 3.1. *The **Closest Pair** is the pair of points in the input stream the distance between which is the minimum among all pairwise distances in the stream.*

Theorem 3.7. *Any exact streaming algorithm for the closest pair requires $\Omega(n)$ memory bits.*

Proof. Again, we reduce from set disjointness. Consider points in one-dimensional space. Construct a stream of this type of points from the disjointness instance. Given (x, y) as an instance of disjointness, if the ith bit of x is "1," add to the stream the point with coordinate i. If it is "0," add nothing. For the subset y, if the ith bit of y is "1," the point $i - \epsilon$ is added, and, if it is "0," nothing is added.

If $x \cap y = \phi$, the minimal pairwise distance will be $1 - \epsilon$. On the other hand, if $x \cap y \neq \phi$, the minimal distance will be ϵ. By solving the closest-pair problem exactly, we could thus solve the set-disjointness problem. □

Unlike the diameter problem, constant-factor approximation of the closest-pair problem is not easy. If there were an algorithm that approximated the minimum distance within some factor ϵ', one could always manipulate the ϵ in the stream construction described above such that a proper ϵ could always be chosen to guarantee that the approximation would allow us to distinguish the case of $x \cap y \neq \phi$ from the case of $x \cap y = \phi$.

Definition 3.2. *The **Convex Hull** of a set of points is the smallest convex set containing the points.*

Definition 3.3. *The K**-promised convex-hull** problem is a convex-hull problem in which the input point set is guaranteed to admit a convex hull with at most K sides.*

We use a reduction from the index problem to show that the space requirement for K-promised convex hull is $\Omega(n)$.

Theorem 3.8. *One-pass streaming algorithms for K-promised convex hull require $\Omega(n)$ space.*

Proof. Given a bit vector S of length n, an algorithm that solves the index problem must return the bit S_i for some specified index i. Consider the points on two concentric circles, one of radius r and the other of radius $r - \epsilon$. For a point a_i on the inner circle, there is a point a_i' on the outer circle such that the center of the two circles lies on the line $a_i a_i'$. Call the pair (a_i, a_i') a "unit." We map the bits in S to the units. Namely, if $S_i = 1$, we put the point a_i' into the stream. Otherwise, a_i is added. We do this for all the bits in S. Note that different bits are mapped to different units that are distributed evenly along the circles. The result of this is a stream of points. At the end of the input, the index i is revealed. We add another $k-1$ free points (*i.e.*, points that can be placed anywhere besides the circles) to the stream. The purpose of the $k-1$ points is to build a convex set that includes the circles and uses exactly one point on one of the circles.

Figure 3.6 gives us an example of such a construction. Again, the solid squares are the points we add to the stream. The example input bit vector is $S = 0001110011$. The index $i = 0$ and four free points p_1, p_2, p_3, and p_4 are placed such that the

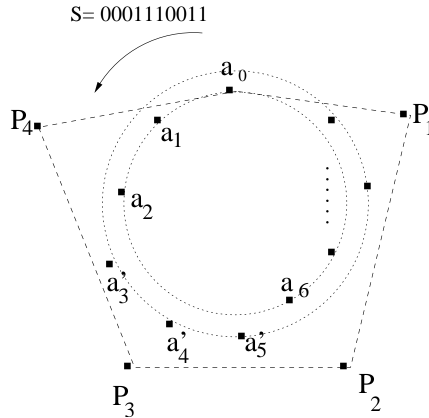

Figure 3.6: Convex hull using only one point on the circles

convex hull of the point set will be p_1, p_2, p_3, p_4 and a_0 or a'_0. In the example, the hull consists of $\{p_1, p_2, p_3, p_4, a_0\}$.

There will be only one point on the hull that is not in the set $\{p_1, p_2, p_3, p_4\}$. Once the convex hull is found, we can locate this point and deduce the bit value of S_i by calculating the distance from this point to the center of the circles.

Thus an algorithm that solved the k-promised convex-hull problem could be used to solve the index problem as well. $\square$

3.6 Conclusions

The study of stream algorithms for basic problems in computational geometry is important for better understanding the computational power of the streaming model. We initiate the study of computational-geometry problems in the streaming and sliding-window models. In particular, we study the diameter problem in the streaming and the sliding-window models. We show that exact computation cannot be achieved using $o(n)$ bits of space. We also devise approximation algorithms that give results with bounded error but use much smaller space.

Subsequent to our work in [40, 41], more computational-geometry problems have been studied and algorithms and lower bounds provided in [53, 27, 56]. On the other hand, there are several problems that are still open. For example, we use a sliding-window model that has a fixed window size. In some applications, it is desirable that the window size vary or be approximated, as in the work on elastic windows by Shasha and Zhu [83]. In general, we believe that it would be interesting to study computational-geometry problems in models with variable window size.

Chapter 4

Massive Data Streams in Graph Theory

4.1 Massive Graphs and Streaming

Many real world networks, *e.g.*, telecommunications networks and the World Wide Web, can be modeled as massive graphs. Also in applications such as structured data mining, the relationships among the data items in the data set are represented as graphs. Many application problems can be modeled as graph problems. It is important to investigate graph-theoretic problems in order to understand the streaming computational model.

We consider undirected graphs in this chapter. The graphs can be weighted—*i.e.*, there can be a weight function $w : E \rightarrow \mathbb{R}^+$ that assigns a non-negative weight to each edge. Unless stated otherwise, in this chapter, we denote by $G(V, E)$ an undirected, unweighted graph G with vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and edge set $E = \{e_1, e_2, \ldots, e_m\}$. Note that n is the number of vertices and m the number of edges.

Definition 4.1 (Graph Stream). *A graph stream is a sequence of edges* $e_{i_1}, e_{i_2}, \ldots, e_{i_m}$*, where* $e_{i_j} \in E$ *and* $i_1, i_2, \ldots, i_m$ *is an arbitrary permutation of* $[m] = \{1, 2, \ldots, m\}$.

While an algorithm goes through the stream, the graph is revealed one edge at a time. This definition generalizes the streams of graphs in which the adjacency matrix or the adjacency list is presented as a stream. In a stream in the adjacency-matrix or adjacency-list model, the edges incident to each vertex are grouped together. Our model of graph streams is more general and can account for graphs such as call graphs where the edges might be generated in any order.

Recall that the salient features of a streaming algorithm are:

1. sequential access to the data stream;
2. small workspace compared to the length of the stream;
3. fast per-element processing time;
4. a small number of passes through the stream.

We remark that while previous streaming algorithms share these features, the quantities that define "small" in features (2) and (4) vary from algorithm to algorithm. For example, most streaming algorithms use polylog$n$ space on a stream of length n. The streaming-clustering algorithm in [51], however, uses $O(n^\epsilon)$ space. Also, the "streaming"[1] algorithm in [30] uses space $O(\sqrt{n})$. Most streaming algorithms access the input stream in one pass, but there are also multiple-pass algorithms [67].

Therefore, we give a more general definition of the streaming model in Chapter 2. Recall that there are three important complexity measures in our definition: the workspace size, the per-element processing time, and the number of passes that the algorithm needs to go through the stream. We consider the following ranges

[1]Instead of "streaming," the authors of [30] use the term pass-efficient algorithms.

for these complexity measures: the workspace size is bounded by $O(n^\epsilon)$. The per-element processing time is $O(n^\delta)$, and polylog(n) is preferred. (ϵ and δ are two small constants.) The number of passes is $O(1)$, and a one-pass algorithm is preferred.

We view the three measures as three parameters of the streaming model. We now demonstrate that for massive graphs, it is necessary to examine the streaming model with respect to different values of the parameters. In particular, the polylog-space streaming model is too limited for graph problems and we need more space. To see this, consider the following simple problem. Given a graph, determining whether there is a length-2 path between two vertices, x and y, is equivalent to deciding whether two vertex sets, the neighborhood of x and the neighborhood of y, have a nonempty intersection. Because set disjointness has linear-space communication complexity [59], the length-2 path problem is impossible in the polylog-space streaming model. See [16] for a more comprehensive treatment of finding common neighborhoods in the streaming model. Here we extend the above length-2 path problem to show the following:

Theorem 4.1. *For all $1 \leq k(n) = o(n)$, any streaming algorithm that $k(n)$-approximates the distance between a fixed pair of vertices in a constant number of passes must use $\Omega(n)$ bits of space.*

Proof. Consider an n-vertex graph and two fixed vertices x and y of the graph. The graph also includes a path $(x, p_1), (p_1, p_2), \ldots, (p_t, y)$, and another set of vertices U, where $t = \Theta(n)$ and $|U| = \Theta(n)$. There are edges between x and some of the vertices in U. Let U_x denote the set of neighbors of x in U. There are also edges between y and some of the vertices in U. Let U_y denote the set of neighbors of y in U. (See Figure 4.1 for an example.) Note that any streaming algorithm that can $k(n)$-approximate the distance between x and y is also able to decide whether the two

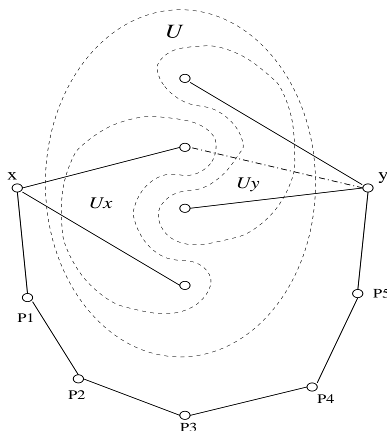

Figure 4.1: The distance between x and y is 6 if there is no common neighborhood of x and y. Otherwise (if the dash-dotted edge exists), the distance is 2.

sets U_x and U_y are disjoint, *i.e.*, whether there is a common neighborhood of x and y. Due to Buchsbaum *et al.* [16], the problem of deciding whether a pair of vertices in a graph has a common neighbor, in the streaming model using a constant number of passes, has space complexity $\Omega(n)$. □

We remark that our streaming model with the parameters taking a broader range of values is also a generalization of the semi-streaming model [68, 37], in which the space restriction is $O(n \cdot \text{polylog} n)$, and the pass-efficient model [30], in which the algorithms can go through the data stream multiple times. In [1], the authors also introduce the semi-external model for computations on massive graphs, *i.e.*, one in which the vertex set can be stored in memory, but the edge set cannot. However, their work addresses the problems in an external memory model in which random access to the edges, although expensive, is allowed. The authors of [61] argue that one of the major drawbacks of such algorithms, when applied to massive Web graphs, is their need to have random access to the edge set. Furthermore, in situations where the graph is revealed in a streaming fashion, such as a Web crawler exploring the Web graph, the streaming model is more suitable.

4.2 The Shortest-Path-Distance Problem and Graph Spanners

A path from x to y in an undirected graph $G = (V, E)$ is a sequence of edges $e_1, e_2, \ldots, e_k$ such that $e_1 = (x, v_1), e_2 = (v_1, v_2), e_3 = (v_2, v_3) \ldots, e_k = (v_{k-1}, y)$. For such a path P, we say P *connects* the vertices x and y. If G is an unweighted graph, the length of the path P is the number of edges on the path. If G is a weighted graph with weight function w, the length of the path is the sum of the weights of the edges on the path, *i.e.*, $\sum_{e \in P} w(e)$. The shortest-path distance between x and y is the length of the shortest path that connects x and y. Note that there may be more than one such shortest path. Because the shortest-path distance is the only distance considered in this chapter, in many places, we will omit the phrase "shortest path" and simply call it the distance between x and y.

The shortest-path problem is a fundamental problem in graph theory. It has been intensively studied and there are many variations of the problem. For example, the single-source, shortest-paths problem (SSSP) asks to find the shortest paths from a particular vertex to all other vertices in the graph. The all-pairs, shortest-paths problem (APSP) asks to find the shortest paths between all pairs of vertices. The associated shortest-path-distance problems ask to find those distances. It is not required, in these problems, to compute the paths.

There are two major approaches for computing shortest-path distances. One approach involves matrix computation. The graphs are represented as adjacency matrices, and the distances are computed using matrix multiplications. It is difficult to adapt such an approach to the streaming model. This is so because it is difficult to perform exact matrix computation in a streaming fashion. Moreover, although there are approximate matrix computations in the streaming model, they do not guarantee

that all the entries in the approximation matrix are within the error bound. That is, the computations ensure that the norm of the approximation matrix is close to the norm of the true result. However, an individual entry in the approximation matrix can be quite far away from the true value. Clearly, such an approximation is not suitable for computing graph distances.

The second approach is combinatorial. Although there are many algorithms that fall into this category, most of them share one common feature: that is, they employ subroutines that resemble breadth first search (BFS). For an unweighted, undirected graph, a BFS may start from a given vertex. It first finds all the vertices that are one hop, *i.e.*, one edge, away from this vertex. Then it continues to find all the vertices that are two hops away, then all the vertices that are three hops away, etc. During this process, it also builds a BFS tree for the graph. The BFS tree is rooted at the starting vertex. The vertices at depth i in the tree, *i.e.*, at distance i from the root of the tree, are also the vertices that are at distance i from the starting vertex in the original graph. If we view the vertices at the same distance from the given vertex as a layer, the BFS explores the graph in a layer-by-layer fashion. The $(i + 1)$-st layer is discovered after the i-th layer has been discovered.

Consider the algorithms for computing single-source, shortest-path distances, for example Djkstra's algorithm and the Bellman-Ford algorithm. Although technically more involved, they bear a certain similarity to BFS. These algorithms modify the search such that they can deal with other properties of the graphs, such as weighted edges, negatively weighted edges, etc. If we follow the algorithms in their processes of finding the shortest paths, we observe that the vertices are explored in a BFS-like fashion. That is, the vertices closer to the starting vertex are explored (settled) before the vertices that are farther from the starting vertex. In addition to exact computations, BFS-like explorations are commonly seen in algorithms that compute

the approximations of the all-pairs, shortest-path distances. For example, the algorithms in [76, 4] all use BFS-like subroutines.

Intuitively, BFS does not fit well with streaming. In the worst case, one may need to go through the whole stream of edges to grow one layer of the BFS tree. This happens when the edges that lead to the vertices on the next layer are scattered along the stream. The complete discovery of a whole layer then requires the algorithm to examine the whole stream. Because the algorithm does not have enough space to store all the edges, a depth-k BFS tree, in the worst-case scenario, will cost k passes through the stream. This is a basic difference between graph-distance computations in the traditional model and in the streaming model. This makes it difficult to adapt the traditional algorithms to the streaming model.

On the other hand, we observe that there is a structure called a *graph spanner* that is used widely in approximations of graph distances. An undirected graph $G = (V, E)$ induces a *metric space* $\mathcal{U}$ in which the vertex set V serves as the set of points, and the shortest-path distances serve as the *distances* between the points. The graph spanner $G' = (V, H), H \subseteq E$, is a *sparse skeleton* of the graph G whose induced metric space $\mathcal{U}'$ is a close approximation of the metric space $\mathcal{U}$ of the graph G. That is, the distance between two vertices in G' is not far from the distance between the same two vertices in G. For example, a subgraph $G' = (V, H)$, $H \subseteq E$ is a (multiplicative) *t-spanner* of the graph G, if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq t \cdot dist_G(u, v)$ (where $dist_G(u, v)$ stands for the distance between the vertices u and v in the graph G). The *stretch factor* of a spanner is the parameter(s) that determines how close the spanner approximates the distances in the original graph, *e.g.*, in the case of a t-spanner, the parameter t.

In the computation of graph distances, because a graph spanner is a sparse skeleton, a straightforward algorithm that runs on the spanner can be more efficient than

a sophisticated algorithm that runs on the original graph. Therefore, an efficient spanner construction often leads to efficient graph-distance approximations: One can construct a graph spanner first and, using the spanner, compute the approximate distances in a straightforward way. We observe that a similar approach can be applied in the streaming model if there is a streaming algorithm that constructs the spanner. In this case, a spanner is a sketch in the streaming model. A streaming algorithm constructs the sketch (the spanner) first, then computes the distance approximation using this sketch. As we discussed in Chapter 2, a streaming algorithm may perform pre- or post-processing. During such processing the algorithm has access to the workspace but not to the input stream. These are essentially computations in the traditional model. Therefore, the algorithm can use BFS-like subroutines during the post-processing, when it approximates the graph distances using the sketch (the spanner). In other words, the streaming distance approximation first constructs a spanner, then computes the distances in a straightforward way on the spanner. This approach still needs BFS-like subroutines. However, the BFS is performed on the sketch in the workspace, not on the graph in the stream.

Unfortunately, many algorithms for spanner construction also require multiple BFS-like explorations on the graph [24, 32]. In these spanner constructions, the algorithms often need distance information about some part of the graph. And BFS-like explorations are used to compute such distance information. This situation happens in many algorithms and we will encounter it repeatedly in this chapter when we compare our constructions to others. Because BFS is difficult in the streaming model, the sketch-based approach works only if there are *novel* spanner construction methods that either constrain the depth of the BFS exploration to be constant or totally avoid BFS. We present such constructions in this chapter. This is the heart of our work and is our main contribution to the knowledge of graph-distance

computations in the streaming model.

Before presenting our two main algorithms, we first show a simple streaming algorithm that constructs a particular spanner. The algorithm is an adaptation of the construction in [7]. It displays a certain connection between the girth of a graph and the spanner. (The *girth* of a graph is the length of the shortest cycle in the graph.) However, we remark that the algorithm needs more than $O(n)$ time to process an edge. Such a processing time is prohibitively high for the streaming model.

For an unweighted graph, a $(\log n / \log \log n)$-spanner S can be constructed in one pass in the streaming model: Because a graph whose girth is larger than k have at most $\lceil n^{1+2/(k-2)} \rceil$ edges [14, 31, 5], the algorithm constructs S by adding an edges in the stream to S if the edge does not cause a cycle of length less than $\log n / \log \log n$ in the spanner S constructed so far. Note that if we discard an edge, we do so because it causes a cycle of length less than $\log n / \log \log n$. That is, in S, there is a path P of length at most $\log n / \log \log n$ that connects the two endpoints of this edge. Any shortest path in the original graph that uses this edge can be replaced by a path in S that uses P. Therefore, S is a $\log n / \log \log n$ spanner of the original graph. For a weighted graph, however, the construction in [7] requires sorting the edges according to their weights, which is difficult in the streaming model. Instead of sorting, we use a geometric grouping technique to extend the spanner construction for unweighted graphs to a construction for weighted graphs. This technique is similar to the one used in [25]. Let ω_{min} be the minimum weight and let ω_{max} be the maximum weight. We divide the range $[\omega_{min}, \omega_{max}]$ into intervals of the form $[(1+\epsilon)^i \omega_{min}, (1+\epsilon)^{i+1} \omega_{min})$ and round all the weights in the interval $[(1+\epsilon)^i \omega_{min}, (1+\epsilon)^{i+1} \omega_{min})$ down to $(1+\epsilon)^i \omega_{min}$. For each induced graph $G^i = (V, E^i)$, where E^i is the set of edges in E whose weight is in the interval $[(1+\epsilon)^i \omega_{min}, (1+\epsilon)^{i+1} \omega_{min})$, a spanner can be constructed

in parallel using the above construction for unweighted graphs. The union of the spanners for all the G^i, $i \in \{0, 1, \ldots, \log_{(1+\epsilon)} \frac{\omega_{max}}{\omega_{min}} - 1\}$, forms a spanner for the graph G. Note that this can be done without prior knowledge of ω_{min} and ω_{max}. Our goal is to break the range $[\omega_{min}, \omega_{max}]$ into a small number of intervals. Given any value $\omega \in [\omega_{min}, \omega_{max}]$, we can use the set of intervals of the form $[(1+\epsilon)^i \omega, (1+\epsilon)^{i+1}\omega)$ and $[\frac{\omega}{(1+\epsilon)^{i+1}}, \frac{\omega}{(1+\epsilon)^i})$. Therefore, we can determine the intervals without the prior knowledge of ω_{min} and ω_{max}.

Theorem 4.2. *For $\epsilon > 0$, and a weighted, undirected graph on n vertices, whose maximum edge weight, ω_{max}, and minimum edge weight, ω_{min}, there is a streaming algorithm that constructs a $(1+\epsilon)\log n$-spanner of the graph in one pass. The algorithm uses $O(\log_{1+\epsilon} \frac{\omega_{max}}{\omega_{min}} \cdot n \log n)$ bits of space and the worst-case processing time for each edge is $O(n)$.*

Once we have the spanner, the distance between any pair of vertices can be approximated by computing their distance in the spanner. The diameter of the graph can be approximated by the spanner diameter too. Note that if the girth of an unweighted graph is larger than k, it can be determined exactly in a k-spanner of the graph. The construction of the $\log n / \log\log n$-spanner thus provides a $\log n / \log\log n$-approximation for the girth.

4.3 Distance Approximation in Multiple Passes

4.3.1 Graph Spanners and Distributed Computing

Spanners are found particularly useful in the following (henceforth called *distributed*) model of computation. In this model every vertex of an n-vertex graph $G = (V, E)$ hosts a processor with *unbounded computational power* but only *limited knowledge*. Specifically, it is assumed that in the beginning of the computation every processor v knows only the identities of its *neighbors*. The communication is *synchronous* and proceeds in *discrete pulses* called *rounds*. In each round each processor is allowed to send short (possibly different) messages to all its neighbors. The (worst-case) *running time* of a distributed algorithm is the (worst-case) number of rounds required for the algorithm to complete its execution. Spanners serve as an important tool in the design of distributed algorithms, and particularly, they are used for routing [70, 24], for constructing synchronizers [70, 10], and for computing almost-shortest paths [32].

For all these applications it is crucially important that the spanner be part of the original network so that the processors can communicate over its links. In particular, the processors can execute over the links of a spanner any protocol that was designed for arbitrary networks. Also, since a spanner approximates the distances in the original network, the execution of a protocol on a spanner is almost as time-efficient as its execution on the original network (spanned by the spanner). However, since spanners are typically *much sparser* than the networks they span, an execution of a protocol on a spanner is typically *much more communication-efficient* than the corresponding execution on the original network. These properties make spanners extremely valuable in the design of distributed protocols, and raise the problem of designing efficient distributed protocols for constructing spanners with good parameters.

In the 1990s most research on spanners and their applications focused on span-

ners whose metric space distorts the original metric space by at most a constant *multiplicative* factor, *i.e.*, multiplicative spanners. A fundamental theorem concerning multiplicative κ-spanners, that was proven by Peleg and Schäffer [69], says that for every n-vertex graph $G = (V, E)$ and a positive integer $\kappa = 1, 2, \ldots$, there exists a κ-spanner with $n^{1+O(\frac{1}{\kappa})}$ edges, and that this is the best possible, in terms of the number of the edges, up to the constant hidden by the O-notation.

More recently, Elkin and Peleg [34] studied a more general notion of (α, β)-*spanner*: a subgraph $G' = (V, H)$ of the graph $G = (V, E)$ is an (α, β)-*spanner* of the graph G if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq \alpha \cdot dist_G(u, v) + \beta$. (We also call an (α, β)-spanner an additive spanner.) They have shown that for every n-vertex graph $G = (V, E)$, there exists a $(1 + \epsilon, \beta)$-spanner $G' = (V, H)$ of G with $O(n^{1+1/\kappa})$ edges, where $\kappa = 1, 2, \ldots$, $0 < \epsilon < 1$ and β is a function of κ and ϵ. This result shows that the tradeoff of Peleg and Schäffer [69] can be drastically improved if one is concerned only with approximating the distances that are large.

While the proof of Elkin and Peleg [34] is not known to translate to an efficient distributed algorithm, Elkin in [32] gave an alternative proof of this theorem, which, though providing somewhat inferior constants, translates directly into efficient algorithms. The latter algorithms enabled [32] to use $(1 + \epsilon, \beta)$-spanners for efficient algorithms for computing almost-shortest paths from s sources.

It was shown in [32] that for every n-vertex graph $G = (V, E)$, positive integer $\kappa = 1, 2, \ldots$, and positive numbers $\epsilon, \rho > 0$, there exists a distributed algorithm that constructs $(1 + \epsilon, \beta)$-spanners with $O(n^{1+1/\kappa})$ edges in time $O(n^{1+\rho})$ with message complexity $O(|E| \cdot n^{\rho})$. Note that while the message complexity of this result is near-optimal, its running time is prohibitively large. In this work [35], we drastically improves this running time and devises a *randomized* distributed algorithm that constructs $(1+\epsilon, \beta)$-spanners with $O(n^{1+1/\kappa})$ edges in time $O(n^{\rho})$, and with message

complexity $O(|E| \cdot n^{\rho})$. Note that the message complexity of our algorithm is no worse than the message complexity of the algorithm of [32], and the parameters of the constructed spanners are also essentially the same as in the result of [32]. This result directly translates to an improved distributed algorithm for computing almost-shortest paths from s sources.

We remark that both our algorithm and the algorithm of [32] can be adapted to the *asynchronous* model of distributed computation in a rather straightforward way by using the synchronizers of [9]. The parameters of the obtained asynchronous algorithms are essentially the same as the parameters of the synchronous algorithms. Furthermore, our algorithm can also be easily extended to a parallel implementation that runs in $O(\log n + (|E| \cdot n^{\rho} \log n)/p)$ time using p processors in the EREW PRAM model. In particular, when the number of processors, p, is at least $|E| \cdot n^{\rho}$, the running time of the algorithm is $O(\log n)$. This is the first known parallel algorithm for constructing sparse $(1 + \epsilon, \beta)$-spanners.

Our distributed algorithm for constructing $(1 + \epsilon, \beta)$-spanners builds upon the previous work of [32]. The algorithm of [32], like our algorithm, uses extensively a subroutine for constructing *neighborhood covers* [8, 24, 32] (see Section 4.3.3 for its definition). In fact, both algorithms invoke this subroutine on many subgraphs of the original graph. The best distributed algorithm for constructing neighborhood covers that was available when the work of [32] was done is the algorithm of Awerbuch *et al.* [8]. Recently a significantly more efficient subroutine for computing neighborhood covers was devised in [33]. However, plugging the subroutine of [33] into the algorithm of [32] does not result in a sublinear algorithm for constructing $(1 + \epsilon, \beta)$-spanners, because the recursive invocations of the subroutine for constructing covers are implemented almost *sequentially* in the algorithm of [32]. The main technical difficulty that we had to overcome in this work is the *parallelization* of these recursive

invocations. The latter task requires far more elaborate analysis of the algorithm, because we have to show that no edge is simultaneously employed by more than a certain number of different subroutines.

4.3.2 Streaming Spanner Construction

Our distributed spanner construction can be adapted to a *streaming algorithm* for constructing $(1+\epsilon,\beta)$-spanners. The streaming algorithm uses a *constant number of passes* and $O(n^{1+1/\kappa}\cdot\log n)$ bits of space, and it has $O(n^{\rho})$ processing time per edge. ($\rho>0$ is an arbitrarily small control parameter. It affects the number of passes of the algorithm and the additive term of the constructed spanner.) This result, in turn, directly gives rise to a streaming algorithm with the same complexity parameters (number of passes and space) that given an input graph computes almost-shortest-path distances between all pairs of vertices of the graph. Note that storing the results of the computation — *i.e.*, the all-pairs, shortest paths and the all-pairs, shortest distances — may require space much larger than that required to store the spanner. On the other hand, the algorithm can output each path/distance in the result once it has the spanner. Hence, the algorithm need not store the results, and the total memory space required is no larger than the space taken by the spanner. (This also applies to our one-pass streaming algorithm in the next section.)

Note that the space complexity $O(n^{1+1/\kappa}\cdot\log n)$ of this streaming algorithm is not far from the optimal one, since we have shown in Section 4.1 that for all $1\leq k(n)=o(n)$, any streaming algorithm that $k(n)$-approximates the distance between a fixed pair of vertices in a constant number of passes must use $\Omega(n)$ bits of space.

We remark that our adaptation of the algorithm of [33] for constructing neighborhood covers for the streaming model is one of the first existing *streaming algorithms*

for clustering in graphs. Despite the fact that the clustering problem is extremely well studied in different disciplines, we are aware of only a very few previous streaming algorithms [51, 21] for clustering metric spaces. However, to the best of our knowledge, these algorithms are not applicable to the problem of clustering in graphs, and hence our result is incomparable with those of [51, 21].

Because the streaming algorithm is an adaptation from our distributed algorithm that constructs the same $(1+\epsilon, \beta)$-spanner, we will introduce and analyze the distributed version of our algorithm first and then adapt it to the streaming model.

4.3.3 Fast Distributed Spanner Construction

First, we introduce some notation. Let $G = (V, E)$ be an unweighted undirected graph. Denote by $dist_G(u, w)$ the *distance* between two vertices u and w in the graph G, that is, the length of the shortest path between them. For two subsets of vertices $V', V'' \subset V$, the distance in G between V' and V'', $dist_G(V', V'')$, is the shortest distance between a vertex in V' and a vertex in V'', *i.e.*, $dist_G(V', V'') = min\{dist_G(u, w) \mid u \in V', w \in V''\}$. Let $diam(G)$ denote the *diameter* of the graph G, *i.e.*, $diam(G) = max_{u,v \in V} dist_G(u, v)$.

Given a subset $V' \subseteq V$, denote by $E_G(V')$ the set of edges in G *induced* by V', *i.e.*, $E_G(V') = \{(u, w) \mid (u, w) \in E \text{ and } u, w \in V'\}$. Let $G(V') = (V', E_G(V'))$. Denote by $\Gamma_k(v, V')$ the *k-neighborhood* of vertex v in the graph $G(V')$, *i.e.*, $\Gamma_k(v, V') = \{u \mid u \in V' \text{ and } dist_{(V', E_G(V'))}(u, v) \leq k\}$. The diameter of a subset $V' \subseteq V$, denoted by $diam(V')$, is the maximum pairwise distance in G between a pair of vertices from V'. For a collection $\mathcal{F}$ of subsets $V' \subseteq V$, let $diam(\mathcal{F}) = \max_{V' \in \mathcal{F}}\{diam(V')\}$. Finally, unless specified explicitly, we say that an event happens with high probability if the probability is at least $1 - \frac{1}{n^{\Omega(1)}}$.

Our spanner construction utilizes graph covers. For a graph $G = (V, E)$ and

two integers $\kappa, W > 0$, a (κ, W)-*cover* [8, 24, 32] $\mathscr{C}$ is a collection of not necessarily disjoint subsets (or clusters) $C \subseteq V$ that satisfy the following conditions. (1) $\bigcup_{C \in \mathscr{C}} C = V$. (2) $diam(\mathscr{C}) = O(\kappa W)$. (3) The *size* of the cover $s(\mathscr{C}) = \sum_{C \in \mathscr{C}} |C|$ is $O(n^{1+1/\kappa})$, and furthermore, every vertex belongs to $\text{polylog}(n) \cdot n^{1/\kappa}$ clusters. (4) For every pair of vertices $u, v \in V$ that are at distance at most W from one another, there exists a cluster $C \in \mathscr{C}$ that contains both vertices, along with the shortest path between them. Note that many constructions of (κ, W)-cover will also build one BFS tree for each cluster in the cover as a by-product. The BFS tree spans the whole cluster and is rooted at one vertex in the cluster.

Quite a number of spanner-constructing algorithms use covers (e.g., [8, 24, 32]). In fact, for an unweighted graph, the union of the BFS-spanning trees of a $(\kappa, 1)$-cover is a κ-spanner. Moreover, it was demonstrated in [32] that covers can serve as a basis for $(1 + \epsilon, \beta)$-spanners as well, but in this case they are combined with other tools such as the shortest paths connecting some of the clusters in the covers. Currently it is not known whether $(1 + \epsilon, \beta)$-spanners can be built using a hierarchy of covers alone.

The algorithm in [32] (Algorithm 4.1) has a recursive structure. It first constructs a cover on the whole graph. The cover contains clusters of different sizes. The algorithm divides the clusters into two groups according to their sizes (the size of a cluster is the number of vertices in that cluster): a group $\mathscr{C}^H$ of large clusters and a group $\mathscr{C}^L$ of small clusters. The algorithm will be recursively invoked on the clusters that belong to the small-cluster group $\mathscr{C}^L$. We can assume that after the recursion returns, a spanner (with smaller estimation error) for each subgraph induced by a cluster in this group will be constructed. What remains then is to deal with the large clusters in the group $\mathscr{C}^H$. For these clusters, the algorithm finds the shortest path between pairs of clusters that are not far away from each other. These paths,

Algorithm 4.1. Recur-Spanner

The input to the algorithm is a graph $G = (V, E)$ and four parameters κ, ν, D, and Δ, where κ, D, and Δ are positive integers and $0 < \nu < 1$.

1. $\ell \leftarrow \lceil \log_{1/(1-\nu)} \log_{\Delta} |V| \rceil$.
2. *Construct a (κ, D^{ℓ})-cover $\mathscr{C}$ for G. Include the edges of the BFS-spanning trees of all the clusters in the spanner. Set $\mathscr{C}^H \leftarrow \{C \in \mathscr{C} \mid |C| \geq |V|^{1-\nu}\}$, $\mathscr{C}^L \leftarrow \mathscr{C} \setminus \mathscr{C}^H$. (We call the clusters from $\mathscr{C}^H$ "large clusters," and the clusters from $\mathscr{C}^L$ "small clusters.")*
3. **Interconnecting subroutine:** *For all pairs of clusters $C_1, C_2 \in \mathscr{C}^H$ s.t. $dist_G(C_1, C_2) \leq D^{\ell+1}$, compute one of the shortest paths between C_1 and C_2 in G. Include this path in the spanner.*
4. *For all clusters $C \in \mathscr{C}^L$, invoke Recur-Spanner with parameters: $G' = (C, E_G(C)), \kappa, \nu, D, \Delta$.*

together with the BSF trees of the clusters and the spanners for the subgraphs, form the spanner of the whole graph.

We cite from [32] the following theorem on the the size and the stretch factors, *i.e.*, the parameters ϵ and β, of the constructed spanner:

Theorem 4.3. [32] *Let G=(V,E) be an n-vertex graph, $\kappa = 1, 2, \ldots, 0 < \nu \leq 1/2 - 1/\kappa, D = 2, 3, \ldots, \Delta = 1, 2, \ldots$. let $\ell = \lceil \log_{1/(1-\nu)} \log_{\Delta} n \rceil$. Let H be the spanner constructed by Algorithm 4.1 invoked on the five-tuple $(G, \kappa, \nu, D, \Delta)$. Then H is a $(1 + O(\frac{\kappa\ell}{D}), O(\kappa D^{\ell}))$-spanner, and the size of H is $O((\Delta + D^{\ell+1}) n^{1+\frac{1}{\kappa\nu}})$.*

A detailed proof of the theorem is presented in [32]. To help the readers understand the connection between the covers and the constructed spanner, we give a short proof sketch here.

Proof. We consider the stretch factor of the spanner H first. Consider a pair of vertices and a shortest path between them in the original graph. We divide the

path into segments of length at most $D^{\ell+1}$. We argue that if the spanner can give a $(1+\epsilon', \beta)$-estimate of the lengths of the segments, it can also provide a $(1+\epsilon, \beta)$-estimate for the whole path, where $\epsilon = \epsilon' + \frac{\beta}{D^{\ell+1}}$. To see this, let $dist_G$ be the distance between the two vertices in the original graph and $dist_G = a \cdot D^{\ell+1} + b$ for some integer a and b. Let $dist_H$ be the distance between the same pair of vertices in the spanner H. For each segment, the spanner provides a $(1+\epsilon', \beta)$-approximation. Hence,

$$\begin{aligned} dist_H &\leq a(D^{\ell+1}(1+\epsilon') + \beta) + b(1+\epsilon') + \beta \\ &= a \cdot D^{\ell+1}(1+\epsilon' + \frac{\beta}{D^{\ell+1}}) + b(1+\epsilon') + \beta \\ &\leq (a \cdot D^{\ell+1} + b)(1+\epsilon' + \frac{\beta}{D^{\ell+1}}) + \beta \\ &= dist_G(1+\epsilon) + \beta \end{aligned}$$

Thus, we need to focus only on how the spanner estimates distances up to $D^{\ell+1}$. (This is also the reason that in step 3 of Algorithm 4.1, only pairs of large clusters that are close to each other are connected.) For two vertices u, w that are at most distance $D^{\ell+1}$ from each other, we consider again some shortest path P_{uw} between them. We can view this path as going from left to right. Let u' (w') be the leftmost (rightmost) vertex on P_{uw} that is contained in a large cluster. Note that the parts of the path P_{uw} going from u to u' and from w' to w are contained in subgraphs induced by some small clusters. In step 4 of Algorithm 4.1, after the recursion returns, we have a spanner (with smaller estimation error) for these subgraphs and we can use it to estimate the length of these two parts. What is left now is the part between u' and w'. Note that both u' and w' are contained in large cluster(s). Their distance can then be approximated by the shortest distance between the two clusters plus the

diameters of the two clusters. The possible overestimation caused by the diameters of the large cluster contributes to the additive part of the approximation.

Now we consider the size of the spanner. The spanner consists of the BFS trees of the clusters and the paths connecting the large clusters. Because the size of the cover is small, so is the number of edges in the union of the BFS trees of the clusters. Furthermore, the algorithm connects only large clusters that are close to one another. As there are not many large clusters (otherwise, the size of the cover would be too large), the number of edges added by the interconnecting paths is not too big. The size of the spanner is controlled by the parameters that determine the size of the cover, the number of the large clusters, and the maximum length of the inter-cluster paths. □

We observe that within each cluster, the cover construction and the interconnecting subroutine are local to the cluster. That is, these processes in one cluster are independent of the analogous processes in other clusters of the same level. Furthermore, the process of interconnecting the clusters of $\mathscr{C}^H$ is independent of the cover constructions within clusters of $\mathscr{C}^L$. Thus they can, in principle, be carried out in parallel.

Our distributed implementation of the algorithm for constructing $(1+\epsilon,\beta)$-spanners is different from that of [32] in two ways. First, the algorithm of [32] traverses a spanning tree of the entire graph, and performs the local subroutines for constructing neighborhood covers and interconnecting large clusters *sequentially*. Our algorithm avoids the traversal of the entire graph and performs these local subroutines *in parallel*. Additionally, the algorithm of [32] uses the algorithm of [8] to construct neighborhood covers, and this algorithm by itself requires superlinear time. Our algorithm instead uses a far more efficient algorithm due to [33] for constructing neighborhood covers. The latter algorithm has running time $O(n^\rho)$ for an arbitrarily

Algorithm 4.2. Cover

The input to the algorithm is a graph $G = (V, E)$ and two positive integer parameters κ and W.

1. $U_1 \leftarrow V$.
2. *In phase* $i = 1, 2, \ldots, \kappa$:
   (a) *Include each vertex* $v \in U_i$ *independently at random with probability* $p_i = min\{1, \frac{n^{i/\kappa}}{n} \cdot \log n\}$ *in the set* S_i *of phase* i.
   (b) *Each vertex* $s \in S_i$ *constructs a cluster by growing a BFS tree of depth* $d_{i-1} = 2((\kappa - i) + 1)W$ *in the graph* $(U, E(U))$. *We call* s *the* center *of the cluster and the set* $\Gamma_{2(\kappa-i)W}(s, U)$ *the* core set *of the cluster* $\Gamma_{2((\kappa-i)+1)W}(s, U)$.
   (c) *Let* R_i *be the union of the core sets of the clusters constructed in step (b). Set* $U_{i+1} \leftarrow U_i \setminus R_i$.

small $\rho > 0$. These two modifications enable us to provide a drastically improved distributed algorithm for constructing $(1 + \epsilon, \beta)$-spanners. We note that while replacing the subroutine of [8] for constructing neighborhood covers by an analogous subroutine from [33] is done in a straightforward way, the parallel execution of different local subroutines is technically much more involved, because we have to show that no edge is simultaneously employed by more than a certain number of different subroutines.

We next give the construction [24, 33] of (κ, W)-covers in Algorithm 4.2. We use this construction as a subroutine in our algorithm. This construction builds a (κ, W)-cover in κ phases. A vertex v in graph G is called *covered* if there is a cluster $C \in \mathscr{C}$ such that $\Gamma_W(v, V) \subseteq C$. Let U_i be the set of uncovered vertices at phase i. At the beginning, $U_1 = V$. At each phase i, a subset of vertices is covered and removed from U_i.

Theorem 4.4. [33] *Given an unweighted, undirected n-vertex graph, Algorithm 4.2*

constructs a (κ, W)*-cover such that, with high probability, every vertex is included in* $O(\kappa n^{1/\kappa} \cdot \log n)$ *clusters of the cover. The construction requires* $O(\kappa^2 n^{1/\kappa} W \cdot \log n)$ *rounds of distributed computation.*

We now present our distributed algorithm for constructing spanners that uses the above cover construction as a subroutine. Given a cluster C, let $\mathscr{C}(C)$ be the cover constructed for the graph $(C, E_G(C))$. For a cluster $C' \in \mathscr{C}(C)$, we define $Parent(C') = C$.

An execution of the algorithm can be divided into ℓ stages (levels). The original graph is viewed as a cluster on level 0. The algorithm starts level 1 by constructing a cover for this cluster. Recall that a cover is also a collection of clusters. The clusters of $\cup\mathscr{C}(C)$, where the union is over all the clusters C on level 0, are called *clusters on level 1*, and we denote the set of those clusters by $\mathscr{C}_1$. If a cluster $C \in \mathscr{C}_1$ satisfies $|C| \geq |Parent(C)|^{1-\nu}$, we say that C is a *large cluster* on level 1. Otherwise, we say that C is a *small cluster* on level 1. We denote by $\mathscr{C}_1^H$ the set of large clusters on level 1 and $\mathscr{C}_1^L$ the set of small clusters on level 1. Note that the cover-construction subroutine (Algorithm 4.2) builds a BFS-spanning tree for each cluster in the cover. Our algorithm includes all the BFS-spanning trees in the spanner and then goes on to make interconnections between all pairs of clusters in $\mathscr{C}_1^H$ that are close to each other. After these interconnections are completed, the algorithm enters level 2. For each cluster in $\mathscr{C}_1^L$, it constructs a cover. We call the clusters in each of these covers the *clusters on level 2*. The union of all the level-2 clusters is denoted by $\mathscr{C}_2$. If a cluster $C \in \mathscr{C}_2$ satisfies $|C| \geq |Parent(C)|^{1-\nu}$, we say that C is a *large cluster* on level 2. Otherwise, we say that C is a *small cluster* on level 2. Again, we denote by $\mathscr{C}_2^H$ the set of large clusters on level 2 and $\mathscr{C}_2^L$ the set of small clusters on level 2. The BFS-spanning trees of all the clusters in $\mathscr{C}_2$ are included into the spanner and all the close pairs of clusters in $\mathscr{C}_2^H$ get interconnected by the algorithm. The

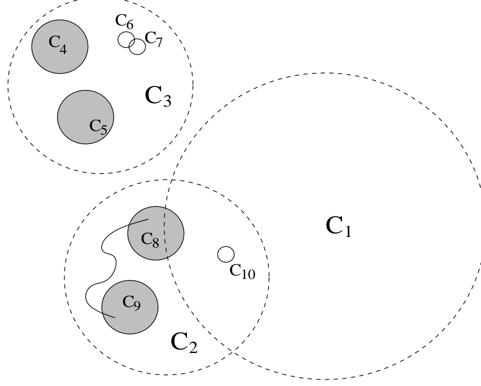

Figure 4.2: Example of clusters in covers.

algorithm proceeds in a similar fashion at levels 3 and above. That is, at level i, the algorithm constructs covers for each small cluster in $\mathscr{C}_{i-1}^L$, and interconnects all the close pairs of large clusters in $\mathscr{C}_i^H$. Similarly, we denote by $\mathscr{C}_i$ the collection of all the clusters in the covers constructed at level i, by $\mathscr{C}_i^H$ the set of large clusters of $\mathscr{C}_i$, and $\mathscr{C}_i^L$ the set of small clusters of $\mathscr{C}_i$. After level ℓ, each of the small clusters of level ℓ contains very few vertices and the algorithm can include in the spanner all the edges induced by these clusters. A formal description of the detailed algorithm is given below (Algorithm 4.3).

See Figure 4.2 for an example of covers and clusters constructed by the algorithm. The circles in the figure represent the clusters. $\mathscr{C}_1 = \{C_1, C_2, C_3\}$, $\mathscr{C}_1^H = \{C_1\}$ and $\mathscr{C}_1^L = \{C_2, C_3\}$. Note that for each cluster in $\mathscr{C}_1^L$, a cover is constructed. The union of the clusters in these covers forms $\mathscr{C}_2$, *i.e.*, $\mathscr{C}_2 = \{C_4, C_5, C_6, C_7, C_8, C_9, C_{10}\}$. The large clusters in $\mathscr{C}_2$ form $\mathscr{C}_2^H = \{C_4, C_5, C_8, C_9\}$ and the small clusters in $\mathscr{C}_2$ form $\mathscr{C}_2^L = \{C_6, C_7, C_{10}\}$. Also note that a pair of close, large clusters C_8 and C_9 is interconnected by a shortest path between them.

Note that the above algorithm is a synchronous protocol. By constructing and using a synchronizer [9], this protocol can be converted to an asynchronous protocol.

Algorithm 4.3. Spanner

The input to the algorithm is a graph $G = (V, E)$ on n vertices and four parameters κ, ν, D, and Δ, where κ, D, and Δ are positive integers and $0 < \nu < 1$.

1. $\mathscr{C}_0^L \leftarrow \{V\}$, $\mathscr{C}_0^H = \phi$.
2. *For level* $i = 1, 2, \ldots, \ell = \lceil \log_{1/(1-\nu)} \log_\Delta n \rceil$ *do*
   - (a) ***Cover Construction****: For all clusters* $C \in \mathscr{C}_{i-1}^L$*, in parallel, construct* (κ, D^ℓ)*-covers using Algorithm 4.2. (Invoking Algorithm 4.2 with parameters* κ *and* $W = D^\ell$*.) Include the edges of the BFS-spanning trees of all the clusters in the spanner. Set* $\mathscr{C}_i \leftarrow \bigcup_{C \in \mathscr{C}_{i-1}^L} \mathscr{C}(C)$, $\mathscr{C}_i^H \leftarrow \{C \in \mathscr{C}_i \mid |C| \geq |Parent(C)|^{(1-\nu)}\}$, $\mathscr{C}_i^L \leftarrow \mathscr{C}_i \setminus \mathscr{C}_i^H$.
   - (b) ***Interconnection****: For all clusters* $C' \in \mathscr{C}_i^H$*, in parallel, construct BFS trees in* $G(C)$*, where* $C = Parent(C')$*. For each cluster* C'*, the BFS tree is rooted at the center of the cluster, and the depth of the BFS tree is* $2D^h + D^{h+1}$*, where* $h = \lceil \log_{1/(1-\nu)} \log_\Delta |Parent(C')| \rceil$*. For all the clusters* C'' *whose center vertex is in the BFS tree, if* $C'' \in \mathscr{C}_i^H$ *and* $Parent(C'') = Parent(C')$*, add to the spanner the shortest path between the center of* C' *and the center of* C''*.*
3. *Add to the spanner all the edges of the set* $\bigcup_{C \in \mathscr{C}_{\ell+1}} E_G(C)$*.*

4.3.4 Analysis of Time and Message Complexity

In this section, we analyze the time and message complexity of Algorithm 4.3. In the distributed model, carrying out the cover constructions in parallel for all clusters of a certain family (as it is done in step (2a) of Algorithm 4.3) does not necessarily mean that the running time is equal to the running time of constructing one single cover. If all the cover constructions utilize the same edges, the running time of step (2a) may be no better than the time required for constructing these covers sequentially because of congestion. We show that this is not the case in our algorithm. Specifically, we show that each edge is utilized by only a small number of subroutines that construct covers. This is also true regarding the interconnecting subroutines in step (2b).

Given a cover $\mathscr{C}$, let $MS(\mathscr{C}) = max_{C\in\mathscr{C}}\{|C|\}$. The number of levels in the algorithm is $\ell = \lceil \log_{1/(1-\nu)} \log_{\Delta} n \rceil$. Throughout the analysis, assume that κ, ν, ℓ are constant, *i.e.*, independent of n.

Lemma 4.1. *For a vertex v and an index $i = 1, 2, \ldots, \ell$, with high probability, the number of clusters $C \in \mathscr{C}_i$ that contain v is $O(n^{\frac{1}{\kappa\nu}})$.*

Proof. By Theorem 4.4, with high probability, in a cover constructed for a cluster C, a vertex is contained in at most $O(\log|C| \cdot |C|^{1/\kappa})$ clusters. Also note that $MS(\mathscr{C}_i^L) \leq n^{(1-\nu)^i}$.

Let $\mathcal{M}_i(v) = \{C \in \mathscr{C}_i \mid v \in C\}$. We have:

$$\begin{aligned}
|\mathcal{M}_{i+1}(v)| &= \sum_{C\in\mathcal{M}_i(v)} O(\log|C| \cdot |C|^{1/\kappa}) \\
&= O(|\mathcal{M}_i(v)| \cdot \log(MS(\mathscr{C}_i^L)) \cdot MS(\mathscr{C}_i^L)^{1/\kappa}) \\
&= O(|\mathcal{M}_i(v)| \cdot \log n^{(1-\nu)^i} \cdot n^{\frac{(1-\nu)^i}{\kappa}}).
\end{aligned}$$

Given that the number of levels is $\ell = O(1)$, $|\mathcal{M}_i(v)| = O(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}) = O(n^{\frac{1}{\kappa\nu}})$. $\square$

Lemma 4.2. *With high probability, all the subroutines for constructing covers require altogether $O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1})$ time and $O(|E| \cdot n^{\frac{1}{\kappa\nu}})$ communication.*

Proof. By Lemma 4.1, for a fixed vertex v, and index $i = 1, 2, \ldots, \ell$, the vertex is contained in $O(n^{\frac{1}{\kappa\nu}})$ clusters on level i. Hence, at level i the vertex is explored $O(n^{\frac{1}{\kappa\nu}})$ times, and for a fixed edge e, the edge is explored $O(n^{\frac{1}{\kappa\nu}})$ times at level i. At level i, each BFS exploration for constructing a cover has depth at most

$$2\kappa \cdot D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}_{i-1}^L)} \leq 2\kappa \cdot D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}} = 2\kappa \cdot D^{\ell+1-i}.$$

Because each cover construction consists of $\kappa = O(1)$ phases, the overall time required for constructing all the covers on level i is $O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1-i})$. Hence, the overall time is: $\sum_{i=1}^{\ell} O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1-i}) = O(n^{\frac{1}{\kappa\nu}} \cdot D^{\ell+1})$.

Because each edge is explored $O(n^{\frac{1}{\kappa\nu}})$ times, and because there are $\kappa = O(1)$ phases and $\ell = O(1)$ levels, the overall number of messages that are sent for constructing all the covers is $O(|E| \cdot n^{\frac{1}{\kappa\nu}})$. $\square$

Lemma 4.3. *With high probability, for each level $i = 1, 2, \ldots, \ell$, each vertex $v \in V$ is explored by $O(n^{\frac{1}{\kappa\nu}+\nu})$ BFS explorations that are initiated by the interconnecting subroutines.*

Proof. Consider a cluster $C \in \mathcal{M}_{i-1}(v) \cap \mathscr{C}_{i-1}^L$. The size of the cover constructed for the cluster C is $O(\log|C| \cdot |C|^{1+1/\kappa})$. The large clusters in this cover have size at least $|C|^{(1-\nu)}$. Hence, the number of such clusters in the cover for C is $O(\frac{\log|C| \cdot |C|^{1+1/\kappa}}{|C|^{(1-\nu)}}) = O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu})$. In each cluster $C \in \mathcal{M}_{i-1}(v) \cap \mathscr{C}_{i-1}^L$ on level $(i-1)$, each of the $O(\log|C| \cdot |C|^{\frac{1}{\kappa}+\nu})$ large clusters on level i may explore the vertex v. Hence, the

overall number of BFS explorations that may visit the vertex v is at most

$$
\begin{aligned}
& \sum_{C \in \mathcal{M}_{i-1}(v)} O(\log |C| \cdot |C|^{\frac{1}{\kappa}+\nu}) \\
= \ & O(|\mathcal{M}_{i-1}(v)| \cdot \log MS(\mathscr{C}_{i-1}^L) \cdot MS(\mathscr{C}_{i-1}^L)^{\frac{1}{\kappa}+\nu}) \\
= \ & O(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^{i-1}}{\nu}} \cdot n^{(\frac{1}{\kappa}+\nu)\cdot(1-\nu)^{i-1}}) \\
= \ & O(n^{\frac{1}{\kappa\nu}+\nu}).
\end{aligned}
$$

For the last inequality, note that $n^{\frac{(1-\nu)^{i-1}}{\kappa\nu} - \frac{(1-\nu)^{i-1}}{\kappa}} = n^{\frac{(1-\nu)^i}{\kappa\nu}} \geq \log^c n$ for all constants c, all sufficiently large n, all constants $\kappa = 1, 2, \ldots$, and $\nu < 1$. (How large n must be depends on the other parameters.) □

Lemma 4.4. *With high probability, for an index $i = 1, 2, \ldots, \ell$, the overall time and communication complexities of all the interconnecting subroutines are $O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu})$ and $O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu})$, respectively.*

Proof. At level i, the depth of the BFS explorations that are required for interconnecting the large clusters is at most

$$
3D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}_{i-1}^L)+1} \leq 3D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}+1} = 3D^{\ell+2-i}.
$$

By Lemma 4.3, a vertex participates in $O(n^{\frac{1}{\kappa\nu}+\nu})$ interconnecting processes. Hence, the overall running time of all the interconnecting subroutines on level i is $O(n^{\frac{1}{\kappa\nu}+\nu} \cdot D^{\ell+2-i})$. Adding up the ℓ levels, we have $\sum_{i=1}^{\ell} O(n^{\frac{1}{\kappa\nu}+\nu} \cdot D^{\ell+2-i}) = O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu})$.

For an upper bound on the communication complexity at level i, note that each cluster C in $\mathscr{C}_i^H$ initiates a BFS exploration in the cluster $Parent(C) \in \mathscr{C}_{i-1}^L$. Following the analysis in Lemma 4.3, the number of large clusters in the cover $\mathscr{C}(Parent(C))$ is $O(\log |Parent(C)| \cdot |Parent(C)|^{\frac{1}{\kappa}+\nu})$. Because for every index $i =$

$1, 2, \ldots, \ell$, $|\mathcal{M}_i(v)| = O\left(\log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}\right)$, it follows that $e(\mathscr{C}_i) = \sum_{C \in \mathscr{C}_i} |E_G(C)|$ is $O\left(|E| \cdot \log^{O(1)} n \cdot n^{\frac{1}{\kappa}\frac{1-(1-\nu)^i}{\nu}}\right)$.

To summarize,

$$
\begin{aligned}
& \sum_{C \in \mathscr{C}_{i-1}^L} O(\log |C| \cdot |C|^{\frac{1}{\kappa}+\nu} \cdot e(C)) \\
= \; & O\left(\log(MS(\mathscr{C}_{i-1}^L)) \cdot MS(\mathscr{C}_{i-1}^L)^{\frac{1}{\kappa}+\nu} \cdot \sum_{C \in \mathscr{C}_{i-1}^L} e(C)\right) \\
= \; & O\left(\log^{O(1)} n \cdot n^{(\frac{1}{\kappa}+\nu)(1-\nu)^{i-1}} \cdot e(\mathscr{C}_{i-1})\right) \\
= \; & O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu}).
\end{aligned}
$$

Because there are ℓ levels, the overall communication complexity is $O(\ell \cdot |E| \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu})$. □

Now we are ready to prove the main result of this section.

Theorem 4.5. *Given an unweighted, undirected graph on n vertices and constants $0 < \rho, \delta, \epsilon < 1$, such that $\delta/2 + 1/3 > \rho > \delta/2$, there is a distributed algorithm that, with high probability, constructs a $(1+\epsilon, \beta)$-spanner of size $O(n^{1+\delta})$ for the graph, where $\beta = \beta(\rho, \delta, \epsilon) = O(1)$. The running time and the communication of the algorithm are $O(n^\rho)$ and $O(|E|n^\rho)$, respectively.*

Proof. Set $\Delta = n^{\delta/2}$, $\frac{1}{\kappa\nu} = \delta/2$, $\frac{1}{\kappa\nu} + \nu = \rho$. This gives $\nu = \rho - \frac{\delta}{2} > 0$, $\nu = O(1)$, $\kappa = \frac{2}{(\rho-\delta/2)\delta} = O(1)$, and $\ell = \log_{1/(1-\nu)} \log_\Delta n = \log_{1/(1-\nu)} \frac{2}{\delta} = O(1)$, satisfying the requirement that κ, ν, and ℓ are all constants.

Also set $D = O(\frac{\kappa\ell}{\epsilon})$. By Theorem 4.3, $(1+\epsilon)$ is the multiplicative stretch factor of the constructed spanner. This gives $\beta = O(\kappa D^\ell) = O(1)$, which is the additive term of the spanner. By Theorems 4.4 and 4.3, the size of our spanner is

$O\left((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}}\right)$. For sufficiently large n, $\Delta = n^{\delta/2} > D^{\ell+1}$; hence the spanner size is $O(n^{1+\delta})$. By Lemmas 4.2 and 4.4, the running time of the algorithm is $O(D^{\ell+2} \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(n^{\rho})$, and its communication complexity is $O(|E| \cdot n^{\rho})$. □

4.3.5 Parallel Implementation

Observe that the basic operation in both Algorithm 4.2 and Algorithm 4.3 is the construction of constant-depth BFS trees on the unweighted graph. Hence, we can use the straightforward parallelization of the BFS search for this operation [26, 24] and implement Algorithm 4.3 in the parallel computing models. We now briefly analyze the complexity of such an implementation in an EREW PRAM model.

The overall number of edges needed to be explored determines the amount of work performed by the parallel implementation. Note that in both the cover construction and the interconnection processes, an edge may be explored by multiple BFS-tree constructions. The algorithm runs in $\ell = O(1)$ iterations (levels). It invokes cover construction and interconnection at each level. For cover construction, by Lemma 4.1 and the fact that there are $\kappa = O(1)$ phases in the construction, an edge may be explored $O(n^{\frac{1}{\kappa\nu}})$ times. For interconnection, by Lemma 4.3, an edge may be explored $O(n^{\frac{1}{\kappa\nu}+\nu})$ times. Hence, the overall number of edges being explored is $O(|E| \cdot n^{\frac{1}{\kappa\nu}+\nu}) = O(|E| \cdot n^{\rho})$.

For a graph $G(V,E)$, recall that $\Gamma_k(u,V)$ is the k-neighborhood of the vertex u, *i.e.*, the set of vertices of distance at most k from u. Also recall that $E_G(\Gamma_k(u,V))$ is the set of edges induced by the set of vertices in the neighborhood $\Gamma_k(u,V)$. The following proposition is from [24]:

Proposition 4.1. [24] *For a graph $G(V,E)$, a set of integers $k_1,\ldots,k_r$ and a set of vertices $s_1,\ldots,s_r$, the computation of $\Gamma_{k_i}(s_i,V)$ for $i = 1,\ldots,r$ can be performed*

using p processors in time

$$O\left((\max_i k_i \log n) + \sum_{i=1}^{r} |E_G(\Gamma_{k_i}(s_i, V))|(\log n)/p\right).$$

Note that our parallel implementation runs in a constant number of iterations. The number of phases in each cover construction is also constant and so is the depth of each BFS exploration. Therefore, using the BFS search algorithm of [24], the parallel implementation of Algorithm 4.3 requires $O(\log n + (|E| \cdot n^{\rho} \log n)/p)$ running time using p processors in the EREW PRAM model. In particular, when the number of processors, p, is at least $|E| \cdot n^{\rho}$, the running time of the algorithm is $O(\log n)$.

4.3.6 Adaptation to the Streaming Model

In this section we adapt Algorithm 4.3 to the streaming model and devise an algorithm for computing all-pairs, almost-shortest paths in this model.

Leaving space limitations aside, it is easy to see that many distributed algorithms with time complexity T translate directly into streaming algorithms that use T passes. For example, a straightforward streaming adaptation of a synchronous distributed algorithm for constructing a BFS tree would be the following: in each pass over the input stream, the BFS tree grows one more level. An exploration of d levels would result in d passes over the input stream. On the other hand, there are cases in which the running time of a synchronous algorithm may not translate directly to the number of passes of the streaming adaptation. In the example of the BFS tree, if two BFS trees are being constructed in parallel, some edges may be explored by both constructions, resulting in congestion that may increase the running time of the distributed algorithm. On the other hand, for a streaming algorithm,

both explorations of the same edge can be done using only one pass over the stream.

We adapt Algorithm 4.2 for constructing covers to the streaming model. The streaming adaptation proceeds in κ phases. In each phase i, the algorithm passes through the input stream d_{i-1} times to build the BFS trees $\tau(v)$ of depth d_{i-1} for each selected vertex $v \in S_i$. The cluster and its core set can be computed during the construction of these BFS trees. Note that for any i, $d_{i-1} \leq 2\kappa W$. Hence,

Lemma 4.5. *With high probability, the streaming adaptation of Algorithm 4.2 constructs a (κ, W)-cover using $2\kappa^2 W$ passes over the input stream.*

Now, we briefly describe the adaptation of Algorithm 4.3 to the streaming model. The adapted algorithm is recursive, and the recursion has ℓ levels. At level i, a cover is constructed for each of the small clusters in $\mathscr{C}_{i-1}^{L}$ using the streaming algorithm for constructing covers described above. Because the processes of building BFS trees for constructing covers are independent, they can be carried out in parallel. That is, when the algorithm encounters an edge in the input stream, it examines its two endpoints. For each of the clusters in $\mathscr{C}_{i-1}^{L}$ that contains both endpoints, for each of the BFS-tree constructions in those clusters that has reached one of the endpoints, the algorithm checks whether the edge would help to extend the BFS tree. If so, the edge would be added to that BFS tree. After the construction of the covers is completed, the algorithm makes interconnections between close, large clusters of each cover. Again, the constructions of the BFS trees that are invoked by different interconnection subroutines are independent and can be performed in parallel.

Lemma 4.6. *With high probability, the streaming adaptation of Algorithm 4.3 requires $2\kappa^2 D^{\ell+1} + 3D^{\ell+2}$ passes over the input stream.*

Proof. Note that at level i, for cover construction, the value of W is bounded by

$$D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}^L_{i-1})} \leq D^{\log_{1/(1-\nu)} \log_\Delta n^{(1-\nu)^{i-1}}} \leq D^{\ell+1-i}.$$

By Lemma 4.5, the number of passes that are required for constructing a cover at level i is at most $2\kappa^2 D^{\ell+1-i}$.

At level i, the algorithm also invokes interconnecting subroutines. The depth of the BFS trees that are required for interconnection on recursion level i is at most $3D^{\log_{1/(1-\nu)} \log_\Delta MS(\mathscr{C}^L_{i-1})+1} \leq 3D^{\ell-i+2}$.

The overall number of passes is at most $\sum_{i=1}^{\ell}(2\kappa^2 D^{\ell+1-i} + 3D^{\ell-i+2}) \leq 2\kappa^2 D^{\ell+1} + 3D^{\ell+2}$. □

Lemma 4.7. *The space complexity of the streaming adaptation of Algorithm 4.3 is* $O\left(\log n \cdot (\Delta + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right)$.

Proof. The algorithm needs storage space for the following: (1) The algorithm stores the edges of the spanner. (2) The covers constructed by the algorithm are collections of clusters. The algorithm includes a vertex in a cluster by labeling the vertex with the ID of the cluster. Hence, for each vertex, the algorithm stores the IDs of the clusters to which the vertex belongs. (3) On each level i, the large clusters of $\mathscr{C}_i^H$ construct BFS trees for interconnection. Each BFS tree is constructed by labeling the vertices layer by layer using the ID of the initiating cluster. Hence, for each vertex, the algorithm also stores the IDs of clusters $C \in \mathscr{C}_i^H$ whose BFS exploration has visited/explored the vertex.

By Theorem 4.3 and Lemma 4.1, items (1) and (2) require $O((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}})$ and $O(n^{\frac{1}{\kappa\nu}})$ cells of memory, respectively. By Lemma 4.3, item (3) requires $O(n^{\frac{1}{\kappa\nu}+\nu})$ cells of memory for each level. We observe that once the interconnections are made, the algorithm will no longer need the BFS trees constructed for the interconnections.

The space used to store these BFS trees on level i can be reused on level $i+1$. Hence, the overall number of memory cells required by item (3) is $O(n^{\frac{1}{\kappa\nu}+\nu})$. Note that all the above quantities are given in terms of the number of edges and IDs. The space in terms of the number of bits is greater by at most a factor of $\log n$.

Hence, the overall space complexity is:

$$\begin{aligned} & O\left((\Delta + D^{\ell+1})n^{1+\frac{1}{\kappa\nu}} \cdot \log n\right) + O\left(n^{\frac{1}{\kappa\nu}} \cdot \log n\right) + O\left(n^{\frac{1}{\kappa\nu}+\nu} \cdot \log n\right) \\ = \; & O\left(\log n \cdot (\Delta + D^{\ell+1}) \cdot n^{1+\frac{1}{\kappa\nu}}\right). \end{aligned}$$

□

Lemma 4.8. *With high probability, the streaming adaptation of Algorithm 4.3 processes each edge using* $O(n^{\frac{1}{\kappa\nu}+\nu})$ *time.*

Proof. For a fixed index $i = 1, 2, \ldots, \ell$, on level i, during the construction of the cover, the algorithm may need to examine the vertex v for each of the clusters on level i that contain the vertex when it encounters the edge (u, v) in the stream. Let $T_{cover}(v)$ be the processing time for this purpose. By Lemma 4.1, $T_{cover}(v) = O(n^{\frac{1}{\kappa\nu}})$.

During interconnection, the algorithm may also need to examine the vertex v for each cluster $C \in \mathscr{C}_i^H$ whose BFS exploration visits v. Let $T_{interconnect}(v)$ be this processing time. By Lemma 4.3, $T_{interconnect}(v) = O(n^{\frac{1}{\kappa\nu}+\nu})$.

The overall time that is required to process the edge (u, v) is $2 \cdot (T_{cover}(v) + T_{interconnect}(v)) = O(n^{\frac{1}{\kappa\nu}+\nu})$. □

To summarize,

Theorem 4.6. *Given an unweighted, undirected graph on* n *vertices, presented as a stream of edges, and constants* $0 < \rho, \delta, \epsilon < 1$, *such that* $\delta/2 + 1/3 > \rho > \delta/2$, *there is a streaming algorithm that, with high probability, constructs a* $(1 + \epsilon, \beta)$*-spanner*

of size $O(n^{1+\delta})$. The algorithm accesses the stream sequentially in $O(1)$ passes, uses $O(n^{1+\delta} \cdot \log n)$ bits of space, and processes each edge of the stream in $O(n^{\rho})$ time.

Note that once the spanner is computed, the algorithm is able to compute all-pairs, almost-shortest paths and distances in the graph by computing the *exactly* shortest paths and distances in the spanner using the same space. Observe that for a pair of vertices, $u, v \in V$, the path $P_{u,v}$ that is computed by the algorithm satisfies the inequality $|P_{u,v}| \leq (1+\epsilon)d_G(u,v) + \beta$. Note also that this computation of the shortest paths in the spanner requires no additional passes through the input, and also, no additional space. (For the latter we assume that once computed, the paths are immediately output by the algorithm and are not stored; obviously, any algorithm that stores estimates of distances for all pairs of vertices requires $\Omega(n^2)$ space.) To summarize,

Corollary 1. *Given an unweighted, undirected graph on n vertices, presented as a stream of edges, and constants $0 < \rho, \delta, \epsilon < 1$, such that $\delta/2 + 1/3 > \rho > \delta/2$, there is a streaming algorithm that, with high probability, computes all-pairs, almost-shortest paths in the graph with error terms of $(1+\epsilon, \beta)$. The algorithm accesses the stream sequentially in $O(1)$ passes, uses $O(n^{1+\delta} \cdot \log n)$ bits of space, and processes each edge in the stream in $O(n^{\rho})$ time.*

4.3.7 Conclusion

We devised a distributed randomized algorithm that improves the distributed algorithm of [32]. Except for the need for randomization, our algorithm drastically reduces the running time at no other cost. Applying our algorithm leads to more efficient distance-approximation algorithms in the distributed setting. For example, the running time of the distributed algorithm for the s-source almost-shortest-path

problems in [32] can be improved using our algorithm. The adaptation of our algorithm to the streaming model provides a $(1 + \epsilon, \beta)$-approximation for the all-pairs, shortest-paths problem in this model. The algorithm uses only a constant number of passes.

4.4 Distance Approximation in One Pass

Although the algorithm presented in the previous section uses only a constant number of passes, it can be argued that the number of passes is still large. Most known streaming algorithms use only a small number of passes (such as one or two). Because it is expensive to take a pass through the input stream, and sometimes it is impossible to store the data stream for a second pass, a one-pass algorithm is always preferred. In this section, we present such an algorithm.

In particular, we devised a randomized streaming algorithm that constructs a $(2t+1)$-spanner for an unweighted, undirected graph in one pass. With high probability, the algorithm uses $O(t \cdot n^{1+1/t} \log^2 n)$ bits of space and processes each edge in the stream in $O(t^2 \cdot n^{1/t} \log n)$ time. Using this spanner, the all-pairs distances in the graph can be $(2t+1)$-approximated. Note that for $t = \log n / \log \log n$, the algorithm uses $n \cdot \text{polylog}(n)$ bits of space and processes each edge in $\text{polylog}(n)$ time. We show in [38] that, with $O(n^{1+1/t})$ bits of space, we cannot approximate the distance between u and v better than by a factor t even if we know the vertices u and v. Therefore, our algorithm is close to optimal.

One major difference between the algorithm here and the algorithms in the previous section is that the algorithm here produces a multiplicative spanner while the algorithm in the previous section gives an additive spanner. Recall that a subgraph $G' = (V, H)$ is a (multiplicative) t-spanner of the graph $G = (V, E)$ if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq t \cdot dist_G(u, v)$ (where $dist_G(u, v)$ stands for the distance between the vertices u and v in the graph G). A subgraph $G' = (V, H)$ of the graph $G = (V, E)$ is an (additive) (α, β)-spanner of the graph G if for every pair of vertices $u, v \in V$, $dist_{G'}(u, v) \leq \alpha \cdot dist_G(u, v) + \beta$.

In [75], Thorup and Zwick provide a construction of distance oracles for ap-

proximating distances in graphs. Although all-pairs, shortest-path distances can be approximated using this oracle, their oracle construction requires the computation of shortest-path trees for certain vertices. That is, they need to compute the exact distances between certain pairs of vertices in order to build a data structure from which the all-pairs, shortest-path distances can be approximated. The exact distance computations are done by BFS-like subroutines, and the depth of the BFS explorations are quite large. We explained at the beginning of this chapter that BFS does not work well with streaming because it may take k passes through the stream to construct a BFS tree of depth k. Hence, a straightforward adaptation of the approaches in [75] will not work in the streaming model. Moreover, in a one-pass algorithm, it is impossible even to perform BFS for constant depth. This leads to the second major difference between the algorithm in this section and the one in the previous section. The algorithm in the previous section still depends on BFS explorations, although of only constant depth, to construct the basic structures of the spanner, *i.e.*, the covers and the connections between some clusters in the covers. The algorithm in this section totally avoids BFS-like explorations and therefore only needs to go through the stream in one pass.

In what follows, we first present the streaming algorithm that constructs a $(2t+1)$-spanner for an unweighted undirected graph in one pass. We then extend this algorithm to construct $((1+\epsilon)\cdot(2t+1))$-spanners for weighted, undirected graphs, using the geometric grouping technique.

4.4.1 Cluster Structures for Distance Approximation

In Section 4.3, graph covers are heavily used for the spanner construction. One of the properties of a (κ, W)-cover is that if two vertices x and y are at distance at most W from each other, they will be included in a cluster in the cover. Therefore, BFS-like

exploration may be necessary in building the clusters because one needs to make sure that the vertices within a certain range are included in at least one cluster. In this section, we use a more relaxed cluster structure for spanner construction. Our clusters still cover the whole graph. However, it is no longer required that if two vertices x and y are within a certain distance they be included in one cluster of the cover.

The spanner is an edge subgraph of the original graph. We can view the construction of a spanner as the following process: Initially, we have a subgraph with all the vertices of the original graph but an empty edge set. We then select some of the edges of the original graph to include in the subgraph. We continue this until the subgraph becomes a spanner. As we have demonstrated in the simple spanner construction in Section 4.2, to make the subgraph H a t-spanner of the original graph G, for each edge that is not included, one needs to make sure that there is a path of length at most t in H that connects the two endpoints of the removed edge. This is also a goal of our algorithm in this section. As shown in the simple spanner construction, the straightforward way to achieve this goal is very expensive. Using the straightforward method, we need to check the existence of the length-t path each time we encounter an edge. This is equivalent to computing the distance between the two endpoints in the subgraph we have constructed so far, which has a rather large time complexity.

In this section, we use a better approach to this problem. We do not need to compute the distance between the two endpoints of an edge in order to decide whether we should include it in the spanner. Instead, we maintain a certain cluster structure on the graph. The cluster structure then assures the existence of the path of length at most t where they are needed. To see how this works, first consider building a $2t + 1$ spanner for the following type of graphs. Imagine a graph that consists

of two clusters of vertices C_1 and C_2. Assume that the diameter of the subgraph induced by each cluster is at most t. Further assume that after seeing some part of the edge stream, we have constructed a spanning tree of small diameter for each of the clusters. Now we need to decide for the coming edges whether they should be included in H. The choice is clear: if an edge connects two vertices in the same cluster (we call such an edge an intra-cluster edge), we discard it. We already have a spanning tree of the cluster that provides the required path connecting the two endpoints of the edge. We will keep the first edge that connects the two clusters, *i.e.*, an edge $e = (u, v)$ such that $u \in C_1$ and $v \in C_2$. (We call such an edge an inter-cluster edge.) After this, we can discard other inter-cluster edges. For such a discarded edge (u', v'), the path connecting $u' \in C_1$ and $v' \in C_2$ consists of the edge e as well as the paths on the spanning trees of the two clusters that connect u' to u and v' to v respectively. The length of the path connecting u' to u is at most t and so is the length of the path connecting v' to v. The total length of the path connecting the two endpoints of the discarded edge is then at most $2t + 1$.

For a general graph, we do not have such a cluster structure at hand. One needs to decompose the graph into clusters that maintain the above property, *i.e.*, the diameter of the subgraph induced by the vertices in each cluster must be small. This can be done if we are not concerned about the number of clusters in the decomposition. (After all, each vertex can form a cluster of its own, and the diameter of the induced subgraph is zero.) However, we cannot afford to have too many clusters in the decomposition for the following reason: in the process described above, it is possible that for each pair of clusters, we need to keep an edge connecting the pair. Hence, if we have too many clusters in the decomposition and we store the edges between each pair of them, there will be too many edges in the spanner constructed. We want to build a cluster structure on the graph with the following properties:

1. the number of clusters N_{cls} is not too large.

2. the diameter of the clusters $Diam_{cls}$ is not too large.

One can see that the two goals work against each other. It is easy to decompose a graph into many clusters of small diameter, and it is easy to decompose a graph into a few clusters of large diameter. The challenge lies in achieving these two goals together. The problem, however, also comes from there. It is impossible to have all combinations of N_{cls} and $Diam_{cls}$ for every graph. In particular, to construct a $(2t+1)$ spanner of small size, we want N_{cls} to be at most $\sqrt{n}$ and $Diam_{cls}$ to be at most t. Clearly, a decomposition with these two parameters is not always possible.

Fortunately, we do not need such a decomposition for the *whole graph*. Intuitively, if a graph is very dense, *i.e.*, contains many edges, either the graph has a small diameter or it contains a small number (compared to the number of vertices n) of subgraphs that are of small diameter and the majority of the graph's edges are contained in these subgraphs. We call the union of these subgraphs the *dense part* of the original graph. If we remove the dense part from the original graph, the remaining graph, whose diameter may be large, will not have too many edges. We call the remaining graph the *sparse part* of the original graph. Clearly, we can include the whole sparse part in the spanner and we can use many clusters to cover the sparse part because there are not many inter-cluster edges in the sparse part. Therefore, we only need to control the number of clusters in the decomposition of the dense part.

Previously, similar cluster structures were used in spanner constructions [8, 32, 13]. However, in [8, 32, 13], the constructions of these structures all employ an approach similar to BFS. For example, the cluster construction in [13] takes multiple phases and the algorithm needs to go through the whole edge set in each phase. Clearly, this would necessitate multiple passes over the input stream. Instead of

relying on BFS-like explorations, we have devised a randomized labeling scheme for constructing the clusters. By bypassing the BFS, this randomized construction enables our algorithm to run in one pass through the stream.

4.4.2 One-Pass Spanner Construction

Our algorithm labels the vertices of the graph while going through the stream of edges. We now describe the generation of the labels in detail. A label l is a positive integer. Given two parameters n and t, the set of labels L used by our algorithm is generated in the following way. Initially, we have the labels $1, 2, \ldots, n$. We denote by L^0 this set of labels and call them the *level* 0 labels. Independently, and with probability $\frac{1}{n^{1/t}}$, each label $l \in L^0$ will be selected for membership in the set S^0 and l will be marked as *selected*. From each label l in S^0, we generate a new label $l' = l + n$. We denote by L^1 the set of newly generated labels and call them level 1 labels. We then apply the above selection and new-label-generation procedure to L^1 to get the set of level 2 labels L^2. We continue this until the level $\lfloor \frac{t}{2} \rfloor$ labels $L^{\lfloor \frac{t}{2} \rfloor}$ are generated. If a level $i+1$ label l is generated from a level i label l', we call l the *successor* of l' and denote this by $Succ(l') = l$. The set of labels we will use in our algorithm is the union of labels of level $1, 2, \ldots, \lfloor \frac{t}{2} \rfloor$, *i.e.*, $L = \cup L^i$. Note that L can be generated before the algorithm sees the edges in the stream. But, in order to generate the labels in L, except in the case $t = O(\log n)$, the algorithm needs to know n, the number of vertices in the graph, before seeing the edges in the input stream. For $t = O(\log n)$, a simple modification of the above method can be used to generate L without knowing n, because the probability of a label's being selected can be any constant smaller than $\frac{1}{2}$.

At first glance it might appear that our labeling scheme resembles the sequences of sets initially constructed by the algorithm of Thorup and Zwick [75]. In our

construction the generation of the clusters of vertices depends on both the labels and the edge set. Thorup and Zwick, however, generate their sequence of sets independent of the edges. But this is just the first step in their algorithm. Subsequently, their algorithm computes the exact distance between a fixed vertex and each of the sets of vertices in the sequence. Such a step is very difficult in the streaming model. Our label generation and labeling scheme do not have such a requirement.

While going through the stream, our algorithm labels each vertex with labels chosen from L. Let $C(l)$ be the collection of vertices that are labeled l. We call the subgraph induced by the vertices in $C(l)$ a *cluster*, and we say that the label of the cluster is l. Note that each label defines a cluster. The top level labels determine the clusters in the decomposition of the dense part of the graph. The clusters determined by the lower level labels cover the sparse part of the graph.

The algorithm may label a vertex v with multiple labels; however, v will be labeled by at most one label from L^i, for $i = 1, 2, \ldots, \lfloor \frac{t}{2} \rfloor$. Moreover, if v is labeled by a label l, and l is selected, the algorithm also labels v with the label $Succ(l)$.

Denote by l^i a label of level i, *i.e.*, $l^i \in L^i$. Let $L(v) = \{l^0, l^{k_1}, l^{k_2}, \ldots, l^{k_j}\}$, $0 < k_1 < k_2 < \ldots < k_j < t/2$ be the collection of labels that has been assigned to the vertex v. Let $Height(v) = max\{j | l^j \in L(v)\}$ and $Top(v) = l^k \in L(v)$ s.t. $k = Height(v)$.

At the beginning of the algorithm, the set $L(v_i)$ contains only the label $i \in L^0$. The set $C(l) = \{v_l\}$ for $l = 1, 2, \ldots, n$ and is empty for other labels. $L(v)$ and $C(l)$ grow while the algorithm goes through the stream and labels the vertices. For each $C(l)$, our algorithm stores a rooted spanning tree $Tree(l)$, on the vertices of $C(l)$. For $l \in L^i$, the depth of the spanning tree is at most i, *i.e.*, the deepest leaf is at distance i from the root.

We say an edge (u, v) connects $C(l)$ and $C(l')$ if u is labeled with l and v is labeled

with l'. If there are edges connecting two clusters at level $\lfloor \frac{t}{2} \rfloor$, our algorithm stores one such edge for this pair of clusters. We denote by H the set of these edges stored by our algorithm.

To capture the inter-cluster edges in the sparse part of the graph, our algorithm stores another small set of edges. We denote by $M(v)$ the edges in this set that are assigned to the vertex v by our algorithm. The spanner constructed by the algorithm is the union of the spanning trees for all the clusters, $M(v)$ for all the vertices, and the set H. The detailed algorithm is given in Algorithm 4.4 and we proceed with its analysis.

4.4.3 Analysis

We first show that with high probability, $M(v)$ does not contain too many edges.

Lemma 4.9. *At the end of the stream, for all $v \in V$, $|M(v)| = O(t \cdot n^{1/t} \log n)$ with high probability.*

Proof. Let $M^{(i)}(v) \subseteq M(v)$ be the set of edges added to $M(v)$ during the period when $Height(v) = i$. Let $L(M(v)) = \bigcup_{(u,v)\in M(v)} L(u)$ be the set of labels that have been assigned to the vertices in $M(v)$. An edge (u, v) is added to $M(v)$ only in step 2(b)(ii). Note that in this case, $\lfloor \frac{t}{2} \rfloor \geq Height(u) \geq Height(v)$. Hence, the set $L_v(u)$ is not empty. Also by the condition in step 2(b)(ii), none of the labels in $L_v(u)$ appears in $L(M(v))$. Thus, by adding the edge (u, v) to $M(v)$, we introduce at least one new label to $L(M(v))$. $M^{(i)}(v)$ will then introduce a set B of distinct labels to $L(M(v))$. Furthermore, the size of B is at least $|M^{(i)}(v)|$. Note that the labels in B are not marked as selected. Otherwise, the algorithm would have taken step 2(b)(i) instead of step 2(b)(ii). Hence, the size of B follows a geometric distribution, *i.e.*, $\mathbf{Pr}(|B| = k) \leq (1 - \frac{1}{n^{1/t}})^k$. Thus, with high probability, $|M^{(i)}(v)| = O(n^{1/t} \log n)$.

Algorithm 4.4. Streaming_Spanner

The input to the algorithm is an unweighted, undirected graph $G = (V, E)$, presented as a stream of edges, and two positive integer parameters n and t.

1. *Set $H \leftarrow \phi$. Generate the set L of labels as described. $\forall\ v_i \in V$, label vertex v_i with label $i \in L^0$. If i is selected, label v_i with $Succ(i)$. Continue until we see a label that is not selected. Set $M(v_i) \leftarrow \phi$.*

2. *Upon seeing an edge (u, v) in the stream, if $L(v) \cap L(u) \neq \emptyset$, nothing needs to be done. Otherwise, consider the following cases:*

   (a) *If $Height(v) = Height(u) = \lfloor \frac{t}{2} \rfloor$, and there is no edge in H that connects $C(Top(v))$ and $C(Top(u))$, set $H \leftarrow H \cup \{(u, v)\}$.*

   (b) *Otherwise, assume, without loss of generality, $\lfloor \frac{t}{2} \rfloor \geq Height(u) \geq Height(v)$. Consider the collection of labels $L_v(u) = \{l^{k_1}, l^{k_2}, \ldots, l^{Height(u)}\} \subseteq L(u)$, where $k_1 \geq Height(v)$ and $k_1 < k_2 < \ldots < Height(u)$. Let $l = l^i \in L_v(u)$ such that l^i is marked as selected and there is no $l^j \in L_v(u)$ with $j < i$ that is marked as selected.*

      i. *If such a label l exists, label the vertex v with the successor $l' = Succ(l)$ of l, i.e., $L(v) \leftarrow L(v) \cup \{l'\}$. Incorporate the edge in the spanning tree $Tree(l')$. If l' is selected, label v with $l'' = Succ(l')$ and incorporate the edge in the tree $Tree(l'')$. Continue this until we see a label that is not marked as selected.*

      ii. *If no such label l exists and there is no edge (u', v) in $M(v)$ such that u, u' are labeled with the same label $l \in L_v(u)$, add (u, v) to $M(v)$, i.e., set $M(v) \leftarrow M(v) \cup \{(u, v)\}$.*

3. *After seeing all the edges in the stream, output the union of the spanning trees for all the labels, $M(v)$ for all the vertices, and the set H as the spanner.*

Because i can take $O(t)$ values, $|M(v)| = \sum_i |M^{(i)}(v)|$ is $O(t \cdot n^{1/t} \log n)$ with high probability. □

Alternatively, we can prove Lemma 4.9 in the following way. we can follow a vertex v while $Height(v)$ increases monotonically to $\lfloor \frac{t}{2} \rfloor$. Assume that v has d neighbors, and the neighbors are revealed in the stream as $u_1, u_2, \ldots, u_d$. We examine how the edges $(v, u_1), (v, u_2), \ldots, (v, u_d)$ are considered for the set $M(v)$ by our algorithm.

An edge is considered for the set $M(v)$ when $Height(v) \leq Height(u_i)$. (If $Height(v) > Height(u_i)$, we consider the edge for the set $M(u_i)$.) Therefore, to analyze $M(v)$, we can focus on the neighbors that, at the time when they are revealed, are at heights ($Height(u_i)$) higher than that of the vertex v. Let $S = u_{i_1}, u_{i_2}, \ldots, u_{i_s}$ be the sequence of these neighbors.

First, if u_i and v belong to the same cluster $C(l)$ for some l, due to the condition for step 2 of the algorithm, the edge (v, u_i) is discarded. Second, a neighbor in S may introduce an edge to $M(v)$ only when its label is "new" to the current $M(v)$, *i.e.*, the label is not in the set of labels used on the vertices in the set $M(v)$ at that time. Let S' be the longest subsequence of S with the property that when $u_i \in S'$ is revealed, $L(v) \cap L(u_i) = \emptyset$ and all the labels in $L(u)$ are also "new" to the current $M(v)$.

Due to step 2(b)(i), each time we see a neighbor in S' whose label is marked as "selected," v gets at least one new label and the value $Height(v)$ increases by at least one. Now consider the neighbors in S' after $Height(v)$ reaches $\lfloor \frac{t}{2} \rfloor$. Let u be such a neighbor. Note that $Height(u) \geq Height(v)$. Therefore, $Height(u) = \lfloor \frac{t}{2} \rfloor$. If the edge (v, u) needs to be included, it is included in the set H due to step 2(a).

Hence, as far as $M(v)$ is concerned, the sequence S' stops once there are $\lfloor \frac{t}{2} \rfloor$ neighbors in the sequence, whose label set $L_v(\cdot)$ contains "selected" labels. The

size of $M(v)$ is at most the length of the shortest prefix of S', which contains $\lfloor \frac{t}{2} \rfloor$ such neighbors. In other words, we have a sequence of "0"s (unselected) and "1"s (selected). Each element of the sequence takes the value "1" with probability $\frac{1}{n^{1/t}}$ and "0" with the probability $1 - \frac{1}{n^{1/t}}$. The length of the shortest prefix containing $\lfloor \frac{t}{2} \rfloor$ "1"s upper bounds the size of $M(v)$. We consider how many "0"s there can be in between the i-th and the $(i+1)$-th "1." This number follows a geometric distribution. We give an upper bound on this number that holds with high probability for all the values of i up to $\lfloor \frac{t}{2} \rfloor$. We then multiply this number by $\lfloor \frac{t}{2} \rfloor$ to give an upper bound on the size of $M(v)$.

Now we analyze the number of edges stored by our algorithm.

Lemma 4.10. *The algorithm stores $O(t \cdot n^{1+1/t} \log n)$ edges, with high probability.*

Proof. The algorithm stores edges in the set H, in the spanning trees for each cluster $C(l)$, and in the sets $M(v)$, $\forall v \in V$. By the Chernoff bound and the union bound, with high probability, the number of clusters at level $t/2$ is $O(\sqrt{n})$ and the size of the set H is $O(n)$.

For each label l, the algorithm stores a spanning tree for the set of vertices $C(l)$. Note that for $i = 1, 2, \ldots, \lfloor \frac{t}{2} \rfloor$, a vertex is labeled with at most one label in L^i. Hence, $\bigcup_{l \in L^i} C(l) \subseteq V$. Thus, the overall number of edges in the BFS trees is $O(t \cdot n)$.

By Lemma 4.9, with high probability, $|M(v)| = O(t \cdot n^{1/t} \log n)$. By the union bound, with high probability $\sum_{v \in V} |M(v)| = O(t \cdot n^{1+1/t} \log n)$. $\square$

Theorem 4.7. *For an unweighted, undirected graph of n vertices, presented as a stream of edges, and a positive number t, there is a randomized streaming algorithm that constructs a $(2t+1)$-spanner of the graph in one pass. With high probability, the algorithm uses $O(t \cdot n^{1+1/t} \log^2 n)$ bits of space and processes each edge in the stream in $O(t^2 \cdot n^{1/t} \log n)$ time.*

Proof. Consider Algorithm 4.4. At the beginning of the algorithm, for all the labels $l \in L^0$, $C(l)$ is a singleton set and the depth of the rooted spanning tree for $C(l)$ is zero. We now bound, for label l^i, where $i > 0$, the depth of the rooted spanning tree T^i on the vertices in $C(l^i)$. A tree grows when an edge (u, v) is incorporated into the tree in step 2(b)(i). Note that in this case, l^i is a successor of some label l^{i-1} of level $i - 1$. Assume that the depth $d_{T^i}(v)$ of the vertex v in the tree is one more than the depth $d_{T^i}(u)$ of the vertex u. Then $u \in C(l^{i-1})$, and the depth $d_{T^{i-1}}(u)$ of u in the rooted spanning tree T^{i-1} of $C(l^{i-1})$ is the same as $d_{T^i}(u)$. Hence, $d_{T^i}(v) = d_{T^{i-1}}(u) + 1$ where T^i is a tree of level i and T^{i-1} is a tree of level $i - 1$. Given that $d_{T^0}(x) = 0$ for all $x \in V$, the depth of the rooted spanning tree for $C(l)$, where l is a label of level i, is at most i.

We proceed to show that for any edge that the algorithm does not store, there is a path of length at most $2t + 1$ that connects the two endpoints of the edge. The algorithm ignores three types of edges. Firstly, if $L(u) \cap L(v) \neq \emptyset$, the edge (u, v) is ignored. In this case, let l be one of the label(s) in $L(u) \cap L(v)$; u and v are both on the spanning tree for $C(l)$. Hence, there is a path of length at most t connecting u and v. Secondly, (u, v) will be ignored if $Height(v) = Height(u) = \lfloor \frac{t}{2} \rfloor$ and there is already an edge connecting $C(Top(u))$ and $C(Top(v))$. In this case, the path connecting u and v has a length of at most $2t+1$. Finally, in step 2(b)(ii), (u, v) will be ignored if there is already another edge in $M(v)$ that connects v to some $u' \in C(l)$ where $l \in L(u)$. Note that u and u' are both on the spanning tree of $C(l)$. Hence, there is a path of length at most $t + 1$ connecting u and v.

Hence, the stretch factor of the spanner constructed by Algorithm 4.4 is $2t + 1$. By Lemma 4.10, with high probability, the algorithm stores $O(t \cdot n^{1+1/t} \log n)$ edges and requires $O(t \cdot n^{1+1/t} \log^2 n)$ bits of space. Also note that the bottleneck in the processing of each edge lies on step 2(b)(ii), where for each label in $L_v(u)$, we need

to examine the whole set of $M(v)$. This takes $O(t^2 \cdot n^{1/t} \log n)$ time. $\square$

Once the spanner is built, all-pairs, shortest distances of the graph can be computed from the spanner. The computation does not need to access the input stream and thus can be viewed as post-processing. Although Algorithm 4.4 constructs spanners for an unweighted, undirected graph, we can use it, along with the geometric grouping technique introduced in Section 4.2, to construct spanners for weighted, undirected graphs.

Theorem 4.8. *For a weighted, undirected graph of n vertices, whose maximum edge weight is ω_{max}, and whose minimum edge weight is ω_{min}, there is a randomized streaming algorithm that constructs a $((1+\epsilon)\cdot(2t+1))$-spanner of the graph in one pass. With high probability, the algorithm uses $O(\log_{1+\epsilon}\frac{\omega_{max}}{\omega_{min}} \cdot t \cdot n^{1+1/t}\log^2 n)$ bits of space and processes each edge in the stream in $O(t^2 \cdot n^{1/t}\log n)$ time.*

In the case where $t = \frac{\log n}{\log\log n}$, Algorithm 4.4 computes a $(2\frac{\log n}{\log\log n}+1)$-spanner in one pass. It uses $O(n\log^4 n)$ bits of space and processes each edge in $\log^4 n$ time. Once we have the spanner, the all-pairs, shortest-path distances, as well as the diameter of the graph, can be approximated. Also, it indirectly $\frac{2t+2}{3}$ approximates the girth of the graph. This is so because if the original graph has a girth larger or equal to $2t+2$, the spanner will contain all the edges of the graph. *i.e.*, the spanner is the graph. Therefore, the girth can be computed exactly. On the other hand, if the constructed spanner is a strict subgraph of G, we know the girth of G must have been between 3 and $2t+2$.

Finally, we cite the following lower bound from [38]:

Theorem 4.9. [38] *In one pass and using $o(n^{1+\gamma})$ bits of space, it is impossible to approximate the distance between two nodes $s, t \in V$ better than by a $\frac{1-\epsilon}{\gamma}$ ratio.*

Furthermore, this lower bound holds even if we are promised that $d_G(s,t)$ *is equal to the diameter of* G.

Given this lower bound, our algorithm is not very far from the optimal.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Streaming is an important model for processing massive data sets. Current research on streaming computation has considered only a few types of problems. To fully understand the computational power of the streaming model, it is necessary to explore more problems. In this dissertation, we have investigated computational-geometry problems and graph-theoretic problems in the streaming model. We showed that, for many geometry and graph problems, because of the space limit and the sequential-access requirement of the streaming model, it is impossible to compute exact solutions. We then turned our attention to approximate computations, where inaccurate results with a bounded error can be accepted. We devised algorithms that approximate diameters of sets of points in the streaming and sliding-window models. We also devised algorithms that give approximations to the shortest-path distances in a graph. In both cases, we also gave lower bounds on these problems. The lower bounds state that, in terms of space, our algorithms are close to optimal.

For geometry and graph problems investigated in this dissertation, our results

show that there are small-space representations (sketches) of the massive data sets that can be used to compute approximate results for the problems. Although similar representations (sketches) may have been used in algorithm designs in the traditional model, our results provide, for the first time, streaming constructions for these representations. Clearly, due to space restrictions, a sketch is not a lossless compression of the original data set. Therefore, a particular sketch may only be good for computing a particular function. We believe that, for streaming computations, it is important to understand which sketches can be constructed for a given problem. We also consider the following two types of open problems to be important in the streaming model.

5.2 Open Problems

In this dissertation, we considered computational-geometry problems in low (constant) dimensions. Geometry problems in high dimensions are also important. In particular, there are many problems, such as those in information retrieval, that are modeled as high-dimensional computational-geometry problems. A streaming algorithm presented in [56] c-approximates the diameter (for $c > \sqrt{2}$) in a d-dimensional space using $O(dn^{1/c^2-1})$ space. However, most high-dimensional geometric problems remain open in the streaming model. It has been shown in [46] that, for a set of points, there exists a small subset called a "coreset." One can use the coreset to compute an approximate clustering for the original input set. It can also be used in solving (approximately) problems such as MaxCut, maximum matching. It would be interesting to further explore coresets for other geometric problems in high dimensions.

The second type of open problems are dynamic problems. In our introduction to massive data streams in Section 2.1.1, we briefly mentioned models of streaming such

as the turnstile model. Problems in this model are dynamic problems, *i.e.*, there is an implicit set of data. The data elements in the stream, however, indicate changes (additions and deletions) to the implicit set. A dynamic problem seeks to compute certain functions on the current set, *i.e.*, the implicit set after the modifications indicated by the stream seen so far. The streaming problems considered in this dissertation (except the problems in the sliding-window model) can be viewed as partially dynamic problems. That is, the stream only adds data elements to the implicit set but never removes elements. It would be interesting to investigate the corresponding full dynamic problems.

5.3 Future Work: From Theory to Applications

We have focused on a theoretical approach to understanding the streaming computational model. We consider fundamental problems in computational geometry and graph theory and study algorithms and complexity for these problems. On the other hand, it is also important to explore real systems and applications where streaming computation is an essential part of the system. We end this dissertation by describing such a system in our ongoing work. This work aims to bring theoretical work into applications by applying streaming computations in networking monitoring systems. A preliminary report on this work will appear in [81].

The stability of BGP affects the stability, availability, and efficiency of the Internet. It is thus of great importance to understand the behavior of BGP. In recent years, a lot of research has been done in analyzing BGP instability [17, 20, 42, 64, 78]. Some studies view any route change as a sign of instability. However, not all route changes are "bad," in that not all cause instability problems for the network. In some cases, route updates may be normal responses by BGP to network changes. In

other cases, they may be the result of traffic-engineering efforts. These route changes actually can help to improve the performance of the network. It is only "abnormal" route changes (*e.g.*, frequent updates due to flaky equipment, protocol oscillation, route hijacking, etc.) that require the network operator's attention.

Of course, the notion of "abnormal" route changes (or "abnormal" BGP updates) is not well defined. One AS's "abnormal" BGP updates may be another AS's "normal" updates. The definition depends on different AS's configurations and policies. Although a general definition may be difficult to obtain, it is helpful to consider features/attributes of BGP updates that may be used to identify "anomalies." Ideally one would also like to automatically learn the "normal" behaviors and use what has been learned to distinguish the "abnormal" from the "normal."

Previously, statistics-based anomaly detection [74, 82] has been studied towards this goal. A statistics-based detection system [58] detects anomaly by comparing the current behavior with the history of known behaviors. If the current behavior deviates radically from the known history, it will be flagged as a possible anomaly. In the statistics-based detection system, the behaviors of BGP updates are normally measured and represented by simple aggregates such as the number of updates within a certain period of time or the time it takes for a prefix to converge. The history is made of a simple statistic (the mean and the variance) of these aggregates, and traditional statistical tests can be used to see whether the current behavior deviates significantly from the normal behavior.

The statistics-based approach is simple and thus may be easily deployed and run with high efficiency. This makes it attractive in situations where huge amounts of data need to be processed. However, the simplicity in its representation also makes it unable to capture complex features that may be important for a better analysis of BGP behavior. Furthermore, the representations in many such systems have the

"magic number" problem. That is, they use parameters, set either arbitrarily or according to statistics, for controlling granuality of the analysis, or for the threshold that determines when a burst of messages ends. For example, a prefix may be determined to have converged to a stable route if there have been no updates for that prefix for T minutes. Clearly, it is hard to set a good value for such a parameter T.

These problems motivate our search for new representations and new frameworks that are independent of "magic numbers" and more powerful in characterizing BGP updates. At the same time, the framework should still be efficient and deployable. Towards this direction, we propose a framework using an instance-based learning. An instance-based learning is a nearest-neighbor learning where the label (say "normal" or "abnormal") of a behavior is determined by the label of the most similar behavior in the knowledge base. Hence, at a high level, our learning-based anomaly detection bears a resemblance to the statistics-based anomaly detection. Both approaches detect abnormal behavior by comparing it to a history of known behaviors. At the detailed level, however, the two approaches differ greatly in how the behaviors are represented, how the history is compiled, and how the current behavior is tested against the history.

In our framework, BGP update behaviors are represented by a vector of quantified features. Such a representation maps a particular behavior to a point in a multidimensional vector space. The set of normal behaviors maps to a set of points whose neighborhood defines the domain of "good" behaviors. A behavior represented by an outlier point that is far away from this domain would be suspicious and may require the network operator's attention. The domain of normal behaviors and the outlier points can be discovered by clustering. Thus, in our framework, the clusters of known behaviors form a knowledge base. To check the current behavior against

the history is to perform a nearest-neighbor query in the knowledge base. By extending the representation to a vector of quantified features, our framework is much more expressive in its ability to characterize BGP-update behaviors.

In a preliminary study, we use the features of BGP-update dynamics to make the representation vectors. That is, we look for features in temporal patterns of the BGP-update dynamics, such as burst duration, inter-burst intervals, etc. We argue that features in BGP-update dynamics are relevant to the detection of BGP-update anomalies. As discovered in [63], different update types have different convergence time and number of update messages. The convergence and update-message numbers also differ from ISP to ISP. Hence, the configuration of the underlying network affects the way the new paths are explored, which in turn can lead to different timing and message numbers in different systems. In fact, the work in [29] used BGP-update dynamics to infer the network's topological information. If we assume that the underlying network does not change configuration often, the temporal dynamics of the updates should also be similar. Thus, unusual dynamics may indicate anomalous updates, as the analysis of BGP updates during certain worm attacks have shown [65]. Indeed, several recent works [82, 79] have examined the BGP-update dynamics for signs of anomaly.

To extract the features (temporal patterns) of BGP-update dynamics, we use wavelet transformations. In signal processing, wavelet transformations map a raw signal (a function of time) into a signal (or coefficients) of the time-frequency domain (a function of both time and frequency). Note that if we treat a sequence of update messages as a signal along time, *i.e.*, a function $f(t)$ whose value is the number of updates (for a view or for a particular prefix) at time t, an update-message burst within this signal can be viewed as a high-frequency signal (the individual updates) modulated by a low-frequency signal (the burst). The wavelet transformation can

thus reveal the temporal structures in the update-message dynamics. Let $\Psi(\frac{x-\tau}{\delta})$ be the wavelet with translation τ and scale δ. The discretized transformation is defined to be $\gamma(\delta, \tau) = \sum_x S(x) \cdot \frac{1}{\sqrt{\delta}} \Psi^*(\frac{x-\tau}{\delta})$. The multiscale analysis employs a set of values for τ and δ. The function value $\gamma(\delta, \tau)$ then defines a surface, and we are interested in the peak values, *i.e.*, local maximum, of $\gamma(\delta, \tau)$. This is because a burst of length t will give a large value of $\gamma(\delta, \tau)$ at the scale δ closest to t and at the time τ when the burst happens. Our representation consists of several histograms. We construct one histogram for the peak values of $\gamma(\delta, \tau)$ for each value of δ. To include time information about the bursts in our representation, we also use a histogram for the time intervals between each pair of consecutive peaks. Note that these two types of histograms are shift-invariant. Therefore, so is our representation.

Our representation has several advantages. First, it avoids the "magic number"' problem by employing a multiscale transformation. Using a set of different frequencies, the wavelet transformation provides a systematic, multigranuality view of the structures and patterns of BGP updates. Note that we view the update dynamics at different levels: 1 second, 2 seconds, $\ldots, \ell_{max}$ second. Ideally, when ℓ_{max} goes to infinitely large, we will have a comprehensive view of the dynamics. However, such a system also requires a huge amount of computing resources. An implementation must decide on a realistic value for ℓ_{max}. Therefore, our system avoids the "magic number" problem not by magically getting rid of all the parameters, but by systematically examining the interval in which the possible values of the parameters can be taken.

Another advantage of our representation is its shift-invariant property. Our anomaly-detection framework compares the current BGP-update dynamics with the dynamics in the knowledge base. Two sequences of update dynamics may be very alike, but they may not be well aligned in time. For example, if the normal behav-

ior of a particular prefix is to undergo a short burst of updates once in a while, it shouldn't matter if the updates happen at 12:00 or 12:30. However, a shift-intolerant representation would view the two sequences of dynamics as quite different behaviors and misclassify them. Furthermore, update dynamics of different prefixes are normally not well aligned by time. Even the updates of multiple prefixes caused by a single event may shift in time because of rate limiting by protocol timers or by the transmission protocol. Again, a shift-intolerant representation will characterize them as very different behaviors. Classification according to such a representation will not produce meaningful categories of BGP-update dynamics. Because our representation is shift-invariant, it provides a metric space in which comparison and classification don't have these problems.

However, the above representation construction does not tell us what kind of feature indicates anomalies in the BGP update. This is done by the learning component in our framework. We use clustering to discover the structures in the BGP update dynamics. Clustering groups the points representing the dynamics into categories, which can then be examined and labeled by network operators. Therefore, clustering helps to learn the positions in our representation space that correspond to anomalies in BGP updates. These positions can be used later in identifying anomalies in future BGP updates.

We have experimented with a preliminary implementation of our framework, investigating daily BGP update behaviors for six months. We cluster the update dynamics of a single prefix as well as the update dynamics across prefixes over a view. Focusing on each prefix in isolation, we show that, for most prefixes, update dynamics are similar from day to day. Furthermore, on a single day, most prefixes also display similar dynamics. Only a few prefixes exhibit behaviors that are quite different from the majority. The small set of prefixes or daily behaviors can be

further examined for anomaly detection. In particular, we observe that most prefixes whose update dynamics deviate from the majority are unstable prefixes with frequent routing changes.

This preliminary implementation is far from a useful application that can be deployed in a real network system. Such an application is our future work. Streaming computation will play a central role in this application. The BGP updates form a stream of messages. The construction of representations requires streaming computation of convolution, and the learning component requires streaming clustering. We believe that such a system not only will provide a place for applications of theoretical work in streaming computation but also will serve as an experiment from which new, important streaming problems can be discovered.

Bibliography

[1] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[2] P. K. Agarwal and S. Har-Peled. Maintaining the approximate extent measures of moving points. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 148–157, 2001.

[3] P. K. Agarwal, J. Matousek, and S. Suri. Farthest neighbors, maximum spanning trees and related problems in higher dimensions. *Computational Geometry: Theory and Applications*, 1(4):189–201, 1992.

[4] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28(4):1167–1181, 1999.

[5] N. Alon, S. Hoory, and N. Linial. The moore bound for irregular graphs. *Graphs and Combinatorics*, 18(1):53–57, 2002.

[6] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[7] I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory, LNCS 447*, pages 26–37, 1990.

[8] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[9] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Saks. Adapting to asynchronous dynamic networks. In *Proc. 24th ACM Symposium on Theory of Computing*, pages 557–570, 1992.

[10] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pages 514–522, 1990.

[11] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

[12] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 82–91, 1999.

[13] S. Baswana and S. Sen. A simple linear time algorithm for computing a $(2k-1)$-spanner of $o(n^{1+1/k})$ size in weighted graphs. In *Proc. 30th International Colloq. on Automata, Languages and Computing*, pages 284–296, 2003.

[14] B. Bollobás. *Extremal Graph Theory.* Academic Press, New York, 1978.

[15] A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer System Science*, 60:630–659, 2000.

[16] A. Buchsbaum, R. Giancarlo, and J. Westbrook. On finding common neighborhoods in massive graphs. *Theoretical Computer Science*, 299(1-3):707–718, 2003.

[17] M. Caesar, L. Subramanian, and R. H. Katz. Towards localizing root causes of bgp dynamics. *Technical Report, CSD-3-1292, UC Berkeley*, 2003.

[18] T. Chan. Approximating the diameter, width, smallest enclosing cylinder and minimum-width annulus. In *Proc. 16th ACM Symposium on Computational Geometry*, pages 300–309, 2000.

[19] T. M. Chan and B. S. Sadjad. Geometric optimization problems over sliding windows. In *Proc. 15th International Symposium on Algorithm and Computation*, pages 246–258, 2004.

[20] D.-F. Chang, R. Govindan, and J. Heidemann. The temporal and topological characteristics of BGP path changes. In *Proc. 11th IEEE International Conference on Network Protocols*, pages 190–199, 2003.

[21] M. Charikar, L. O'Callaghan, and R. Panigrahy. Better streaming algorithms for clustering problems. In *Proc. 35th ACM Symposium on Theory of Computing*, pages 30–39, 2003.

[22] Cisco. Netflow, 1998.

[23] K. Clarkson and P. Shor. Applications of random sampling in computational geometry II. *Discrete Computational Geometry*, 4:387–421, 1989.

[24] E. Cohen. Fast algorithms for t-spanners and stretch-t paths. In *Proc. 34th IEEE Symposium on Foundation of Computer Science*, pages 648–658, 1993.

[25] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1998.

[26] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17:128–142, 1988.

[27] G. Cormode and S. Muthukrishnan. Radial histograms for spatial streams. *DIMACS Technical Report 2003-11*, 2003.

[28] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[29] G. A. David, N. Feamster, S. Bauer, and H. Balakrishnan. Topology inference from BGP routing dynamics. In *Proc. Internet Measurement Workshop*, 2002.

[30] P. Drineas and R. Kannan. Pass efficient algorithms for approximating large matrices. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 223–232, 2003.

[31] R. D. Dutton and R. C. Brigham. Edges in graphs with large girth. *Graphs and Combinatorics*, 7(4):315–321, 1991.

[32] M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 53–62, 2001.

[33] M. Elkin. A fast distributed protocol for constructing the minimum spanning tree. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 352–361, 2004.

[34] M. Elkin and D. Peleg. $(1+\epsilon, \beta)$-spanner constructions for general graphs. In *Proc. 33rd ACM Symposium on Theory of Computing*, pages 173–182, 2001.

[35] M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\epsilon, \beta)$-spanners in the distributed and streaming models. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 160–168, 2004.

[36] D. Eppstein, Z. Galil, and G. Italiano. Dynamic graph algorithms. *CRC Handbook of Algorithms and Theory of Computation*, Chapter 8, CRC-Press 1998.

[37] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. In *Proc. 31st International Colloquium on Automata, Languages and Programming, LNCS 3142*, pages 531–543, 2004.

[38] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: The value of space. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, 2005.

[39] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate L^1 difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002.

[40] J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Yale University Technical Report, YALEU/DCS/TR-1245*, Dec. 2002. Also partially presented at the November 6-9, 2001, and March 24-26, 2003, meetings of the DIMACS working group on streaming data analysis.

[41] J. Feigenbaum, S. Kannan, and J. Zhang. Computing diameter in the streaming and sliding-window models. *Algorithmica*, 41(1):25–41, 2005.

[42] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. ACM SIGCOMM Conference*, pages 205–218, 2004.

[43] P. Flajolet and G. Martin. Probabilistic counting. In *Proc. 24th IEEE Symposium on Foundation of Computer Science*, pages 76–82, 1983.

[44] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[45] J. H. Fong and M. Strauss. An approximate L^p-difference algorithm for massive data streams. *Discrete Mathematics and Theoretical Computer Science*, 4(2):301–322, 2001.

[46] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. In *Proc. 37th ACM Symposium on Theory of Computing*, pages 209–217, 2005.

[47] P. Gibbons and Y. Matias. Synopsis data structures for massive data sets. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization*, A:39–70, 1999.

[48] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. 34th ACM Symposium on Theory of Computing*, pages 389–398, 2002.

[49] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. 27th International Conference on Very Large Data Bases*, pages 79–88, 2001.

[50] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. 33rd ACM Symposium on Theory of Computing*, pages 471–475, 2001.

[51] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[52] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *Technical Report 1998-001, DEC Systems Research Center*, 1998.

[53] J. Hershberger and S. Suri. Convex hulls and related problems in data streams. In *Proc. Workshop on Management and Processing of Data Streams*, 2003.

[54] J. Hopcroft and J. Ullman. Some results on tape-bounded turing machines. *Journal of the ACM*, 16:160–177, 1969.

[55] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 189–197, 2000.

[56] P. Indyk. Better algorithms for high-dimensional proximity problems via asymetric embeddings. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 539–545, 2003.

[57] P. Indyk. Algorithms for dynamic geometric problems over data streams. In *Proc. 36th ACM Symposium on Theory of Computing*, pages 373–380, 2004.

[58] H. Javitz and A. Valdes. The NIDES statistical components: Description and justification. *Technical report, SRI Network Information Center*, 1993.

[59] B. Kalyanasundaram and G. Schnitger. The probabilistic communication complexity of set intersection. *SIAM Journal on Discrete Mathematics*, 5:545–557, 1990.

[60] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *Proc. 28th International Conference on Very Large Data Bases*, pages 814–825, 2002.

[61] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the Web. In *Proc. 25th International Conference on Very Large Data Bases*, pages 639–650, 1999.

[62] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[63] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed internet routing convergence. *IEEE/ACM Transactions on Networking*, 9(3):293–306, 2001.

[64] M. Lad, A. Nanavati, D. Massey, and L. Zhang. An algorithmic approach to identifying link failures. In *Proc. 10th Pacific Rim International Symposium on Dependable Computing*, pages 25–34, 2004.

[65] M. Lad, X. Zhao, B. Zhang, D. Massey, and L. Zhang. Analysis of BGP update surge during slammer worm attach. In *Proc. 5th International Workshop on Distributed Computing, LNCS 2918*, pages 66–79, 2003.

[66] R. Lipton. Efficient checking of computations. In *Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science, LNCS 415*, pages 207–215, 1990.

[67] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[68] S. Muthukrishnan. Data streams: Algorithms and applications. 2003. Available at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[69] D. Peleg and A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[70] D. Peleg and J. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18:740–747, 1989.

[71] E. Ramos. Deterministic algorithms for 3-d diameter and some 2-d lower envelopes. In *Proc. 16th ACM Symposium on Computational Geometry*, pages 290–299, 2000.

[72] A. Shiryayev. *Probability*. Springer-Verlag, New York, 1995.

[73] A. Silberschatz, P. Galvin, and G. Gagne. *Applied operating system concepts*. John Wiley & Sons, New York, 2000.

[74] S. T. Teoh, K. Zhang, S.-M. Tseng, K.-L. Ma, and S. F. Wu. Combining visual and automated data mining for near-real-time anomaly detection and analysis in BGP. In *Proc. ACM CCS Workshop on Visualization and Data Mining for Computer Security*, 2004.

[75] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. 33rd ACM Symposium on Theory of Computing*, pages 183–192, 2001.

[76] J. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[77] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[78] J. Wu, Z. M. Mao, J. Rexford, and J. Wang. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *Proc. 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[79] K. Xu, J. Chandrashekar, and Z.-L. Zhang. A first step toward understanding inter-domain routing dynamics. In *Proc. ACM SIGCOMM Workshop on Mining Network Data*, 2005.

[80] A. Yao. Some complexity questions related to distributive computing. In *Proc. 11th ACM Symposium on Theory of Computing*, pages 209–213, 1979.

[81] J. Zhang, J. Rexford, and J. Feigenbaum. Learning-based anomoly detection in BGP updates. In *Proc. ACM SIGCOMM Workshop on Mining Network Data*, 2005.

[82] K. Zhang, A. Yen, X. Zhao, D. Massey, S. F. Wu, and L. Zhang. On detection of anomalous routing dynamics in BGP. In *Proc. 3rd International IFIP-TC6 Networking Conference, LNCS 3042*, pages 259–270, 2004.

[83] Y. Zhu and D. Shasha. Efficient elastic burst detection in data streams. In *Proc. 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 336–345, 2003.