Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XX Ciclo – 2007

Streaming Algorithms for Graph Problems

Andrea Ribichini

Università degli Studi di Roma "La Sapienza"

Dottorato di Ricerca in Ingegneria Informatica

XX Ciclo - 2007

Andrea Ribichini

Streaming Algorithms for Graph Problems

Author's address:

Andrea Ribichini
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Ariosto 25, 00185 Roma, Italy
e-mail: `ribichini@dis.uniroma1.it`
www: `http://www.dis.uniroma1.it/∼ribichini`

Abstract

This thesis studies algorithms for graph problems in a data streaming setting. In its classical form, the *Streaming* computational model assumes that the input data are presented as a sequential read-only stream, that has to be processed in one or more passes, using a memory that is small compared to the length of the stream. Despite its heavy restrictions, major success has been achieved in this model for many data sketching and statistics problems. Several graph problems have recently been solved in one or few passes, with a working memory of size $O(n \cdot \text{polylog}\, n)$, where n is the number of nodes of the input graph (in other words, there is enough memory to store the vertices, but not the edges of the graph). A natural question, often posed in the literature, is whether the space usage can be reduced, at the expenses of increasing the number of passes. And yet, somewhat surprisingly, no algorithm with both sublinear space and passes is known for natural graph problems in read-only streaming.

In the first part of this thesis, we show that space/passes tradeoff results are possible in read-only streaming, for some specialized variants of more general problems, such as *chain reachability* (a restriction of the general *reachability* problem to directed chain graphs), or even for a fundamental problem such as *undirected s-t connectivity*, for which both sublinear space and number of passes can be achieved, at least for sparse graphs.

Motivated by technological factors of modern storage systems, and in pursuit of space/passes tradeoff results, some authors have recently started to investigate the computational power of less restrictive models, where writing streams is also allowed. In one of these models, i.e., the *W-Stream* model, at each pass one input stream is read and one output (intermediate) stream is written; streams are pipelined in such a way that the output stream produced at pass i is used as input stream at pass $i + 1$.

We prove that space/passes tradeoffs are possible in a read-write streaming model for several fundamental graph problems, by devising efficient *W-Stream* algorithms, that allow a smooth tradeoff between the number of passes and the amount of memory they require. In particular, for any space restriction of s bits, we show upper bounds of the form $p = O((n \log n)/s)$ (where p denotes the number of passes required by the algorithm) for problems such as *connected components* and *minimum spanning tree*. We also prove, through communication complexity-based techniques, these algorithms to be optimal up to a logarithmic factor. For *single-source shortest paths* on directed graphs with small positive edge weights, we provide an upper bound of the form $p = O((n \log^{3/2} n)/\sqrt{s})$. The result can be generalized to deal with multiple sources within the same bounds. This is the first known streaming algorithm for shortest paths in directed graphs.

In the second part of the thesis, rather than focusing on problem specific approaches, we turn our attention to more general results, by developing reduction techniques that allow turning parallel algorithms into efficient *W-Stream* tradeoff ones. We provide upper bounds of the form $p = O((n \cdot \mathrm{polylog}\, n)/s)$ for several classical combinatorial problems, including *sorting*, *biconnected components* and *maximal independent set*. We also prove these algorithms to be optimal up to a polylogarithmic factor.

Despite the wealth of theoretical results, surprisingly little experimental work has been done in the field of streaming algorithms for graph problems. We contribute to bridge this gap in the third part of this thesis, by investigating, from an experimental point of view, the performances of three very recent one-pass read-only streaming algorithms for constructing graph spanners. We have put the algorithms to the test on a number of graph classes (power law, clustered, random), measuring both the quality of the produced spanners, in terms of their size and stretch factors, and their space/time requirements. Our study provides evidence that all three algorithms we have examined are likely to be very efficient in practice, as they find spanners of size and stretch much smaller than the theoretical bounds, and are competitive with off-line algorithms, while enjoying a much wider range of applicability, due to their greatly reduced memory requirements.

Contents

Chapter 1

Introduction

The typical data size of a wide range of applications in computational sciences can easily reach the order of Terabytes or even Petabytes. In all such applications managing massive data sets, using secondary and tertiary storage devices is a practical and economical way to store and move data: such large and slow external memories, however, are best optimized for sequential access, and thus naturally produce huge streams of data that need to be processed in a small number of sequential passes. Typical examples include data access to database systems [62] and analysis of Internet archives stored on tape [72]. Information naturally occurs in the form of huge data streams also in applications that monitor in real-time network traffic, on-line auctions, transaction logs such as Web usage logs, telephone call records or automated bank machine operations [59, 62, 100]. Among the computational models that have been proposed to deal with massive data sets, data stream processing has therefore received an ever increasing attention in the last few years.

In the *classical streaming* model [72, 87, 88], input data can be accessed sequentially in the form of a data stream, and need to be processed using a working memory that is small compared to the length of the stream. The main parameters of the model are the number p of sequential passes over the data and the size s of the working memory (in bits). A typical additional parameter is the per-item processing time, which should also be kept small. Despite the heavy restrictions of the *classical streaming* model, major success has been achieved for several data sketching and statistics problems, where $O(1)$ passes and polylogarithmic working space have been proven enough to find approximate solutions (see, e.g., [7, 55, 58] and the bibliographies in [14, 88]). On the other hand, many other problems seem far from being solved within similar bounds, including most classical graph problems. Relevant examples are *graph connectivity* and *shortest paths*, for which linear (in the number of vertices) lower bounds on $p \times s$ are known [72]. Some recent papers show that several graph problems can be solved with one or few passes in the *Semi-streaming* model [53, 54, 85] where the working memory size is $O(n \cdot \text{polylog } n)$ for an input graph with n vertices: in other words, akin to semi-external memory models [2, 103] there is enough space to store the vertices, but not the edges of the input graph. While $O(n \cdot \text{polylog } n)$ space seems to be a "sweet spot" for streaming graph problems [88], a natural question already posed in [72, 87] is whether we can reduce the space usage at the price of increasing the number of passes. Surprisingly, to the best of our knowledge no algorithms with both sublinear space and passes are known for natural graph problems in *classical streaming*. Finding effective space-passes tradeoffs in this context appears therefore to be a challenging research direction for both its theoretical and practical implications. Consider for instance the problem of processing a very large graph stored on a file in secondary memory. On a standard computing platform with 1 GB of available main memory, a tradeoff algorithm that runs in $p = (n \log n)/s$ passes[1] can process a graph with 4 billion vertices and 6 billion edges stored in a 50 GB file in less than 16 passes. Using a RAID disk with 100 MB/sec sequential access rate, this would take roughly 2.5 hours. With a streaming algorithm that requires $s \geq n \log n$ bits without being able to trade space for passes, we would simply not be able to solve the problem at hand (even with infinite time) unless 16 GB of main memory are available.

Motivated by technological factors, some authors have recently started to investigate the computational power of less restrictive streaming models. Today's computing platforms are equipped with large and inexpensive disks highly optimized for sequential read/write access to data. These considerations have led Ruhl to introduce the *W-Stream* model [97], which extends *classical streaming* with the ability to write intermediate streams. In this model, one pass, while reading data from the input stream and processing them in the working memory, produces items that are sequentially appended to an output (intermediate) stream, which will be used as input stream in the next pass (and so on). Among the primitives that can efficiently access data in a non-local fashion, sorting is perhaps the most optimized and well understood. This further consideration has induced Aggarwal *et al.* [3] to propose the "streaming and sorting" model (also known as *StreamSort*). This model extends *W-Stream* with the ability to reorder the intermediate streams at each pass for free. A *StreamSort* algorithm, therefore, alternates two kinds of passes: streaming passes which, analogously to *W-Stream*, sequentially read an input stream and sequentially append items to an output stream, and sorting passes which reorder the input stream according to some (global) partial order

[1] Throughout this dissertation, unless otherwise stated, we will assume that all logarithms are to the base of 2.

and produce the sorted stream as output. As before, streams are pipelined in such a way that the output stream produced during pass i is used as input stream at pass $(i + 1)$. As shown in [3], the combined use of intermediate temporary streams and of a sorting primitive yields enough power to solve efficiently (within polylogarithmic passes and polylogarithmic memory) a variety of problems, including *graph connectivity*, *minimum spanning tree*, and various geometrical problems. It remains however an open question whether problems such as *shortest paths* (and even *breadth first search*) can be solved efficiently in this more powerful model.

1.1 Related Work

The problem of computing a given function over a stream of data using a small memory already appears in papers from the late 1970s. Morris, for instance, addressed the problem of counting large numbers of events with limited storage [86]. The streaming model was first formally defined by Munro and Paterson in 1980, in their seminal work on pass-efficient selection (e.g., of the k–th highest element in a sequence) and sorting [87].

Starting from the early 1990s, a wide range of statistics and data sketching problems, under the severe restrictions imposed by the streaming model, have been the object of intensive study. Most notably, the computation of frequency moments, a field pioneered by the seminal works of Alon *et al.* [7], is now fully understood, with almost matching (up to polylogarithmic factors) upper bounds [23, 39, 76] and lower bounds [17, 31, 75, 98]. Histograms provide succinct representation of data, by dividing them into groups (or *buckets*), and maintaining suitable statistics for each bucket. The problem of computing histograms, and the closely related one of maintaining quantiles over a data set, have been addressed in [35, 58, 61, 68, 69, 89, 90]. Wavelet decomposition is another widely used tool to provide summarized representations of data; works on how to compute wavelet coefficients in the streaming model include [6, 58, 60, 91]. Problems related to norm estimation, e.g., dominance norms and L_p sums, have been addressed in [41, 74]. Geometric problems, and in particular *clustering* problems (i.e., given a set of points and a distance function defined on them, find a partition into clusters that optimizes a certain objective function), have been the subject of much recent research [33, 56, 57, 70, 73].

The above results have found applications in various domains, including sensor networks, transaction logs, network management and monitoring [59, 62, 72, 100], and database systems [14, 62]. In this last field, in particular, several authors have focused on the space-efficient evaluation of queries over data streams, through approximation techniques such as random (reservoir) sampling [34, 36, 104] and *sliding windows* (i.e., only the most recent data, that fall within a specified time interval, or *window*, are considered for query evaluation) [15, 43]. Data summarization techniques, such as histograms and wavelets, have also been employed in query planning, evaluation and optimization [11, 60, 84].

Despite the major successes achieved for data sketching and statistics problems (where $O(1)$ passes and polylogarithmic working space have been proven enough to find approximate solutions), however, many other problems remain far from being solved within similar bounds, including most classical graph problems. Relevant examples are *graph connectivity* and *shortest paths*, for which linear lower bounds on the $p \times s$ product (where p is the number of passes, and s the amount of working memory) are known [72]. Some recent papers show that several graph problems (including *connectivity*, *minimum spanning tree* and *matching*) can be solved with one or few passes in the *Semi-streaming* model [53, 54, 85] where the working memory size is $O(n \cdot \text{polylog } n)$ for an input graph with n vertices: in other words, akin to semi-external memory models [2, 103] there is enough space to store the vertices, but not the edges of the input graph. Other results, that require superlinear memory (in the number of vertices of the input graph), include *triangle counting* [18, 28, 78] and the construction of graph spanners [20, 48, 54]. A very natural question, in the context of streaming algorithms for graph problems, already posed in [72, 87], is whether it is possible to reduce the memory requirements, at the expenses of increasing the number of passes that an algorithm has to perform over the data stream. In spite of both its theoretical and practical importance, somewhat surprisingly, this question remains as yet unanswered in the literature.

Motivated by technological factors, some authors have recently started to investigate the computational power of less restrictive streaming models. These include the *W-Stream* model [97], which allows sequential reading and writing of intermediate streams, and the *StreamSort* model [3], which further extends the streaming model with the ability to reorder the intermediate streams at each pass for free. As shown in [3], the combined use of intermediate temporary streams, and of a sorting primitive, yields enough power to solve efficiently (within polylogarithmic passes and polylogarithmic memory) a variety of problems, including fundamental graph problems such as *graph connectivity* and *minimum spanning tree*.

Since random accesses in external memory are significantly slower than sequential passes, Grohe *et al.* have also proposed an abstract model that captures the essence of external memory and stream processing [66]. This model restricts the size of the main memory and the number of random accesses to external memory, but does not restrict sequential reads. Similarly to *StreamSort*, the model admits the usage of external memory for storing intermediate results. Lower bounds for sorting the input data and for other decision problems in this model have been proved in [65, 66, 67]. Very recently, these results have been extended to 2-sided error randomized algorithms [22].

1.2 Original Contributions of the Thesis

As pointed out in Section 1.1, algorithms are known in read-only streaming for many fundamental graph problems, such as *connected components*, *biconnected components*, *minimum spanning tree*, *matching*. For many of these problems lower bounds of the form $p \times s = \Omega(n)$ exist (where p is the number of passes, s the size of the working memory in bits and n the number of nodes of the input graph), and the corresponding algorithms are optimal up to polylogarithmic factors. However, while running in one or constant number of passes, they all require $\Theta(n \log n)$ bits of main memory. A very natural question, already posed in the literature [72, 87], is whether the memory usage can be reduced at the expense of increasing the number of passes. Despite its theoretical and practical relevance, the question of whether space/passes tradeoffs can be achieved, remains to this day largely unanswered.

As a first original contribution, in this thesis we will show that some tradeoffs between memory usage and number of passes are possible in read-only streaming, for simpler specialized variants of more general graph problems (such as *chain reachability*, which is the restriction of the general *reachability* problem on directed graphs to a directed chain), or even for natural problems, such as *undirected s-t connectivity*, where sublinear space and passes can be achieved, at least in the case of sparse graphs. Turning our attention to the more powerful *W-Stream* model, which allows reading/writing of intermediate streams (see Section 2.2.3 for a formal definition), we will also prove that smooth space/passes tradeoffs are possible in this model, for fundamental problems such as *connected components*, *minimum spanning tree* and *multiple-sources shortest paths*.

As a second goal of this thesis, rather than focusing on problem specific approaches, we will devote our attention to more general techniques, that will enable us to derive results valid for classes of combinatorial problems. As a second original contribution, in particular, we will propose general reduction techniques that allow turning parallel algorithms into tradeoff *W-Stream* ones. We will show that our techniques yield near-optimal algorithms for a variety of combinatorial problems, such as *maximal independent set*, *biconnected components* and *sorting*.

In spite of the abundance of theoretical results, very few experimental papers can be found in the literature concerning streaming algorithms for graph problems. We contribute to bridge this gap by presenting, as a third original contribution, an experimental study comparing *Semi-streaming* algorithms for building graph spanners (almost certainly, the most extensively studied graph problem in a streaming setting). Our experiments indicate that the streaming algorithms we have studied achieve results competitive with those of off-line algorithms, while having greatly reduced memory needs.

The rest of this section is devoted to a more detailed overview of our original contributions.

Trading Off Space for Passes.

As a first original contribution, we devise several algorithms in the *W-Stream* model, that provide effective space-passes tradeoffs for fundamental graph problems. In particular, we show that for any space restriction of s bits:

- *Connected components* and *minimum spanning tree* can be solved in *W-Stream* by deterministic algorithms in $O((n \log n)/s)$ passes. By adapting classical communication complexity arguments previously used in the *classical streaming model*, we can prove an $\Omega(n/s)$ lower bound on the number of passes for both problems in *W-Stream*. Our algorithms are thus optimal up to a logarithmic factor.

- *Single-source shortest paths* in directed graphs with positive integer edge weights up to C can be solved in *W-Stream* by a randomized algorithm in $O((C\, n \log^{3/2} n)/\sqrt{s})$ passes. The result can be generalized to deal with $\rho \cdot \sqrt{s/\log n}$ sources within the same bounds for any $\rho \in (0, 1)$. This is the first known algorithm for shortest paths on directed graphs in a streaming model. We remark that other results on distances in streaming models are based on the computation of graph spanners, and these yield approximate distances in undirected graphs only [20, 48, 54]. We also note that the lower bound for connectivity implies an $\Omega(n/s)$ lower bound on the number of passes also for *single-source* (and thus *multiple-sources*) *shortest paths*.

We remark that for these problems we have exactly the same lower bounds on $p \times s$ in both *classical streaming* and in *W-Stream*. The only known upper bounds in *classical streaming* assume $s = \Theta(n \log n)$. On the other hand, our *W-Stream* algorithms adapt to the available working memory, yielding a full range of possible space/passes tradeoffs. This motivates us to include some observations related to the computational power of *W-Stream*.

In particular, we show that there are problems impossible to solve in *Stream* for a given space restriction, that are instead solvable in *W-Stream* in a finite number of passes, and that there are problems that require an asymptotically smaller number of processed items in *W-Stream* than in *Stream*. On the other hand, we also show that many communication complexity-based lower bounds hold in *Stream* as well as *W-Stream*, and there are problems that maintain the same hardness.

We conclude by showing that space/passes tradeoffs are possible, even in *classical streaming*, for some specialized variants of fundamental graph problems (e.g., *chain reachability*), or even for a natural graph problems like *undirected s-t connectivity*, for which both sublinear space and number of passes can be achieved, at least for sparse graphs.

Most of these results have been published in the *Proceedings of the 17-th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)* [45], and in the in the *Proceedings of the 32-nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)* [44].

Reductions to Parallel Algorithms.

It is well known that algorithmic ideas developed in the context of parallel computational models have inspired the design of efficient algorithms in other models. For instance, Chiang *et al.* [38] showed that efficient external memory algorithms can be derived from PRAM algorithms using a general simulation. Aggarwal *et al.* [3] discussed how circuits with uniform linear width and polylog depth (**NC**) can be simulated efficiently in *StreamSort*, providing a systematic way of constructing algorithms in this model for problems in **NC** that use a linear number of processors. Examples of problems in this class include *undirected connectivity* and *maximal independent set.*

As an original contribution, we show how classical parallel algorithms designed in the PRAM model can be turned into near-optimal algorithms in *W-Stream* for several classical combinatorial problems. We first show that any PRAM algorithm that runs in time T using N processors and memory M can be simulated in *W-Stream* using $p = O((T \cdot N \cdot \log M)/s)$ passes. This yields near-optimal trade-off upper bounds of the form $p = O((n \cdot \text{polylog } n)/s)$ in *W-Stream* for several combinatorial problems, where n is the input size (most notably, *sorting*). For other problems, however, this simulation does not provide good upper bounds. One prominent example concerns graph problems, for which efficient PRAM algorithms typically require $O(m + n)$ processors on graphs with n vertices and m edges. For those problems, this simulation method yields $p = O((m \cdot \text{polylog } n)/s)$ bounds, while $p = \Omega(n/s)$ almost-tight lower bounds in *W-Stream* are known for many of them.

To overcome this problem, we study an intermediate parallel model, which we call RPRAM, derived from the PRAM model by relaxing the assumption that a processor can only access a constant number of cells at each round. This way, we get the PRAM algorithms closer to streaming algorithms, since a memory cell in the working memory can be processed against an arbitrary number of cells in the stream. For some problems, this enhancement allows us to substantially reduce the number of processors while maintaining the same number of rounds. We show that simulating RPRAM algorithms in *W-Stream* leads to near-optimal algorithms (up to polylogarithmic factors) for several fundamental problems, including *connected components*, *minimum spanning tree*, *biconnected components*, and *maximal independent set*. We remark that for some of these problems better (by a logarithmic factor) *ad hoc* algorithms designed directly in *W-Stream*, without using simulations, exist (e.g., for *connected components* and *minimum spanning tree*), while for others a logarithmic factor improvement can be obtained by maintaining the algorithm's overall structure, and implementing directly in *W-Stream* only certain operations (e.g., for *biconnected components*). For some problems, on the other hand, no better direct *W-Stream* implementation is known (e.g., *maximal independent set*).

Finally, we show that there exist problems for which the increased computational power of the RPRAM model does not help in reducing the number of processors required by a PRAM algorithm while maintaining the same time bounds, and thus cannot lead to better *W-Stream* algorithms. An example is deciding whether a directed graph contains a cycle of length two.

Most of the above results have been published in the *Proceedings of the 32-nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)* [44].

Engineering Streaming Algorithms.

In spite of the wealth of theoretical results, surprisingly little experimental work has been done in the field of streaming algorithms for graph problems (one notable exception is the paper by Buriol *et al.* on the problem of estimating the number of triangles in a graph [28]). As an original contribution, we contribute to bridge this gap, by presenting an experimental investigation aimed at comparing the performances of three recent *Semi-streaming* algorithms for computing graph spanners of unweighted graphs, in a single pass over the input data. We have considered two randomized algorithms, one by Baswana [20] and one by Elkin [48], for producing purely multiplicative $(2k-1)$-spanners, for arbitrary integer k, and a deterministic algorithm by Ausiello, Franciosa and Italiano [12] that produces a multiplicative-additive $(3, 2)$-spanner of the input graph. All three are natural choices since they have the best theoretical bounds and use simple data structures, and are therefore likely to be very efficient in practice. Our experimental analysis is intended to complement the theoretical results, by measuring the performances of the algorithms on typical instances, rather than focusing on worst-case scenarios only.

The algorithms have been coded in C using a uniform programming style and common data structures. Our experimental setup includes instances from different graph families (*power law*, *clustered*, *random*), of variable sizes and densities, and also an implementation of an internal memory algorithm by Zwick [105], as a benchmark against which to measure the performances of the streaming algorithms. A wide range of parameters has been taken into consideration: average and maximum stretch factors, stretch variance, size of the produced spanner, running time and peak memory usage.

We have come to the following conclusions:

- All three streaming algorithms we have considered are likely to be very efficient in practice, as they find spanners of size and stretch much smaller than the theoretical bounds, and are in fact competitive with off-line algorithms, while enjoying a much wider range of applicability due to their greatly reduced memory requirements, when compared to off-line algorithms.

- Setting large values of k in the algorithms by Baswana and by Elkin does not produce sparser spanners, in all the test sets we have considered. In fact, small values of k seem to be the case of interest in practice.

- The algorithm by Ausiello, Franciosa and Italiano, when compared to the other streaming algorithms, in addition to being deterministic and not requiring prior knowledge of the number of vertices in the input graph (which is instead required by both Baswana's and Elkin's randomization schemes), seems to have the additional benefit of producing spanners of better quality while using a comparable amount of space and time resources.

Most of the results described above have been published in the *Proceedings of the 15-th Annual European Symposium on Algorithms (ESA'07)* [12]. This paper has also been invited to a special issue of *Algorithmica*.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows:

- *Chapter 2* provides technical preliminaries (well known results from the literature, sometimes cast in a slightly different form than usual, to better suit our future needs) that will prove instrumental in deriving the original results of the thesis. The *data streaming model*, and several variants of it, are also formally defined in this chapter.

- *Chapter 3* presents efficient *W-Stream* algorithms for several fundamental graph problems (*connected components*, *minimum spanning tree* and *shortest paths*), that achieve a smooth tradeoff between the available memory and the number of passes over the data stream that they require. Some hardness and separation results between *Stream* and *W-Stream* are also presented. Finally, some first tradeoff results in *Stream* will be outlined.

- *Chapter 4* introduces general reduction techniques that allow turning parallel algorithms into efficient *W-Stream* tradeoff ones. Fundamental combinatorial problems such as *sorting*, *biconnected components* and *maximal independent set* are proved to be solvable optimally (up to a polylogarithmic factor) using these techniques.

- *Chapter 5* presents an experimental investigation, comparing the performances of three one-pass *Semi-streaming* algorithms for spanner construction on a variety of input graph classes. Performance measures such as output spanner size, average stretch factor, stretch variance, memory requirements and running time have been taken into consideration. The algorithms have also been put to the test against an off-line one for the same problem, as a benchmark measure.

- *Chapter 6* summarizes the results presented in this thesis, and outlines some open problems and future research directions.

Chapter 2

Preliminaries

2.1 Introduction

The objective of this chapter is to furnish technical preliminaries that will prove instrumental, throughout the thesis, for proving the original results it contains. Occasionally, we will reformulate some results, known in the literature, in a different form better suited to our needs. We will first formally define several variants of the data streaming computational model, as they have been devised over the years, coupled with a brief sketch of the most relevant results achieved in each. We will then present various algorithmic techniques that will be exploited in the following chapters for proving both upper and lower bounds.

The chapter is structured as follows. Section 2.2 will introduce several variants of the streaming model and briefly describe the most relevant results achieved in each, while Section 2.3 will present several algorithm design techniques that we will exploit in the following chapters. In particular Subsection 2.3.1 will introduce a communication complexity-based technique for proving lower bounds in streaming settings, Subsection 2.3.2 will briefly recall the graph theoretic result known as *long path property* and finally, Subsection 2.3.3 will present some techniques for adapting parallel algorithms to the *external memory* model and to a specific variant of the *classical streaming* model, known as the "streaming and sorting" (or *StreamSort*) model.

2.2 Models

This section introduces and motivates the various computational models for data stream processing that have been developed in the past few decades, together with an overview of the most significant results achieved in each of them.

2.2.1 Classical Streaming

The *classical streaming* model was defined implicitly in 1980, in the work of Munro and Paterson on pass efficient selection and sorting [87], and even earlier, in the context of systems with tape-based storage devices and small memory (compared to the capacity of the tape). In recent years, the ever increasing need to store massive data sets in large and slow secondary and tertiary storage devices, which are best optimized for sequential access, and thus naturally produce huge streams of data that need to be processed in a small number of sequential passes, has led to new interest towards this model [72, 88].

In *classical streaming*, input data can be accessed sequentially in the form of a data stream, and need to be processed using a working memory that is small compared to the length of the stream. The main parameters of the model are the number p of sequential passes over the data and the size s of the working memory (in bits). More formally, let Σ be some alphabet. We will call a sequence of symbols $S = x_1 x_2 \dots x_n$, with $x_i \in \Sigma$ for all $i \in \{1 \dots n\}$, a *stream.* Let M be a RAM machine with a local memory of s bits. A *streaming pass* is defined as the as the computation performed by M when accessing a stream S sequentially, i.e., when first reading x_1, then x_2, then x_3, and so forth. We will say that a problem is solvable in p streaming passes if the correct answer can be computed by M after p consecutive streaming passes (notice that the working memory of M is maintained between successive passes). This leads to the following definition:

Definition 2.1 *Let $p, s : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. Then* Stream*(p, s) denotes the class of problems solvable by a RAM machine using $s(n)$ bits of local memory and $p(n)$ streaming passes, for an input stream of length n. The state of the local memory is maintained between passes.*

With a slight abuse of notation, the following abbreviation will often be used:

$$\mathit{Stream}(O(f), O(g)) := \{\mathit{Stream}(p, s) \mid p = O(f), s = O(g)\}.$$

Please notice that similar notation shortcuts will also be used for the other complexity classes defined in this chapter.

The *classical streaming* model then results as the set of complexity classes for which $p(n), s(n) \ll n$, capturing the notion that the size of the input stream is large compared to the machine's memory and the number of passes allowed. Notice that our definition imposes no restrictions on the amount of computation performed by the RAM machine, as it is often the case when dealing with external memory models, where it is assumed that I/O operations take orders of magnitude longer than internal memory operations. There are, however, applications in which the *per-item processing time* (average, maximum) is a significant parameter that should be taken into account.

Despite the heavy restrictions of the *classical streaming* model, major success has been achieved for several data sketching and statistics problems, e.g., *approximate frequency moments* [7], *histogram maintenance* [58], L^1 *difference* [55], where $O(1)$ passes and polylogarithmic working space have proven enough to find approximate solutions (see also the bibliographies in [14, 88]). On the other hand, many other problems seem to be far from being solved within similar bounds, including many fundamental graph problems. Relevant examples are *graph connectivity* and *shortest paths*, for which linear lower bounds on the $p \times s$ product are known (see Section 2.3.1 and [72]).

2.2.2 Semi-streaming

Since the restrictions imposed by *classical streaming* have been proven too severe too allow efficient solutions to fundamental graph problems such as *connectivity* and *shortest paths* (see Section 2.3.1 and [72]), in the last few years several authors have looked towards relaxing some of these restrictions. In particular, some recent papers show that several graph problems can be solved with one or few passes in the *Semi-streaming* model [53] where the working memory size is $O(n \cdot \text{polylog } n)$ for an input graph with n vertices (or even $O(n^{1+\epsilon})$, with $\epsilon < 1$, for applications like *spanners*, for which linear memory in the number of vertices is provably not sufficient): in other words, akin to semi-external memory models [2, 103] there is enough space to store the vertices, but not the edges of the input graph. These results include *triangle counting* [18, 28, 78], *bipartiteness*, *bipartite matching*, *connected components*, *minimum spanning tree* [53, 54], *matching* [85], and *t-spanners* [20, 48, 54], as well as lower bounds on the $p \times s$ product for distance and girth evaluation [54].

Despite significant success in dealing with graph problems in the *Semi-streaming* model (in fact, $O(n \cdot \text{polylog } n)$ space has come to be known as a “sweet spot” for streaming graph problems [88]), a very natural question, already posed in the seminal works of Munro and Paterson [87] and Henzinger *et al.* [72], i.e., whether it is possible to reduce the space usage at the price of increasing the number of passes, still remains mostly unanswered in the literature. We will provide some first steps in this direction in Section 3.6.

2.2.3 W-Stream

The crucial limitation of the streaming models examined so far is their inability to modify the data stream itself. Motivated by technological factors (today’s computing platforms are equipped with quite large and inexpensive disks, highly optimized for sequential read/write access to data), some authors have recently started to investigate the computational power of less restrictive streaming models, in which the writing of temporary streams is allowed.

In the *W-Stream* model [97], a streaming pass, while reading data from the input stream and processing them in the working memory, produces items that are sequentially appended to an output stream. Streams are pipelined in such a way that the output stream produced during pass i is used as input stream at pass $(i+1)$. It is often assumed that the intermediate streams are not asymptotically longer than the input stream, in keeping with the assumption that the input stream is already huge, and increasing the stream length significantly is not reasonable. More formally, given a *RAM* machine M with a local memory of s bits, we will call a *streaming-writing-pass* (or *s/w-pass* for short) a computation performed by M when reading a stream S of symbols from a given alphabet Σ sequentially, and while doing so, outputting a stream S' of symbols over the same alphabet. This leads to the following definition:

Definition 2.2 *Let $p, s : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. Then* W-Stream*(p, s) denotes the class of problems solvable by a RAM machine using $s(n)$ bits of local memory and $p(n)$ s/w-passes, for an input stream of length n, where the output stream produced at one pass is taken as input stream in the next. The state of the local memory is maintained between passes.*

Clearly, *Stream*$(p,s) \subseteq$ *W-Stream*(p,s) since at every s/w-pass the output stream may simply consist of a copy of the input stream. Intuitively, however, the ability to write intermediate streams should make *W-Stream* more powerful than *Stream*, at least for some problems. We will see that this is indeed the case in Section 3.5, where we will prove various hardness and separation results, that will help elucidate the relationship between *Stream* and *W-Stream*. Here we will simply recall a result by Ruhl [97], namely that if we restrict ourselves to problems in *W-Stream* whose output consists of a single stream element (a restriction we will refer to as *W-Stream-1*), then a *W-Stream* algorithm can be simulated in *Stream*, by building the intermediate streams only implicitly, at the expense of a blow-up of the working memory by a factor of p (number of passes performed by the *W-Stream* algorithm):

Lemma 2.1 ([97]) W-Stream-1$(p,s) \subseteq$ Stream$(p, s \cdot p)$*, even if the lengths of intermediate streams are not bounded by $O(n)$, where n is the size of the input stream.*

Proof. Let M be a machine solving a problem in *W-Stream-1*(p,s), and M_i be the memory content of M at the beginning of pass i $(i = 1, 2, \ldots, p(n))$. M can be simulated in *Stream*$(p, s \cdot p)$ as follows. In the i^{th} streaming pass $(1 \le i \le p(n))$, we simulate i copies of M in parallel. The j^{th} copy of M (for

$j = 1, \dots, i$) performs the same operations as M would during its j^{th} pass. Its memory content starts out as M_j, it reads the stream produced by the $(j-1)^{st}$ copy of M, and writes a stream to be read by the $(j+1)^{st}$ copy of M. The key observation is that the intermediate streams do not need to be explicitly written. We start out by running the i^{th} copy of M. When it wants to read an input symbol, we run the $(i-1)^{st}$ copy of M until it produces an output symbol. In general, when the j^{th} copy of M wants to read a symbol, we run the $(j-1)^{st}$ copy of M until it produces a symbol, unless $j = 1$, in which case we read directly from the input stream. In this manner, in the i^{th} pass we simulate the first i passes of M, ending up with memory contents corresponding to $M_2, \dots, M_{i+1}$ for the next pass. $\Box$

The above lemma establishes that, if we restrict ourselves to problems that require a constant-sized output (but we may easily remove this restriction if we augment *Stream* with a write-only output tape, as was also suggested by Munro and Paterson [87]) and few passes, then the ability to write intermediate stream does not add much computational power. It should be noted, however, that even in this restricted case, the per-item processing time is higher in the *Stream* simulation than in the corresponding *W-Stream-1* algorithm, and so may be the overall number of processed items. The per-item processing time goes up because every time an item is needed, simulations of multiple copies of M (up to $p-1$) have to be run in order to produce it, while the overall number of processed items may increase because the *Stream* simulation will always run on the same input stream, while a *W-Stream* algorithm may produce intermediate streams that are asympotically shorter, at least for some problems (we will see that this is indeed the case in Section 3.5, where we will present some hardness and separation results between *Stream* and *W-Stream*).

We will show a number of fundamental combinatorial problems in the *W-Stream* model in chapters 3 and 4, including *list ranking*, *sorting*, *Euler tour of a tree*, *connected components*, *multiple source shortest-paths*, *minimum spanning tree*, *biconnected components* and *maximal independent set*. All these algorithms exhibit a smooth tradeoff between memory usage and number of passes, and many of them are provably optimal up to polylogarithmic factors, in terms of their $p \times s$ product.

2.2.4 StreamSort

Among the primitives that can efficiently access data in a non-local fashion, sorting is perhaps the most optimized and well understood. This consideration, along with others akin to those that have led to the introduction of the *W-Stream* model (see Section 2.2.3), has induced Aggarwal *et al.* [3] to propose the "streaming and sorting" model (which we will denote in the following as *StreamSort*). This model extends *classical streaming* in two ways:

the ability to write intermediate temporary streams and the ability to reorder them at each pass for free. More in detail, a *StreamSort* algorithm alternates streaming and sorting passes: a streaming pass, while reading data from the input stream and processing them in the working memory, produces items that are sequentially appended to an output stream; a sorting pass consists of reordering the input stream according to some (global) partial order and producing the sorted stream as output. Analogously to *W-Stream*, streams are pipelined in such a way that the output stream produced during one pass is used as input stream in the next: in fact, *StreamSort* can also be viewed as *W-Stream*, augmented with the ability to reorder the temporary stream at each pass at no additional cost. More formally, given a *RAM* machine M with a local memory of s bits that computes a partial order on some alphabet Σ (i.e., given two items in Σ, it returns which one is greater, or whether they are equal or incomparable), a *sorting pass* is defined as a function that takes as input a stream S of symbols from Σ, and outputs a stream S' which consists of S reordered according to the partial order defined by M. A *streaming pass* instead is defined as the computation performed by M as it reads sequentially a stream S of symbols from Σ, and output sequentially a stream S' over the same alphabet. This leads to the following definition:

Definition 2.3 *Let $p_{s/w}, p_{Sort}, s : \mathbb{N} \to \mathbb{N}$ be functions on the natural numbers. Then* StreamSort$(p_{s/w}, p_{Sort}, s)$ *denotes the class of problems solvable by the composition of up to $p_{s/w}(n)$ streaming passes and $p_{Sort}(n)$ sorting passes, each with local memory $s(n)$ for an input stream of length n. The state of the local memory is maintained between passes.*

We will also set:

$$\mathit{StreamSort}(p, s) := \cup_{p'+p'' \leq p} \mathit{StreamSort}(p', p'', s).$$

As shown in [3, 97], the combined use of intermediate temporary streams and of a sorting primitive yields enough power to solve efficiently (within polylogarithmic passes and polylogarithmic memory) a variety of problems, including *undirected s-t connectivity*, *minimum spanning tree*, *maximal independent set*, *substring matching*, *suffix array computation* and various geometrical problems, some of which are provably impossible to solve within similar bounds in the other less powerful streaming models (for example, a linear lower bound on the $p \times s$ product exists in both *Stream* [72] and *W-Stream* [45] for *undirected s-t connectivity*). Some of the results mentioned above are consequences of the fact that *StreamSort* is powerful enough to simulate parallel algorithms for problems in **NC** (and **RNC**) that require a linear number of processors. Section 2.3.3 is devoted to a more extensive discussion of this topic.

Despite major success in solving a wide range of important problems in this (relatively weak) model, to this day it remains an open question whether fundamental graph problems such as *shortest paths* (and even *breadth first search*) can be solved efficiently in *StreamSort*.

2.3 Algorithmic Tools and Techniques

In this section, we recall some algorithmic tools and techniques known in the literature, that will later be exploited for proving many of the original results in this thesis. Subsection 2.3.1 will present a communication complexity-based technique for obtaining lower bounds in *Stream*, which as we will see can also be adapted to *W-Stream*. Subsection 2.3.2 will present the so called "long path property", which will prove a fundamental tool for efficiently solving *shortest paths* in *W-Stream*. Finally, Subsection 2.3.3 will introduce techniques for simulating parallel algorithms in *external memory* and in *StreamSort*; as we will see in Chapter 4, related techniques will yield efficient algorithms for a variety of problems also in *W-Stream*.

2.3.1 Communication Complexity-Based Lower Bounds

Many lower bounds in *Stream*, for problems such as *undirected connectivity*, *k-edge connected components*, *k-vertex connected components* (both with $1 < k < n$), *testing graph planarity* and *finding sinks in a directed graph*, are obtained through reductions from problems for which a communication complexity-based lower bound is known [72].

As an example, consider the *bit-vector disjointness* problem:

Definition 2.4 (Bit vector disjointness) *Let two parties, A and B, have two n-bit-vectors x and y, respectively. The* bit vector disjointness problem (BVDJ) *asks whether x and y are disjoint, i.e., whether $x \cdot y = 0$ (or, equivalently, whether there is an index i such that $x_i = 1$ and $y_i = 1$).*

In order to solve this problem, the two parties necessarily have to exchange $\Omega(n)$ bits or more precisely, any algorithm that outputs the correct answer to the *bit vector disjointness* problem with probability at least $1 - \epsilon$ (for some small enough ϵ) must communicate $\Omega(n)$ bits [80].
Formally, *undirected connectivity* can be defined as follows:

Definition 2.5 (Undirected connectivity) *Given an undirected graph G,* undirected connectivity (UCON) *is the problem of determining whether there is a path between every two nodes in G.*

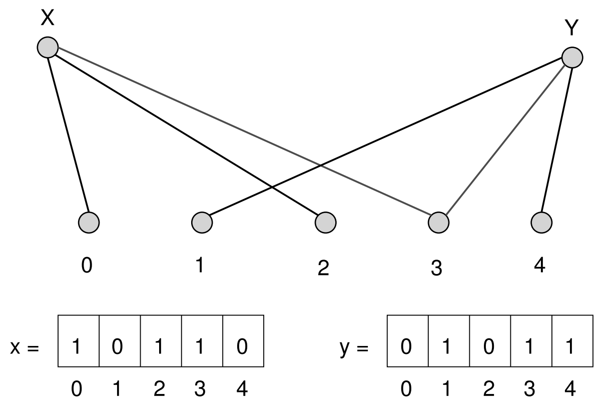


Figure 2.1: Reducing BVDJ to *undirected graph connectivity.*

A lower bound for UCON in *Stream* can be obtained by reducing BVDJ to it as follows.

Theorem 2.1 ([72]) *Any p-pass* Stream *algorithm for* undirected connectivity *requires $s = \Omega(n/p)$ bits of working memory, where n is the number of nodes of the input graph.*

Proof. BVDJ (see Definition 2.4 above) can be reduced to UCON as follows: construct a graph whose node set is $\{X, Y, 0, 1, \ldots, n-1\}$, then insert an edge (X, i) for any index i such that $x_i = 1$ and an edge (Y, i) for any index i such that $y_i = 1$. Ignoring isolated nodes, the constructed graph will be connected if and only if $x \cdot y > 0$. In order to solve *bit-vector disjointness* A creates a stream containing all the edges of the form (X, i), and B does the same for the edges of the form (Y, i). Then A runs a streaming algorithm for *undirected connectivity* on her stream, and when the stream is over she sends the content of her working memory to B. B continues to run the same streaming algorithm, starting from the memory image received from A, until his stream is over. He then sends his memory image to A, who starts a second pass (and so on). Eventually, the streaming algorithm will determine whether the input graph is connected or not. Therefore, by the communication complexity lower bound for *bit-vector disjointness*, A and B must have exchanged $\Omega(n)$ bits [80], and since they exchange $O(s)$ bits at every pass (where s is the size of the working memory in bits), the lower bound on the memory size results. □

As we will show in Section 3.5.3, this technique can be adapted to *W-Stream* yielding lower bounds for many problems, including *undirected connectivity*, *maximal independent set*, *length-2 cycle detection* and *element dis-*

tinctness. The same technique, however, cannot be applied to the more powerful *StreamSort* model, for which, as noted in Section 2.2.4, algorithms exist, that require polylogarithmic space and number of passes, for problems such as *undirected s-t connectivity* and *minimum spanning tree*, for which a linear lower bound on the $p \times s$ product holds in both *Stream* and *W-Stream*.

2.3.2 Random Sampling on Graphs

Randomization and approximation have long proven powerful tools when dealing with the most diverse algorithmic problems, and all the more so in data streaming settings, where typically computational resources are scarce, and good approximate results are often acceptable in lieu of (very hard, or even impossible to compute) exact ones. We will now discuss an interesting combinatorial property of long paths in graphs, which makes the random sampling of a subset of nodes in a graph a particularly fruitful technique in both parallel and streaming settings.

Intuitively, if we pick a subset S of nodes at random from a graph G, then a path with sufficiently many vertices will be likely to intersect S. To the best of our knowledge, this property was first given in [63], and it has subsequently proved a powerful tool for designing efficient algorithms for problems such as *transitive closure* and *shortest paths* [46, 79, 102, 106]. We will present it in a slightly more general form, which is better suited to our future needs:

Theorem 2.2 (Long path property) *Let S be a subset of nodes chosen uniformly at random from an n-node graph G. Let $\{\pi_1, \pi_2, \ldots, \pi_q\}$ be any set of simple paths in G, with $q = O(n^\epsilon)$ for any $\epsilon \geq 0$, each of them containing at least $\frac{cn \ln n}{|S|}$ nodes, for any $c > \epsilon$. Then $Prob\{\pi_1 \cap S = \emptyset \vee \pi_2 \cap S = \emptyset \vee \ldots \vee \pi_q \cap S = \emptyset\} \leq \frac{1}{n^{c-\epsilon}}$.*

Proof. Let π be any simple path containing $\ell \geq \frac{cn \ln n}{|S|}$ nodes. Since a node of G belongs to S with probability $|S|/n$, the probability that π does not contain nodes in S will be:

$$Prob\{\pi \cap S = \emptyset\} = \left(1 - \frac{|S|}{n}\right)^{\ell} \leq e^{-\frac{|S|\ell}{n}} = \frac{1}{n^c}.$$

Since the probability of the union of random events is always smaller than or equal to the sum of the probabilities of the events themselves, we have:

$$Prob\{\bigvee_{i=1}^{q} (\pi_i \cap S = \emptyset)\} \leq \sum_{i=1}^{q} Prob\{\pi_i \cap S = \emptyset\} = \frac{1}{n^{c-\epsilon}}.$$

□

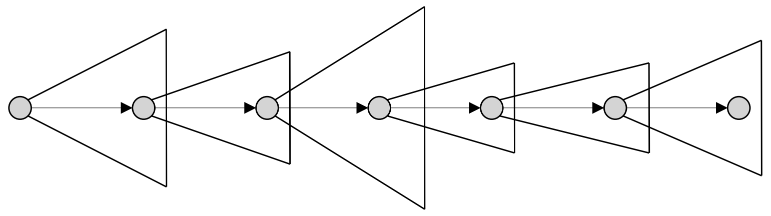


Figure 2.2: Finding a long path as the concatenation of short searches.

The property can be very useful in finding long paths in directed graphs (apparently, an inherently sequential problem) as the concatenation of shorter searches (see Figure 2.2), which can be run in parallel, or equivalently, in a streaming setting, within the same passes. We will take advantage of this technique in Chapter 3, where we will present a *W-Stream* algorithm for *shortest paths* on directed graphs (to the best of our knowledge, the first shortest paths algorithm in a streaming setting).

2.3.3 Simulating Parallel Algorithms

It is well known that algorithmic ideas developed in the context of parallel computational models have inspired the design of efficient algorithms in other models. In particular, in both the *external memory* model and in *StreamSort*, I/O-efficient algorithms are frequently based upon the simulation of parallel algorithms, either directly or through the evaluation of equivalent boolean circuits [3, 97, 38]. As we will see in Chapter 4, related techniques will yield efficient algorithms for a number of graph problems even in the (less powerful) *W-Stream* model.

PRAM Simulations in External Memory.

As shown by Chiang *et al.* [38], efficient external memory algorithms can be derived from PRAM algorithms using a general simulation:

Theorem 2.3 ([38]) *Let A be a* PRAM *algorithm that uses N processors and runs in time T using $O(N)$ space. Then A can be simulated in external memory in $O(T \cdot sort(N))$ I/Os.*

Proof. To simulate the PRAM memory, an array of $O(N)$ items is kept on disk. At every computational step, each processor reads $O(1)$ operands from memory, performs some computation, and then writes $O(1)$ results to memory. To provide the operands for the current step, the elements of a copy of the PRAM memory are sorted according to the indices of the processors that will need them. Then in one scan of this copy, the computation of each processor can be simulated, and the results can be written to disk. Finally, the results of the computation are sorted according to the memory addresses to which the processors would write them, and in one scan they are merged with the PRAM memory. The whole process uses a constant number of scans and sorts, and therefore takes $O(sort(N))$ I/Os. To simulate an entire algorithms, we merely simulate all of its steps. □

Bounded fan-in, topologically sorted boolean circuits, whose description is stored in external memory, can be evaluated using a similar technique.

These techniques produce efficient external memory algorithms for a variety of problems, including *list ranking*, *expression tree evaluation*, *Euler tour*, *connected components*, *biconnected components* and *minimum spanning forest* [38].

Evaluation of Circuits in StreamSort.

As proven in [3, 97] with techniques similar to the PRAM simulations in external memory [38], *StreamSort* is powerful enough to evaluate uniform, linear width, polylogarithmic depth boolean circuits:

Lemma 2.2 ([3]) *A uniform, bounded fan-in boolean circuit with width $O(n)$ and depth $d(n)$ can be evaluated in deterministic* StreamSort *using $d(n)$ streaming and sorting passes, and $O(\log n)$ memory.*

Proof. Inductively, for each level ℓ of the circuit, we generate a stream S_ℓ containing a list of the inputs taken by the circuit nodes on that level, ordered by node (note that S_1 can easily be computed from the input). One streaming pass over S_ℓ suffices to compute the outputs of all nodes on level ℓ. In order to produce $S_{\ell+1}$, these outputs must be rearranged according to the input pattern of level $\ell + 1$. This can be done by labeling the outputs with the numbers of the gates that take them as inputs, and creating duplicates if an output is given as input to multiple gates. Sorting on these labels will yield the desired order. □

Since these circuits can solve problems in **NC** that require a linear number of processors, Lemma 2.2 provides a systematic way for turning parallel algorithms into *StreamSort* ones. Relevant examples include *undirected connectivity* and *maximal independent set* [3].

Chapter 3

Trading Off Space for Passes

3.1 Introduction

In this chapter we present original tradeoff algorithms in the *W-Stream* model, for several fundamental graph problems (namely, *connected components*, *minimum spanning tree* and *multiple-sources shortest paths*). Our algorithms solve these problems (optimally up to a logarithmic factor, in the case of the first two), plus allowing a smooth tradeoff between the working memory size and the number of passes they require. In particular, *connected components* and *minimum spanning tree* can be solved deterministically in $O((n \log n)/s)$ passes, while *multiple-sources shortest paths* in a directed graph with positive integer edge weights up to C can be solved by a randomized algorithm in $O((Cn \log^{3/2} n)/\sqrt{s})$ passes. To the best of our knowledge, this is the first known algorithm for *shortest paths* on directed graphs in a streaming model: previous results on distances in streaming settings are based on the computation of graph spanners, and these yield approximate results for undirected graphs only [20, 48, 54].

We remark that lower bounds of the form $p \times s = \Omega(n)$ (where n is the number of nodes in the input graph) exist for all these problems in both *Stream* and *W-Stream*, but the only known upper bounds in *Stream* assume $s = \Theta(n \log n)$, while our *W-Stream* algorithms adapt to the available working memory, yielding a full range of possible space/passes tradeoffs. It is still an unanswered question in the literature whether such tradeoffs are possible for natural problems even in the less powerful *Stream* model, and this motivates us to conclude the chapter with some observations on the computational power of *W-Stream*, and with some preliminary tradeoff results in *Stream*.

The chapter is organized as follows. Sections 3.2, 3.3 and 3.4 will present the *connected components*, *minimum spanning tree* and *multiple-sources shortest paths* tradeoff algorithms, respectively. In Section 3.5, several hardness and separation results between *Stream* and *W-Stream* will be discussed. Section 3.6 will present some preliminary tradeoff results in *classical streaming*, for the *chain reachability* and *undirected s-t connectivity* problems. Finally, Section 3.7 will summarize the results achieved in the chapter.

3.2 Connected Components

Throughout this section, we will assume that an input graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, is described as an *adjacency stream* [18], i.e., as a stream Σ of edges in arbitrary order (each edge given as a pair of node IDs representing its endpoints), with $|\Sigma| = m$. We will also assume that node IDs can be encoded using $O(\log n)$ bits.

The *connected components* problem can be defined formally as follows:

Definition 3.1 (Connected components) *Given an undirected graph $G = (V, E)$ with n nodes and m edges,* connected components (CC) *is the problem of determining a function $c : V \to \{0, 1, \ldots, n-1\}$ such that $\forall u, v \in V$ $c(u) = c(v)$ if and only if u and v are connected by a path in G.*

We will now present a deterministic *W-Stream* algorithm that computes the connected components of a given graph within $p = O((n \log n)/s)$ passes, where s is the size of the working memory in bits. Since, by adapting to our setting the communication complexity arguments often employed for deriving lower bounds in *Stream* (see Section 2.3.1), it is possible to prove a lower bound of the form $p = \Omega(n/s)$ for *undirected connectivity* in *W-Stream* as well (see Section 3.5.3), and *undirected connectivity* can be reduced to CC, this upper bound is optimal up to a logarithmic factor.

Theorem 3.1 *In* W-Stream*, CC requires $p = \Omega(n/s)$ passes with a space restriction of s bits.*

Our algorithm will elect a representative node for each *connected component* (i.e., for each *maximal* connected subgraph of the input graph), and provide a mapping c from node ID to node ID, that will relate each node to the representative of the connected component it belongs to, thus solving the CC problem. We will first describe the algorithm in detail, then prove its correctness and performance bounds.

3.2.1 Algorithm

Given an undirected graph $G = (V, E)$, let $C(G) = (V, E')$ be the undirected graph on the same vertex set such that $(u, v) \in E'$ if and only if v is the

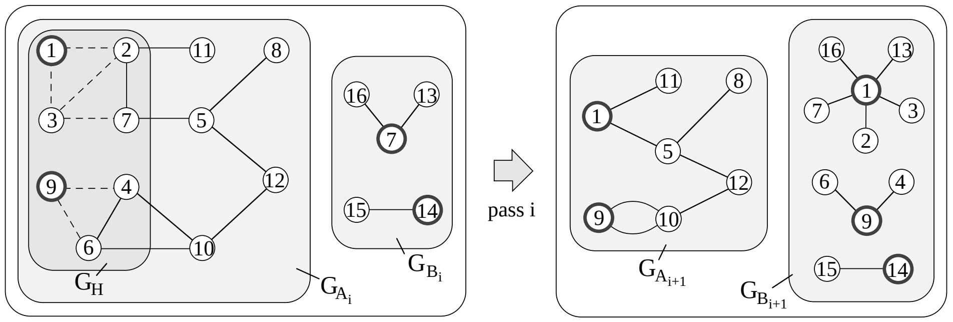


Figure 3.1: Example of the effects of one pass of the *connected components* algorithm.

representative vertex of the connected component of G that contains u. We note that $C(G)$ represents explicitly the connected components of G as stars around component representatives. If L is a list of edges, we denote by $G_L = (V_L, L)$ the graph induced by edges in L. Thus, $G_\Sigma = G$.

The algorithm works as follows. Each intermediate stream Σ_i produced by the algorithm is divided into two consecutive parts A_i and B_i such that G_{B_i} is a collection of stars, and $G_{A_i} \cup G_{B_i}$ has the same connected components as G. At the beginning, $A_0 = \Sigma$ and $B_0 = \emptyset$, and thus $G_{A_0} = G$ and $G_{B_0} = \emptyset$. At the end, $G_{A_p} = \emptyset$ and $G_{B_p} = C(G)$ is the desired result. The generic pass i of the algorithm works in four phases:

1. Read a prefix H of edges from A_i and store in main memory M each newly encountered vertex until either M gets full, or all edges of A_i have been read. Let $G_H = (V_H, H) \subseteq G_{A_i}$ be the graph induced by the edges in the prefix H of A_i read in this phase. As edges are streamed in, also form in M the connected components of G_H, e.g., by building a spanning forest. No output items are produced during this phase.

2. Read all remaining edges from A_i (if any). Let $c(v)$ be the representative vertex of the connected component of G_H that contains v, if $v \in V_H$, and let $c(v) = v$ otherwise. For each input item (u, v) read from A_i such that $c(u) \neq c(v)$, write $(c(u), c(v))$ as output item to A_{i+1}.

3. Read all edges from B_i. For each input edge (u, v) read from B_i, write $(u, c(v))$ as output edge to B_{i+1}.

4. No edges remain to be read from input stream. For each vertex v in V_H that does not appear in A_{i+1}, write $(v, c(v))$ as output edge to B_{i+1}.

The algorithm repeats the generic pass described above until A_i gets empty. We note that phase 1 is a memory loading phase, which stores V_H in main memory along with a sparse certificate of connectivity of G_H (e.g., a spanning forest). Phase 2 produces an output graph $G_{A_{i+1}}$ obtained from G_{A_i} by contracting each connected component of G_H into its representative vertex. Vertices that are in G_{A_i}, but disappear from $G_{A_{i+1}}$ due to the contraction are put in $G_{B_{i+1}}$ by connecting them to their component representatives in G_H. These representatives may be later replaced by newer representatives in successive executions of phase 3 so as to maintain the invariant that G_{B_i} is a collection of stars. The example of Figure 3.1 illustrates the effects of one pass of the algorithm.

3.2.2 Analysis

To prove the correctness of the algorithm, it suffices to check that the following invariant is maintained at each pass.

Invariant 3.1 *For each $i \in \{0, \ldots, p\}$, G_{B_i} is a collection of stars, and $G_{A_i} \cup G_{B_i}$ has the same connected components as G.*

Proof. We prove our claim by induction on the number of passes performed by the algorithm. The base for $i = 0$ is straightforward, since $G_{A_i} = G$ and $G_{B_i} = \emptyset$. We assume by inductive hypothesis that the invariant holds at pass i, and we show that it also holds at pass $(i + 1)$. First, observe that $G_{B_{i+1}}$ is obtained in phases 3 and 4 as union of stars from G_{B_i} and stars that represent the connected components of G_H. If the stars produced in the two phases are not disjoint, the union may not be a collection of stars. For this reason, if a star in G_{B_i} intersects a component of G_H, its center is replaced in phase 3 by its representative in G_H. Thus, $G_{B_{i+1}}$ is a collection of stars. To prove that $G_{A_{i+1}} \cup G_{B_{i+1}}$ has the same connected components as G, observe that each connected component of $G_H \subseteq G_{A_i} \subseteq G_{A_i} \cup G_{B_i}$ is replaced by a star in $G_{A_{i+1}} \cup G_{B_{i+1}}$, and thus connectivity information is maintained. □

Assuming that the main memory M has a size of s bits, we now show that the algorithm terminates in at most $p = O((n \log n)/s)$ passes.

Theorem 3.2 *In* W-Stream, *CC can be solved with $p = O((n \log n)/s)$ passes when the space restriction is s.*

Proof. Without loss of generality, we assume that the input graph G contains no self loops. Indeed, self loops can be easily removed with a preprocessing phase, as follows. Clearly, $O(s/ \log n)$ vertices can be kept in main memory at any time. For each of those vertices, it can be checked whether it is the endpoint of a self loop, and if so, whether it also appears as endpoint of an edge that is not a self loop. If it does, then the self loop can just be dropped,

otherwise the vertex must be stored as a connected component. The operations just described take $O(1)$ passes, and since there are n vertices, it is easy to see that the whole preprocessing phase takes $O((n \log n)/s)$ passes.
Notice that during pass i, all vertices of V_H that are not representatives of connected components of G_H disappear from $G_{A_{i+1}}$ (phase 2). Since G_H is induced by a set of edges, each connected component of G_H contains at least two vertices. Thus, there are at least $|V_H|/2$ vertices in G_{A_i} that disappear from $G_{A_{i+1}}$, so in at most $p \leq 2n/|V_H|$ passes G_{A_p} gets empty. Since phase 1 fills memory M with vertices and a spanning forest of G_H until it gets full, and storing each vertex label requires $\log n$ bits of space, then $|V_H| = \Theta(s/\log n)$. This implies that $p = O((n \log n)/s)$. $\square$

The crucial point in the analysis is the choice of G_H, which is the largest graph induced by a prefix of the stream such that a sparse certificate of its connected components fits in s bits of memory. If G_H is dense, we may contract at each pass a number of edges much larger than s. This makes the number of passes proportional to the number of nodes of the graph, instead of the number of edges.

3.3 Minimum Spanning Tree

Using the *connected components* algorithm described in the previous section as a subroutine, it is also possible to compute a *minimum spanning tree* of a weighted undirected graph in *W-Stream*. A minimum spanning tree is defined as follows:

Definition 3.2 (Minimum spanning tree) *A* minimum spanning tree (MST) *of a connected weighted undirected graph $G = (V, E)$ is a subgraph $G' = (V', E')$ such that:*

- *it contains all the vertices in V,*
- *it is a tree,*
- *the sum of the weights of the edges in E' is minimized.*

We will also refer to the problem of finding a minimum spanning tree of a connected graph, or a minimum spanning forest of an unconnected graph (defined as the union of the minimum spanning trees for each connected component), as the MST problem.
Throughout this section, we will assume that an input graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, is described as a stream of edges in arbitrary order, and each edge is given as a pair of node IDs, denoting its endpoints, and a weight

w. It is assumed that $O(\log n)$ bits suffice to encode edge weights and node IDs.

Our algorithm takes $p = O((n \log n)/s)$ passes to solve the MST problem, and therefore, by the $p = \Omega(n/s)$ lower bound for *undirected connectivity* (which can be reduced to MST) in *W-Stream* (see Section 3.5.3), it is optimal up to a logarithmic factor.

Theorem 3.3 *In* W-Stream*, MST requires $p = \Omega(n/s)$ passes when the space restriction is s bits.*

The algorithm is based on a *greedy* approach, and it takes advantage of the property that given a subset W of vertices, a minimum weight edge having one and only one endpoint in W is in some MST. We will first describe the algorithm, than formally prove its performance bounds.

3.3.1 Algorithm

An MST (or a minimum spanning forest, in the case the input graph consists of more than one connected component) can be progressively built by adding edges as follows. We compute for each vertex the minimum weight edge incident to it. This set of edges E' is added to the MST. We then compute the connected components induced by E' and contract the graph by considering each connected component as a single vertex. We repeat these steps until the graph contains a single vertex or there are no more edges to add. More precisely, we consider at each iteration a contracted graph where the vertices are the connected components of the partial MST so far computed. Denoting $G_i = (V_i, E_i)$ the graph before the i^{th} iteration, the $(i+1)^{th}$ iteration consists of the following steps.

1. for each vertex $u \in V_i$, we compute a minimum weight edge (u, v) incident to u, and flag (u, v) as belonging to the MST. Cycles that might occur due to weight ties are avoided by using a tie-breaking rule (for example, whenever a weight tie occurs, the vertex with lowest ID may be preferred). We will denote by $E'_i = \{(u, v), u \in V_i\}$ the set of flagged edges.

2. we run the CC algorithm of Section 3.2 on the graph (V_i, E'_i). The resulted connected components are the vertices of V_{i+1}.

3. we replace each edge (u, v) by $(c(u), c(v))$, where $c(u)$ and $c(v)$ denote the labels of the connected components previously computed.

3.3.2 Analysis

We will now bound the number of passes required by the algorithm.

Theorem 3.4 *An MST of a weighted undirected graph can be computed in* $O((n \log n)/s)$ *passes in* W-Stream.

Proof. Let $|V_i| = n_i$. The first and the third steps of the algorithm require $O((n_i \log n)/s)$ passes each, since we can process in one pass $O(s/\log n)$ vertices. Computing the connected components also takes $O((n_i \log n)/s)$ passes (see Theorem 3.2), and therefore the i^{th} iteration requires $O((n_i \log n)/s)$ passes. We note that at each iteration we add an edge for every vertex in V_i and thus:

$$|V_{i+1}| \leq |V_i|/2$$

i.e., the number of connected components is divided by at least two. Therefore we obtain that the total number of passes performed in the worst case is given by:

$$T(n) = T(n/2) + O((n \log n)/s)$$

which sums up to $O((n \log n)/s)$. □

3.4 Multiple-Sources Shortest Paths

Let $G = (V, E)$ be a directed weighted graph with n vertices and m edges. The *multiple-sources shortest paths* problem is defined as follows:

Definition 3.3 (Multiple-sources shortest paths) *Given a directed edge weighted graph* $G = (V, E)$ *and a subset of vertices* $S \subseteq V$*, the* multiple-sources shortest paths (MSSP) *problem consists of finding paths from every vertex in* S *to all vertices in* G*, such that the sum of the weights of the constituent edges of each path is minimized.*

We will also refer to the case where S contains a single vertex as the *single-source shortest paths (SSSP)* problem.
In the following we will assume that an input graph is given as a stream of edges in arbitrary order, where each edge is described as a triple (u, v, w_{uv}), with u and v are unique labels (IDs) representing respectively the source and target vertex, and w_{uv} is the edge weight. It is also assumed that $O(\log n)$ bits suffice to encode both vertex labels and edge weights.

We first observe that since *undirected connectivity* can be reduced to SSSP, then the communication complexity-based lower bound for this problem (see Section 3.5.3) also holds for SSSP, and thus for MSSP (and the same lower bound obviously also holds in the more restrictive *Stream* model):

Theorem 3.5 *In* W-Stream *(and thus in* Stream*), SSSP requires $p = \Omega(n/s)$ passes for a space restriction of s bits.*

This implies that, if we want to achieve sublinear space $s = o(n)$, then $p = \omega(1)$ passes are required. One may wonder whether $p = O(1)$ passes would be enough to solve the problem using $s = O(n)$ space. Unfortunately, as showed by Feigenbaum *et al.* in [54] a higher lower bound can be proven in the *Stream* model: the lower bound implies that finding vertices up to distance $d = O(1)$ from a given source in less than d passes requires $\Omega(n^{1+1/2d})$ space. Since p is constant and *W-Stream* can be simulated in *Stream* at the price of increasing the size of the working memory by a factor of p (see Section 2.2.3 and [97]), it is not difficult to see that this result also holds in *W-Stream.* This confirms that space efficient algorithms for SSSP in both *Stream* and *W-Stream* always require multiple passes.

We also remark that finding efficient streaming algorithms for the simpler problem of breadth-first traversal of a graph has been posed as an open problem even in the *StreamSort* model [3].

In the remainder of this section, we devise the first algorithm for *single-source shortest paths* in directed graphs in a streaming model. In particular, we will prove the following theorem:

Theorem 3.6 *In* W-Stream*, MSSP from $\rho \cdot \sqrt{s/\log n}$ sources, for any $\rho \in (0,1)$, can be solved with $p = O((C \cdot n \cdot \log^{3/2} n)/\sqrt{s})$ passes in directed graphs with positive integer edge weights up to C under a space restriction of s bits. Distances produced by the algorithm are correct with probability at least $1 - 1/n^{\beta}$ for any positive constant β. The size of each intermediate stream is $O(m + n \cdot \sqrt{s/\log n})$.*

Notice that, for $C = o(\sqrt{s}/\log^{3/2} n)$ we can get both p and s sublinear in n.

3.4.1 Overview of the Algorithm

A typical issue in streaming settings where edges are given in arbitrary order is that following a path seems to require in the worst case as many passes as its length. Finding long paths may therefore require lots of passes. To overcome this difficulty, we argue that, if we were able to find long paths as the concatenation of short paths built "in parallel" within the same passes, this would result in a substantial reduction of the worst-case number of passes required to follow a path of arbitrary length. Similarly to previous algorithms for path problems in parallel and dynamic settings (see, e.g., [71, 102]), the main idea of our algorithm is to perform many short searches from a random subset of vertices of the graph "in parallel". This yields short distances in the graph. To find longer distances, the algorithm stitches together short paths. Using a probabilistic argument, we can prove that distances obtained in this way are correct with high probability.

3.4.2 Finding Distances up to ℓ

Let $A = \{c_1, c_2, \ldots, c_{|A|}\}$ be a subset of vertices of the graph and let $\ell > 0$ be an integer parameter. As a first ingredient for solving SSSP in *W-Stream*, we describe a procedure `shortDist`(A, ℓ) that finds the distances from each source $c_j \in A$ to all other vertices that are up to distance ℓ from c_j in

$$p = O\left(\frac{n\,|A|\,\log n}{s} + \ell\right)$$

passes in a graph with positive integer edge weights.

Our procedure is a multi-source, bounded depth, streamed implementation of Dijkstra's algorithm [40], where the "priority queue" is maintained implicitly on intermediate streams. We will now first describe how distance information is kept on intermediate streams (i.e., the *stream layout*), and then get into the details of the algorithm itself, and its analysis.

Stream Layout.

Let $\{\gamma_1, \gamma_2, \cdots, \gamma_q\}$ be a partition of the input stream $\Sigma_0 = \Sigma$ into the minimum number q of groups γ_i such that:

- all edges in a group γ_i share the same end vertex y_i, i.e.,

$$\gamma_i = (a, y_i, w_{ay_i})\,(b, y_i, w_{by_i}) \cdots (z, y_i, w_{zy_i})$$

- the concatenation of groups γ_i yields Σ_0, i.e.,

$$\Sigma_0 = \gamma_1\,\gamma_2 \cdots \gamma_q$$

Notice that, if the edges in the input stream Σ were ordered by their end vertex, there would exist a unique group γ_i per vertex. In general, the same end vertex may be shared by more than one group, i.e., it may be $y_a = y_b$ with $a \neq b$.

Each intermediate stream Σ_h, with $h > 0$, created by the algorithm will have the form

$$\Sigma_h = \gamma_1\,\delta_1\,\gamma_2\,\delta_2 \cdots \gamma_q\,\delta_q$$

where

$$\delta_i = (d_{i1}, f_{i1})\,(d_{i2}, f_{i2}) \cdots (d_{i|A|}, f_{i|A|})$$

The following invariants will then be maintained throughout the execution of the algorithm:

Invariant 3.2 *In any intermediate stream generated by algorithm* `shortDist`*, for each pair* $(d_{ij}, f_{ij}) \in \delta_i$*,* d_{ij} *is an upper bound to the distance* $dist_{c_j y_i}$ *from* $c_j \in A$ *to* y_i*.*

and

Invariant 3.3 *In any intermediate stream generated by algorithm* `shortDist`*, flag* f_{ij} *is true if and only if* y_i *is* settled *w.r.t.* c_j*. (A vertex* y_i *is said to be* settled *w.r.t.* c_j *if and only if* $dist_{c_j y_i}$ *has been correctly determined by the algorithm.)*

In a preliminary pass, the algorithm lets $f_{ij} = \mathit{false}$ for each i and j; it also lets $d_{ij} = 0$ if $c_j = y_i$, and $d_{ij} = +\infty$ otherwise. The goal of successive passes is to progressively decrease each d_{ij} to the weight of a minimum weight path from c_j to y_i that goes through one of the edges in γ_i.

Algorithm.

The core loop of algorithm `shortDist` alternates *extraction* and *relaxation* passes. During an extraction pass, the algorithm loads in main memory, for each $c_j \in A$, a pool P_{c_j} of at most $k = s/(|A| \cdot \log n)$ vertices v together with their exact distance $dist_{c_j v}$ from c_j (after the first extraction pass, each pool P_{c_j} includes only vertex c_j and $dist_{c_j c_j} = 0$). During a relaxation pass, the algorithm improves the distance upper bounds d_{ij} using edges in γ_i that emanate from P_{c_j}. In more details:

- *Extraction pass.* Let $d_j(v) = \min_{i:v=y_i}\{d_{ij}\}$ be the *priority* of v w.r.t. c_j. For each $c_j \in A$, load in P_{c_j} up to k unsettled vertices with the same minimum priority w.r.t. c_j, if it does not exceed ℓ. For each vertex v in P_{c_j}, it holds that $dist_{c_j v} = d_j(v)$. When all P_{c_j}'s get empty, the algorithm halts.
- *Relaxation pass.* For each $i = 1, 2, \ldots, q$, decrease each d_{ij} in the output δ_i to the weight of a minimum weight path from c_j to y_i that goes through one of the edges in γ_i that emanate from P_{c_j}. Also, make vertices in each P_{c_j} settled w.r.t. c_j by letting $f_{ij} \leftarrow \mathit{true}$ in the output stream for each $y_i \in P_{c_j}$.

We remark that all the vertices that are in each pool P_{c_j} at the end of an extraction pass have exactly the same distance from c_j. All these vertices would be extracted from the priority queue in consecutive iterations of a classical implementation of Dijkstra's algorithm with source c_j. When the algorithm is over, for each c_j and each vertex v that is settled w.r.t. c_j, the distance $dist_{c_j v}$ is implicitly encoded in the output stream as $\min_{i:v=y_i}\{d_{ij}\}$ and can be easily made explicit with a simple post-processing in $O((n \log n)/s)$ passes.

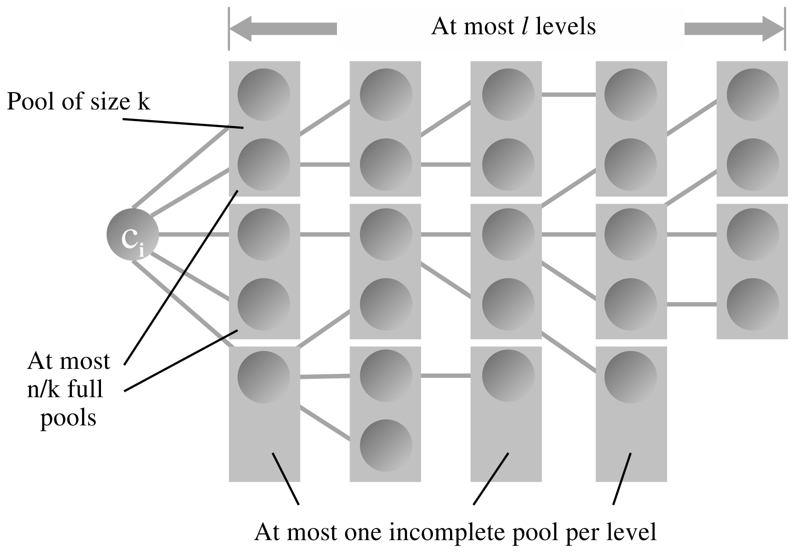


Figure 3.2: Bounding the number of passes of algorithm `shortDist`(A, l): let $c_i \in A$ be the vertex whose pool is the last to get empty. During one execution, the total number of pools relative to c_i cannot exceed $(n/k + l)$. Since the algorithm generates a new pool for c_i every two passes, the bound on the total number of passes follows.

Analysis.

We now discuss the number of passes required by algorithm `shortDist`.

Lemma 3.1 *Algorithm* `shortDist`(A, ℓ) *runs in* $p = O(\frac{n\,|A|\,\log n}{s} + \ell)$ *passes using* s *bits of working memory and intermediate streams of size* $O(m \cdot |A|)$.

Proof. The algorithm keeps in the working memory up to $k = s/(|A| \cdot \log n)$ vertices in each of the $|A|$ pools P_{c_j}. Since storing each vertex label requires $\log n$ bits, the algorithm uses at most $k \cdot |A| \cdot \log n = s$ bits of main memory. The bound on the size of intermediate streams follows from the fact that each of them contains $m + q \cdot |A|$ items and q can be as high as m in the worst case.

To bound the number of passes, consider the vertex $c_j \in A$ such that P_{c_j} is the last pool to get empty. Let $P_1, P_2, \cdots, P_t$, be the content of pool P_{c_j} after successive extraction passes. We say that P_i is *full* if it contains exactly k vertices, and it is *incomplete* otherwise. Notice that, since each vertex appears in at most one P_i, there can be at most (n/k) full P_i's. We now bound the number of incomplete P_i's. Note that in each set P_i, all vertices have the same distance from c_j. Denote this distance by $d(P_i)$. Set P_i can be incomplete only if $d(P_i) < d(P_{i+1})$, or $i = t$. Since $d(P_t) \leq \ell$ and edge weights are positive integers, there can be at most ℓ passes i such that $d(P_i) < d(P_{i+1})$, and thus at most ℓ P_i's can be incomplete. Hence, the total number t of P_i's cannot exceed $(n/k + \ell)$. Since the algorithm generates a new P_i every two passes in the core loop, then it performs a total number of

$$p = O(t) = O(n/k + \ell) = O\left(\frac{n\,|A|\,\log n}{s} + \ell\right)$$

passes. □

Notice that for $A = \{t\}$ algorithm `shortDist` solves SSSP up to distance $\ell = O((n \log n)/s)$ from a given source t in $O((n \log n)/s)$ passes. By Theorem 3.5, this bound is optimal in *W-Stream* up to a log factor.

3.4.3 Reducing the Size of Intermediate Streams

In this section we show how to reduce the size of intermediate streams produced by algorithm `shortDist`, from $O(m \cdot |A|)$ items to $O(m + n \cdot |A|)$. The main idea is to preprocess the input stream so as to reduce the number q of groups γ_i before starting the `shortDist` algorithm. To this aim, we simply partition the input stream Σ into $\max\{1, m/(n \cdot |A|)\}$ subsequences of size $\leq n \cdot |A|$ each, and we reorder edges (x, y, w_{xy}) in each subsequence by end vertex y. This can be done in $O((n \cdot |A| \cdot \log n)/s)$ passes by using our *W-Stream* sorting algorithm (see Section 4.3 for a complete description). The preprocessing can thus be performed within the same asymptotic number of passes as `shortDist`. Notice that the number of groups γ_i in each reordered subsequence cannot be larger than n. Hence, the total number q of groups γ_i in the whole preprocessed stream given as input to `shortDist` will not exceed $n \cdot \max\{1, m/(n \cdot |A|)\} = \max\{n, m/|A|\}$. The size of each intermediate stream in `shortDist` will therefore be

$$m + q \cdot |A| \leq m + \max\{n \cdot |A|, m\} = O(m + n \cdot |A|)$$

as claimed.

3.4.4 Finding All Distances from a Given Source

We now describe an algorithm `sssp`(G, t) that solves SSSP with source t in a graph $G = (V, E, w)$ with n vertices and positive integer edge weights up to C in $O((C\, n \log^{3/2} n)/\sqrt{s})$ passes, assuming a space restriction of s bits in the *W-Stream* model. Algorithm `sssp`(G, t) works as follows:

1. Pick a subset $A \subseteq V$ of $\sqrt{s/\log n}$ vertices, including source t. All vertices but t are chosen uniformly at random.

2. Find distances up to $\ell = (\alpha\, C\, n \log^{3/2} n)/\sqrt{s}$ in G from each of the vertices in A, where α is any constant > 2.
3. Build a weighted graph $G^* = (A, E^*, w^*)$ on vertex set A such that there is an edge $(c_1, c_2) \in E^*$ with weight $w^*_{c_1 c_2} = dist_{c_1 c_2}$ if and only if $dist_{c_1 c_2} \leq \ell$.
4. Compute distances $dist^*_{tc}$ from $t \in A$ to all other vertices $c \in A$ in G^*.
5. For each vertex $v \in V$ whose distance from t has not been determined in step 2 being higher than ℓ, compute it as $dist_{tv} = \min_{c \in A}\{\, dist^*_{tc} + dist_{cv}\,\}$.

Before describing a *W-Stream* implementation of `sssp`, we prove that all distances larger than ℓ computed by the algorithm are correct with high probability, assuming that distances up to ℓ computed in step 2 are correct.

Lemma 3.2 *The probability that all distances $dist_{tv} > \ell$, for any $v \in V$, computed by algorithm* `sssp` *are correct is at least* $1 - 1/n^{\alpha-2}$.

Proof. For any $v \in V$, if $dist_{tv} > \ell$, then it is obtained in step 5 as $dist_{tv} = \min_{c \in A}\{\, dist^*_{tc} + dist_{cv}\,\}$. Since edge weights of G are $\leq C$, then any shortest path from t to v in G will necessarily contain at least $r = \ell/C$ edges. Let π_{tv} be any shortest path from t to v. It can be broken into at most $n/r \leq n$ disjoint subpaths with r vertices. Since algorithm `sssp` computes $n - 1$ distances, and each shortest path can be broken into subpaths as we have just shown, we will have at most $O(n^2)$ subpaths of r vertices. By Theorem 2.2, the probability that each of them contains a vertex of A will be at least $1 - 1/n^{\alpha-2}$. This implies that, with probability at least $1 - 1/n^{\alpha-2}$, each shortest path π_{tv} can be broken into the concatenation of subpaths of at most r vertices of the form $\pi_{c_i c_j}$, with $c_i, c_j \in A$, plus one final subpath of the form $\pi_{c^* v}$, where c^* minimizes $\min_{c \in A}\{\, dist^*_{tc} + dist_{cv}\,\}$. Since w.h.p. each $\pi_{c_i c_j}$ is a shortest path with weight at most ℓ, then it corresponds to an edge $(c_i, c_j) \in E^*$. Thus, w.h.p. the value $dist^*_{tc^*}$ computed in step 4 is the correct distance from t to c^*. To conclude the proof, observe that $dist_{c^* v}$ has also weight at most ℓ w.h.p., and thus it has been correctly determined in step 2. Thus, $dist_{tv} = dist^*_{tc^*} + dist_{c^* v}$, for all $v \in V$, with probability at least $1 - 1/n^{\alpha-2}$. □

Implementation.

In this section we sketch how steps 1–5 of algorithm `sssp` can be implemented in *W-Stream*:

1. As $|A| = \sqrt{s/\log n}$, then vertices of A can be sampled and maintained in main memory.

2. To find distances up to ℓ from each vertex in A, we can just run algorithm `shortDist`(A, ℓ) described earlier in this section. The algorithm stores distances on the output stream.
3. Graph G^* can be stored in main memory, since it requires no more than $|A|^2 \log n = s$ bits. To build it, we can just make one pass and build an $|A| \times |A|$ weight matrix w^* such that for each $c_j, c \in A$ $w^*_{c_j c} = \min_{i:c=y_i}\{ d_{ij} \}$.
4. Distances $dist^*_{tc}$ can be computed by running any internal-memory single-source shortest paths algorithm on G^* with source t, and can be stored in main memory using $O(|A|/\log n) = O(\sqrt{s}\log^{3/2} n)$ bits of space.
5. Compute final distances for $s/\log n$ vertices $K \subseteq V$ at a time. For each K, compute in one pass $dist_{tv} = \min_{i,j\,:\,v=y_i}\{dist^*_{tc_j} + d_{ij}\}$ for each $v \in K$. At the end of the pass, flush computed distances to the output stream in any desired format.

Analysis.

We now discuss the time and space requirements of algorithm `sssp`.

Lemma 3.3 *Algorithm* `sssp` *requires $O((C\,n\,\log^{3/2} n)/\sqrt{s})$ passes, when using s bits of main memory and intermediate streams of size $O(m+n\cdot\sqrt{s/\log n})$ in the* W-Stream *model.*

Proof. Steps 1 and 4 are entirely performed in main memory, and thus require no streaming passes. Step 3 requires one pass and step 5 takes $O((n \log n)/s)$ passes. The entire procedure is dominated by the number of passes of algorithm `shortDist` in step 2, which is

$$O\left(\frac{n\,|A|\,\log n}{s} + \ell\right) = O\left(\frac{C\,n\,\log^{3/2} n}{\sqrt{s}}\right)$$

by Lemma 3.1. The size of intermediate streams also follows from Lemma 3.1 and from the preprocessing technique described thereafter. The largest data structure maintained by the algorithm in main memory is matrix w^* created in step 3, which requires s bits. □

3.4.5 Dealing with Multiple Sources

The algorithm described in Section 3.4.4 can be extended to deal with $\rho \cdot \sqrt{s/\log n}$ sources within the same asymptotic bounds, for any $\rho \in (0, 1)$. Of the $\sqrt{s/\log n}$ vertices loaded into set A in step 1, let only a $(1-\rho)$ fraction of these be chosen uniformly at random, while the remaining vertices are taken to be sources for the shortest paths problem. In step 2, we find distances up to $\ell/(1-\rho)$. Using algorithm `shortDist`$(A, \ell/(1-\rho))$ this will take $O((C\, n \log^{3/2} n)/\sqrt{s})$ passes. Notice that, since only a $(1-\rho)$ fraction of A is chosen at random, algorithm `shortDist` needs to find paths of length up to $\ell/(1-\rho)$. In view of Lemma 3.2, this is crucial for maintaining correctness with high probability. Step 3 of the algorithm remains unchanged, while in step 4 the distances from each source to all other vertices in main memory can be computed, by running any internal-memory single-source shortest paths algorithm for each source vertex. The results can be stored in internal memory. The final distances can be computed by repeating step 5 of the original algorithm once for every source. Since there are $\rho \cdot \sqrt{s/\log n}$ sources, this will take

$$O\left(\sqrt{\frac{s}{\log n}} \cdot \frac{n \log n}{s}\right) = O\left(\frac{n \log^{1/2} n}{\sqrt{s}}\right)$$

passes. Therefore the number of passes of the entire algorithm is determined by the asymptotic performance of the procedure `shortDist`, and this is the same as in the *single-source* case. We call the algoritm described in this section `mssp` and we summarize its bounds in the following lemma.

Lemma 3.4 *Algorithm* `mssp` *requires* $O((C\, n \log^{3/2} n)/\sqrt{s})$ *passes, when using* s *bits of main memory and intermediate streams of size* $O(m+n\cdot\sqrt{s/\log n})$ *in the* W-Stream *model.*

3.5 Hardness and Separation Results

The algorithms presented in sections 3.2, 3.3 and 3.4 show that the ability to write intermediate streams makes it possible to obtain, at least for some problems, a full range of possible space/passes tradeoffs in *W-Stream*, whereas the only known upper bounds in *Stream* assume $s = \Theta(n \log n)$. These results naturally raise the question of whether *W-Stream* is more powerful than *Stream*. In this section we therefore study some aspects related to the computational power of *W-Stream*. We first exemplify problems in *W-Stream* that are impossible to solve in *Stream* for a given space restriction, and then present problems that require a smaller number of processed items in *W-Stream* than in *Stream*; note that in *W-Stream* the size of intermediate streams can vary from pass to pass, while in *Stream* the same input stream is read at each pass: counting the total number of processed stream items, rather than the number of passes, may therefore be a more accurate measure for comparing algorithms in the two models. On the other hand, we also provide examples of problems that are as hard in *W-Stream* as in *Stream* for a given space restriction (regardless of the number of passes). We obtain this result by adapting to *W-Stream* classical communication-complexity arguments used for proving lower bounds in *Stream*. This kind of arguments cannot instead be applied to *StreamSort*, because of the non-local nature of the sorting primitive. In this model, in fact, graph problems such as *undirected s-t connectivity* can be solved within logarithmic space and passes (see Section 2.2.4), whereas lower bounds of the form $p \times s = \Omega(n)$ exist in both *Stream* and *W-Stream*.

3.5.1 Breaking the Space Wall

From a space complexity perspective, intermediate streams can be thought of as part of the algorithm's working memory. It is therefore conceivable that one should be able to solve problems in *W-Stream* with a space restriction that would make them unsolvable in *Stream*. We will show in this section that this is indeed the case.

Consider, for instance, the following problem:

Definition 3.4 (Parenthesis language recognition) *Let L be the context-free parenthesis language* $[\mathtt{S} \rightarrow ()|(\mathtt{S})|(\mathtt{SS})]$. *Let x be a string of n symbols in* $\{(,)\}$ *represented as a data stream. The* parenthesis language recognition (PLR) *problem consists in finding out whether $x \in L$.*

We first prove that PLR cannot be solved in *Stream* using less than logarithmic working memory:

Lemma 3.5 *$PLR \notin \mathrm{Stream}(p, o(\log n))$, for any number p of passes.*

Proof. Since the parenthesis language is a nonregular context-free language, we can use the following result of Alt *et al.* [8]: if L is a nonregular deterministic context–free language and $L \in NSPACE(s(n))$, then the recognition of L requires space $s(n) \geq c \cdot \log n$ for some constant c and infinitely many n. Clearly, this lower bound also applies to *Stream* algorithms and implies that, independently of the number of passes, PLR cannot be solved using less than logarithmic space. □

On the other hand, we can easily solve PLR in *W-Stream* with a constant size working memory, by removing pairs of consecutive matching parentheses from the stream at each pass, until possible, and returning true if the stream gets empty. Hence, $PLR \in$ *W-Stream*$(n, O(1))$. Our separation result immediately follows from this observation and from Lemma 3.5:

Theorem 3.7 $\mathrm{Stream}(n, O(1)) \subset \mathrm{W\text{-}Stream}(n, O(1))$.

3.5.2 Reducing the Number of Processed Items

In principle, the ability, provided by *W-Stream*, to manipulate the data stream raises the question of whether it might be possible, at least for some problems, to discard at each pass items that are no longer useful, thus reducing the overall number of items that an algorithm has to process. In this section, we give a positive answer.

Consider, as an example, the following FORK problem:

Definition 3.5 (FORK) *Let A and B be two vectors of n numbers with $A[1] = B[1]$ and $A[n] \neq B[n]$. The* FORK *problem consists of finding a "fork" index i such that $A[i] = B[i]$ and $A[i+1] \neq B[i+1]$.*

Assume that A and B are given as an input stream of the form

$$A[1], A[2], \ldots, A[n], B[1], B[2], \ldots, B[n]$$

with items in $\{1, \ldots, n\}$. The following lower bound on the space $\times$ passes product follows from a communication complexity lower bound on FORK by Grigni and Sipser [64] (we will discuss in detail how lower bounds in *W-Stream* can be derived from communication complexity lower bounds in Section 3.5.3):

Lemma 3.6 *FORK in* W-Stream *(and thus in* Stream*) requires $p \times s = \Omega(\log^2 n)$.*

Using this result, we can prove a separation result between *Stream* and *W-Stream* with respect to the number of processed items:

Theorem 3.8 *FORK can be (optimally) solved in* W-Stream *with $s = O(\log n)$ space, $p = O(\log n)$ passes and $O(n)$ processed items. This is impossible to achieve in* Stream.

Proof. Lemma 3.6 implies that, if we stick to logarithmic space, then the number of passes of any streaming algorithm solving FORK must be $p = \Omega(\log n)$. Since in *Stream* we have to process all the items in the input stream at each pass, it follows that the number of processed items of any *Stream* algorithm must be $\Omega(n \log n)$ when $s = O(\log n)$.
Instead, we can solve FORK in *W-Stream* more efficiently as follows. Consider a simple binary search-like algorithm, recurring upon the following conditions:

(1) if $A[n/2] = B[n/2]$, then there must be a fork index in the second half of the vectors;

(2) if $A[n/2] \neq B[n/2]$, then there must be a fork index in the first half of the vectors.

At each pass, we can thus halve the size of the intermediate stream, just by not copying the uninteresting half of the input stream. It is easy to see that this algorithm uses $O(\log n)$ space, runs in $O(\log n)$ passes, and processes only $O(n)$ items overall. □

3.5.3 Hardness Results

Even if the use of intermediate streams makes *W-Stream* more powerful than *Stream* for some problems, as we have seen in sections 3.5.1 and 3.5.2, in this section we will show that for other problems instead *W-Stream* maintains most, if not all of the hardness of *classical streaming* (except for possibly simplifying the task of designing streaming algorithms).

Ruhl and Aggarwal *et al.* [3, 97] already observed that, for a small number of passes, intermediate streams do not help much, regardless of the problem considered. Indeed, they showed that any *W-Stream* algorithm can be simulated in *Stream* at the price of increasing the size of the working memory by a factor of p (see Section 2.2.3), making intermediate streams unuseful when p is small.

In the following we show that *Stream* lower bounds for a number of fundamental problems carry over to *W-Stream*, and also that there are problems for which using intermediate streams cannot lead to any polynomial improvement, even regardless of the number of passes. As an example, we take the following problem:

Definition 3.6 (Element distinctness) *Given a stream S of n numbers in $\{1, \dots, n\}$, the* element distinctness (ED) *problem asks whether there are any duplicates in S.*

We first give a lower bound on the passes × space product for ED, obtained by adapting to the *W-Stream* setting a communication complexity argument often used for proving lower bounds in *Stream* (see Section 2.3.1 and [72]).

Theorem 3.9 *Any* W-Stream *algorithm for* element distinctness *requires $p \times s = \Omega(n)$.*

Proof. Consider the bit-vector-disjointness problem (see Definition 2.4). This problem can be reduced to ED in the following way. A creates a stream containing the indices corresponding to the 1's in vector x, and B does the same for vector y. Then A runs a *W-Stream* algorithm for *element distinctness* on her stream, producing an intermediate stream, and when the input stream is over she sends the content of her working memory to B. B continues to run the same *W-Stream* algorithm starting from the memory image received from

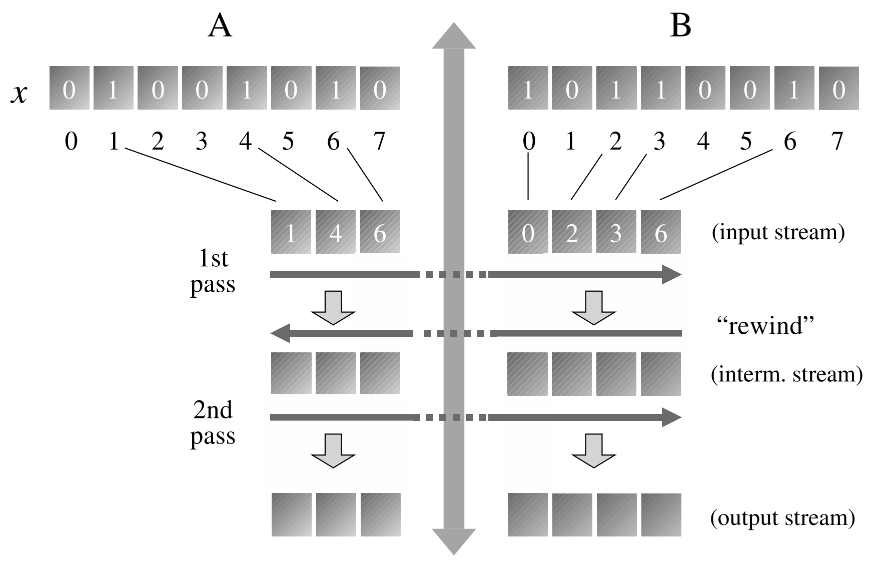


Figure 3.3: Reduction from *bit vector disjointness* to *element distinctness*, and distributed execution of a *W-Stream* algorithm for *element distinctness*: every time the boundary between A and B is crossed (dashed line) the content of the working memory (i.e., the algorithm's internal state) has to be transmitted.

A, reading from his own input stream and producing his own intermediate stream. When the stream is over, B sends his memory image back to A, who starts a second pass by taking as input the intermediate stream that she produced at the previous pass. At the end, the streaming algorithm will determine whether all the elements in the two input streams are distinct or not: notice that the elements are distinct if and only if $x \cdot y > 0$, which is exactly the solution to *bit-vector disjointness*. If, by contradiction, the total working memory used by A and B has size $o(n/p)$, then the total number of bits sent between A and B in p passes would be $o(n/p) \cdot p = o(n)$, which would violate the $\Omega(n)$ communication complexity lower bound for *bit-vector disjointness* [80]. □

Since there is a folklore *Stream* algorithm that solves ED within $p = O((n \log n)/s)$ passes (it suffices to divide the stream into $O((n \log n)/s)$ consecutive buckets of size $O(s/\log n)$, load each bucket in main memory and compare it with the rest of the stream, to identify duplicates), the use of intermediate streams cannot yield any polynomial improvement for this problem.

We remark that arguments similar to the proof of Theorem 3.9 can be used for proving also other lower bounds in *W-Stream*, e.g., for problems such as *maximal independent set* (Theorem 4.13), FORK (Lemma 3.6), *length-2 cycle detection* (Theorem 4.14) and *sorting* (Theorem 4.6). Moreover, the classical communication-complexity based lower bound for *undirected connectivity* in *Stream* (Theorem 2.1) can be adapted to *W-Stream* as well, yielding the following result:

Theorem 3.10 *In* W-Stream, *undirected connectivity requires $p = \Omega(n/s)$ passes with a space restriction of s bits.*

This in turn implies analogous *W-Stream* lower bounds for graph problems such as *connected components* (Theorem 3.1), *minimum spanning tree* (Theorem 3.3), *shortest paths* (Theorem 3.5) and *biconnected components* (Theorem 4.11), to which *undirected connectivity* can be reduced.

3.6 Tradeoffs in Classical Streaming

In sections 3.2 through 3.4 we have proven a number of tradeoff results for fundamental graph problems (namely, *connected components*, *minimum spanning tree* and *shortest paths*) in the *W-Stream* model. An interesting open question, however, is whether similar tradeoffs can be achieved even in the more restrictive *Stream* model. We will now provide first a space/passes tradeoff result for a simpler specialized variant of the more general *reachability* problem in a directed graph (namely, *chain reachability*), and then a limited tradeoff result for a natural graph problem such as *undirected s-t connectivity.* Both results are in the *Stream* model.

3.6.1 Chain Reachability

We define the *chain reachability* problem as follows.

Definition 3.7 (Chain reachability) *Given a directed chain graph (i.e., a directed graph in which every node has only one ingoing and one outgoing edge, except for a* start node *that has no incoming edge and an* end node *that has no outgoing edge) $C = (V, E)$, and two vertices $u, v \in V$,* chain reachability *is the problem of determining whether there is a directed path from u to v in C.*

By using a sampling-based technique inspired by the *shortest paths* algorithm described in Section 3.4, we can prove the following upper bound for *chain reachability* in *Stream.*

Theorem 3.11 Chain reachability *in* Stream *can be solved in $O((n \log^2 n)/s)$ passes with high probability, when the space restriction is s bits.*

Proof. We pick a subset $S \subseteq V$ of $s/(2 \log n)$ vertices, including verices u and v, between which *chain reachability* has to be determined. All vertices but u and v are chosen uniformly at random. Then at each pass we grow forward paths from the vertices in S, keeping track in main memory of the original set S and of the frontier nodes. By Theorem 2.2, within

$$p = (cn \log n)/|S| = O((n \log^2 n)/s)$$

passes we will have found a directed path from u to v, if it exists, with high probability. □

Therefore, a smooth space/passes tradeoff can be achieved even in *classical streaming*, at least for a simplified version of a fundamental graph problem.

3.6.2 Undirected s-t Connectivity

Undirected s-t connectivity is the following problem:

Definition 3.8 (Undirected s-t connectivity) *Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$,* undirected s-t connectivity (USTCON) *is the problem of determining whether there is a path connecting s and t in G.*

In their paper, Broder *et al.* [27] describe an internal memory algorithm for *undirected s-t connectivity* that achieves a tradeoff between memory usage and execution time. The algorithm is a Monte Carlo randomized one, and is based on the idea of reiterated random walks of Aleliunas *et al.* [5]. The algorithm achieves the space-time tradeoff by picking at random a subset of k vertices, or *landmarks*, that includes s and t (while all other vertices are selected with probability proportional to their degrees), and executing random walks from each of them, keeping track of whether s and t get connected in the process. The results of Broder *et al.* can be summarized by the following theorem.

Theorem 3.12 ([27]) *There is an algorithm that decides USTCON with one sided error using space k (in memory words of $O(\log n)$ bits each) and time $O(m^2 \log^5 n/k)$. If s and t are in the same connected component, the algorithm outputs YES with probability $1 - O(n^{-1})$, otherwise it outputs NO.*

As pointed out by the authors themselves, the algorithm is easily parallelizable using k processors and $O(k)$ memory words [27].

More recently, Feige [52] has proposed a new landmark distribution scheme, called *inverse distribution* ($p/2$ nodes chosen with probability linear in their degrees, and $p/2$ chosen with probability proportional to the inverse of their degrees). The new landmark distribution scheme, coupled with tighter analysis, has led to the following improved tradeoff result.

Theorem 3.13 ([52]) *There is an algorithm A that achieves a time space tradeoff of $ST = \widetilde{O}(m\widehat{R})$, where $\widehat{R} = \sum_v 1/d_v$ (where d_v denotes the degree of vertex v). More explicitly, for any T in the range $m \leq T \leq m\widehat{R}$, algorithm A can work in time $\widetilde{O}(T)$ and space $\widetilde{O}(m\widehat{R}/T)$ (in memory words of $O(\log n)$ bits). For any S in the range $1 \leq S \leq \widehat{R}$, algorithm A can work in space $\widetilde{O}(S)$ and time $\widetilde{O}(m\widehat{R}/S)$.*

The parallel version of Feige's algorithm can easily be adapted to *classical streaming* if $k = O(s/\log n)$, where s is the size of the working memory in bits. In this case $O(1)$ passes suffice to simulate one step of the $O(k)$ simultaneous random walks: we keep in internal memory the original set of landmarks and the current frontier node for each random walk. Then, in one pass for each frontier node the number of neighbors is determined, and in a second pass one of the neighbors is chosen with correct probability as the successor of each frontier node. The initial selection of landmarks can be implemented as follows. Assuming that the number of nodes n is known in advance, that the nodes are numbered with integers in $[0, n-1]$, and that the graph is given as a atream of edges, $O((n \log n)/s)$ passes suffice to assign expected k landmarks with the required distribution: in the first pass we calculate the degree of the $O((n \log n)/s)$ vertices with lowest id, then toss appropriately biased coins to determine if any of these vertices are landmarks; by repeating the process for all vertices, the expected number of landmarks is obtained.

Therefore, by Theorem 3.13, if the graph is sparse (i.e., $m = O(n)$), then $\widehat{R} = O(n)$, and for a space restriction of $s \leq \widehat{R} \log n$ bits there is a *Stream* algorithm for USTCON that runs in $p = \widetilde{O}(n^2/s^2)$ passes. Hence, if for example $s = O(n^{3/4} \log n)$ then the streaming version of Feige's algorithm will run in $\widetilde{O}(n^{1/2})$ passes, and we achieve both sublinear space and passes for USTCON in *Stream*.

As we will see in Chapter 4, the idea of simulating parallel algorithms in a streaming setting (see Section 2.3.3), when applied to the *W-Stream* model, will lead to general reductions to parallel algorithms, and to efficient tradeoff algorithms for a variety of combinatorial problems, including *sorting*, *biconnected components* and *maximal independent set*.

3.7 Conclusions

In this chapter, we have presented algorithms for fundamental graph problems such as *connected components*, *minimum spanning tree* and *shortest paths*, in the *W-Stream* model. Our algorithms adapt to the amount of working memory available, allowing a smooth tradeoff between memory requirements and number of passes over the data stream.

We have also proven several separation results between *Stream* and *W-Stream*, as well as some hardness results for the *W-Stream* model. Most notably, we have shown that some communication complexity-based techniques, commonly used for obtaining lower bounds in *classical streaming*, can be adapted to *W-Stream* as well, yielding proof that many of our tradeoff algorithms (some presented in this chapter, such as *connected components* and *minimum spanning tree*, and others presented in Chapter 4, such as *sorting*, *biconnected components* and *maximal independent set*) are optimal up to a polylogarithmic factor.

Finally, we have shown that memory/passes tradeoffs are possible even in the more restrictive *classical streaming* model, for a simpler specialized variant of the *reachability* problem in directed graphs, and with some limitations also for a fundamental problem such as *undirected s-t connectivity.*

Chapter 4

Reductions to Parallel Algorithms

4.1 Introduction

General reductions to parallel algorithms exist in both *external memory* and *StreamSort* (see Section 2.3.3), and these reductions have either produced directly, or inspired, many efficient algorithms for fundamental combinatorial problems in these models. We remark, however, that both models are significantly more powerful than *W-Stream*.

In this chapter, we will show how classical parallel algorithms designed in the PRAM model can be turned into near-optimal algorithms for several classical combinatorial problems, even in the less powerful *W-Stream* model.

We will first show that any PRAM algorithm that runs in time T using N processors and memory M can be simulated in *W-Stream* using $p = O((T \cdot N \cdot \log M)/s)$ passes. This yields near-optimal tradeoff upper bounds of the form $p = O((n \cdot \text{polylog } n)/s)$ in *W-Stream* for several problems, where n is the input size (*sorting* is a relevant example). For other problems, however, this simulation does not provide good upper bounds. One prominent example concerns graph problems, for which efficient PRAM algorithms typically require $O(m+n)$ processors on graphs with n vertices and m edges. For those problems, this simulation method yields $p = O((m \cdot \text{polylog } n)/s)$ bounds, while $p = \Omega(n/s)$ almost-tight lower bounds in *W-Stream* are known for many of them.

To overcome this problem, we study an intermediate parallel model, which we call RPRAM, derived from the PRAM model by relaxing the assumption that a processor can only access a constant number of cells at each round. This way, we get the PRAM algorithms closer to streaming algorithms, since a memory cell in the working memory can be processed against an arbitrary number of cells in the stream. For some problems, this enhancement allows us to substantially reduce the number of processors while maintaining the same number of rounds. We show that simulating RPRAM algorithms in *W-Stream* leads to near-optimal algorithms (up to polylogarithmic factors) for several fundamental problems, including *sorting*, *minimum spanning tree*, *biconnected components*, and *maximal independent set*. We remark that algorithms obtained in this way are not always optimal, although very close to being so. For some of them better *ad hoc* algorithms designed directly in *W-Stream*, without using simulations, exist (e.g., the *connected components* algorithm of Section 3.2 and the *minimum spanning tree* algorithm of Section 3.3). Other bounds can be improved by maintaining the algorithm's overall structure, and only replacing some of its simulated steps with direct *W-Stream* implementation (this is the case of the *biconnected components* algorithm described in Section 4.6), while for others no better direct *W-Stream* implementations are known (e.g., for *maximal independent set*).

Finally, we show that there exist problems for which the increased computational power of the RPRAM model does not help in reducing the number of processors required by a PRAM algorithm while maintaining the same time bounds, and thus cannot lead to better *W-Stream* algorithms. An example is deciding whether a directed graph contains a cycle of length two.

The chapter is organized as follows. Section 4.2 will present the general PRAM and RPRAM simulation techniques. Sections 4.3 through 4.7 will will show that these techniques yield near-optimal *W-Stream* algorithms for a variety of classical combinatorial problems (*sorting*, *connected components*, *minimum spanning tree*, *biconnected components* and *maximal independent set*, respectively). In Section 4.8 we will show that there are problems for which the increased power of the RPRAM model does not lead to improved *W-Stream* bounds. Finally, Section 4.9 will briefly summarize the results achieved in this chapter.

4.2 Simulating Parallel Algorithms in W-Stream

In this section we will introduce general techniques for simulating parallel algorithms in *W-Stream*. In Subsection 4.2.1 we will discuss how to simulate general CRCW PRAM algorithms, while Subsection 4.2.2 will deal with the simulation of a *relaxed* PRAM model (namely, RPRAM), which often leads to better *W-Stream* bounds, most notably when dealing with graph problems.

In the following, we will assume that each memory address, cell value, and processor state can be stored using $O(\log M)$ bits, where M is the memory size of the parallel machine.

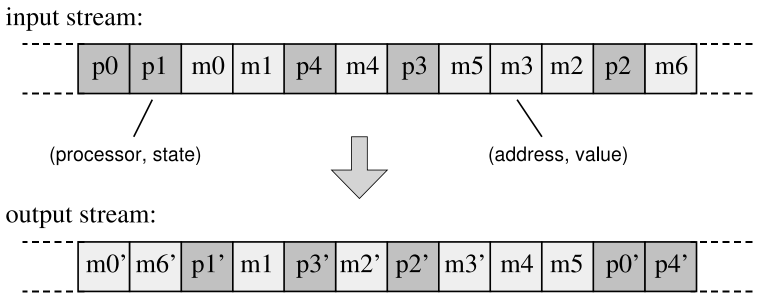


Figure 4.1: Simulating a PRAM algorithm in *W-Stream*. Processor states and memory cells are updated as needed, and stored on the output stream; unchanged memory cells are just propagated by copying them from input stream to output stream.

4.2.1 PRAM Simulation

A generic PRAM algorithm can be simulated in *W-Stream* within the following bounds.

Theorem 4.1 *Let A be a* PRAM *algorithm that uses N processors and runs in time T using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof. In the PRAM model, at each parallel round, every processor may read $O(1)$ memory cells, perform $O(1)$ instructions to update its internal state, and write $O(1)$ memory cells. A round of A can be simulated in *W-Stream* by performing $O((N \log M)/s)$ passes, whereas $O(1)$ passes suffice to simulate the execution of $\Theta(s/\log M)$ processors using s bits of working memory. The content of the memory cells accessed by the algorithm and the state of each processor are maintained on the intermediate streams as items of the form $(address, value)$ and $(processor, state)$, respectively. We simulate the task of each processor in a constant number of passes as follows. We first read from the input stream its state and the content of the $O(1)$ memory cells used by A and then we execute the $O(1)$ instructions performed. Finally, we write to the output stream the new state and possibly the values of the $O(1)$ output cells. Memory cells that remain unchanged are simply propagated through the intermediate streams by just copying them from the input stream to the output stream at each pass. $\square$

Applications. By Theorem 4.1, tradeoff algorithms can be obtained for a number of problems that, not only are interesting in their own right, but will also prove valuable tools in the following sections, in designing efficient *W-Stream* algorithms for graph problems.
List ranking, for instance, is the following problem:

Definition 4.1 (List ranking) *Given a linked list L, the* list ranking *problem consists in determining how distant each item is from the bottom of L.*

List ranking can be solved in PRAM within the following bounds.

Theorem 4.2 ([10]) List ranking *can be solved on a PRAM using $O(n/\log n)$ processors in $O(\log n)$ parallel rounds, for a list with n items.*

Therefore, by Theorem 4.1 we obtain the following result:

Corollary 4.1 *In* W-Stream*,* list ranking *can be solved within $O((n \log n)/s)$ passes (where n is the number of items in the list), for a space restriction of s bits.*

An *Euler tour of a tree* is defined as follows.

Definition 4.2 (Euler tour of a tree) *Given a tree $T = (V, E)$, an* Euler tour *of T is a visit of T that traverses each edge in E exactly twice.*

An *Euler tour of a tree* can be computed by a PRAM within the following bounds.

Theorem 4.3 ([77]) *An* Euler tour of a tree *with n vertices can be computed by a PRAM with $O(n)$ processors in $O(1)$ parallel rounds.*

And, by Theorem 4.1, the following corollary results:

Corollary 4.2 *In* W-Stream*, an* Euler tour of a tree *with n verices can be found in $O((n \log n)/s)$ passes, for a space restriction of s bits.*

Another useful tool will be *sorting*, which can be solved by Theorem 4.1 near-optimally in *W-Stream* in $O((n \cdot \text{polylog } n)/s)$ passes (see Section 4.3 for a full discussion on this problem).

Finally, we remark that, even though Theorem 4.1 can yield near optimal *W-Stream* algorithms for some problems (e.g., *sorting*), for others the bounds obtained this way are far from being optimal. For instance, efficient PRAM algorithms for graph problems typically require $O(m+n)$ processors, where n is the number of vertices, and m is the number of edges. For these problems, Theorem 4.1 yields bounds of the form $p = O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds are known for many of them.

4.2.2 RPRAM Simulation

In Definition 4.3 we will introduce RPRAM as an extension of the PRAM model. It allows every processor to handle in a parallel round not only $O(1)$ memory cells, but an arbitrary number of cells. Since in *W-Stream* a value in the working memory might be processed against all the data in the stream, we view RPRAM as a natural link between PRAM and *W-Stream*, even though it may be unrealistic in a practical setting. We will first introduce a generic simulation that turns RPRAM algorithms into *W-Stream* algorithms and then, in the following sections, provide RPRAM implementations that lead to efficient algorithms in *W-Stream* for a number of problems where the PRAM simulation in Theorem 4.1 does not yield good results.

Definition 4.3 (RPRAM) *An* RPRAM *(Relaxed* PRAM*) is an extended CRCW* PRAM *machine with N processors and memory of size M where at each round each processor can execute $O(M)$ instructions that:*

- *can read an arbitrary number of memory cells. Each cell can only be read a constant number of times by each processor during the round, and no assumptions can be made as to the order in which values are given to the processor;*
- *can write an arbitrary subset of the memory cells. The result of concurrent writes to the same cell by different processors in the same round is undefined. Writing can only be performed after all read operations have been done.*

Similarly to a PRAM*, each processor has a constant number of registers of size $O(\log M)$ bits.*

The jump in computational power provided by RPRAM allows substantial improvements for many classical PRAM algorithms such as decreasing the number of parallel rounds while preserving the number of processors or reducing the number of processors used while maintaining the same number of parallel rounds. We show in Theorem 4.4 that parallel algorithms implemented in this more powerful model can be simulated in *W-Stream* within the same bounds of Theorem 4.1.

Theorem 4.4 *Let A be an* RPRAM *algorithm that uses N processors and runs in time T using space $M = \mathrm{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof. We follow the proof of Theorem 4.1. The main difference is that a processor in the RPRAM model can read and write an arbitrary number of memory cells at each round, executing many instructions while still using $O(\log M)$ bits to maintain its internal state. Since the instructions of algorithm A performed by a processor during a round do not assume any particular order for reading the memory cells, reading memory values from the input stream can still be simulated in one pass. Replacing cell values read from the input stream with the new values written on the output stream can also be performed in one additional pass. □

In the remainder of this chapter, we will show that parallel algorithms for several classical problems can be naturally implemented in the RPRAM model, yielding by Theorem 4.4 efficient algorithms in *W-Stream*.

4.3 Sorting

As a first application of the simulation techniques introduced in Section 4.2, we show how to derive efficient sorting algorithms in *W-Stream*. We first recall that n items can be sorted on a PRAM with $O(n)$ processors in $O(\log n)$ parallel rounds and $O(n \log n)$ comparisons [77]. By Theorem 4.1, this yields a *W-Stream* sorting algorithm that runs in $p = O((n \log^2 n)/s)$ passes. In RPRAM, however, *sorting* can be solved by $O(n)$ processors in constant time as follows.

Theorem 4.5 *Sorting n items in* RPRAM *can be done in $O(1)$ parallel rounds using $O(n)$ processors.*

Proof. Each processor is assigned to an input item; in one parallel round it scans the entire memory and counts the numbers i and j of items smaller than and equal to the item the processor is assigned to, respectively. Then each processor writes its own item into the cells with indices between $i + 1$ and $i + 1 + j$, and thus we obtain a sorted sequence. □

Using the simulation in Theorem 4.4, we obtain the result stated below.

Corollary 4.3 *Sorting n items in* W-Stream *can be performed in $O(n \log n/s)$ passes.*

We obtain a *W-Stream* sorting algorithm that takes $p = O((n \log n)/s)$ passes, thus matching the performance of the best known algorithm for *sorting* in a streaming setting [87]. Since *sorting* requires $p = \Omega(n/s)$ passes in *W-Stream*, this bound is essentially optimal.

Theorem 4.6 *In* W-Stream*, sorting n items requires $p = \Omega(n/s)$ passes when the space restriction is s bits.*

Proof. It suffices to observe that *element distinctness* (see Definition 3.6) can be reduced to *sorting* plus one extra pass to detect duplicates in the sorted sequence. The result then follows from the communication complexity-based lower bound for *element distincness* (see Theorem 3.9). □

We note, however, that both our algorithm and the algorithm in [87] perform $O(n^2)$ comparisons. We reduce the number of comparisons to the optimal $O(n \log n)$ at the expense of increasing the number of passes to $O((n \log^2 n)/s)$ by simulating an optimal PRAM algorithm via Theorem 4.1, as stated before.

4.4 Connected Components

The *connected components* problem has already been formally introduced in Section 3.2 (see Definition 3.1). In this section, we will first describe a classical PRAM random-mating algorithm for computing the *connected components* of a graph $G = (V, E)$ (with $|V| = n$ and $|E| = m$) which uses $O(m + n)$ processors and runs in $O(\log n)$ time with high probability [24, 94], and then we will give an RPRAM implementation that uses only $O(n)$ processors which, by Theorem 4.4, leads to a nearly optimal algorithm in *W-Stream*.

4.4.1 PRAM Algorithm

The random-mating technique for graph contraction is based on building a set of star subgraphs and contracting the stars (a *star* is a tree of depth one, consisting of a root, or *parent*, and an arbitrary number of leaves, or *children*). The parallel algorithm for *connected components* recursively finds a set of non-overlapping stars, and contracts each star into a single vertex by merging the children into their parents. It stops when there are no more edges left. At each parallel round the algorithm performs the following sequence of steps:

1. Each vertex is assigned the status of parent or child independently with probability 1/2.

2. For each child vertex u, determine whether it has any adjacent parent vertices. If so, choose one of such vertices arbitrarily to be the parent $f(u)$ of u, and replace each edge (u, v) by $(f(u), v)$ and each edge (v, u) by $(v, f(u))$.

3. For each vertex having parent u, set the parent to $f(u)$.

A probabilistic analysis reveals that the algorithm is expected to reduce the number of active vertices by a constant fraction at each iteration, requiring $O(\log n)$ parallel rounds, with high probability, to run [24].

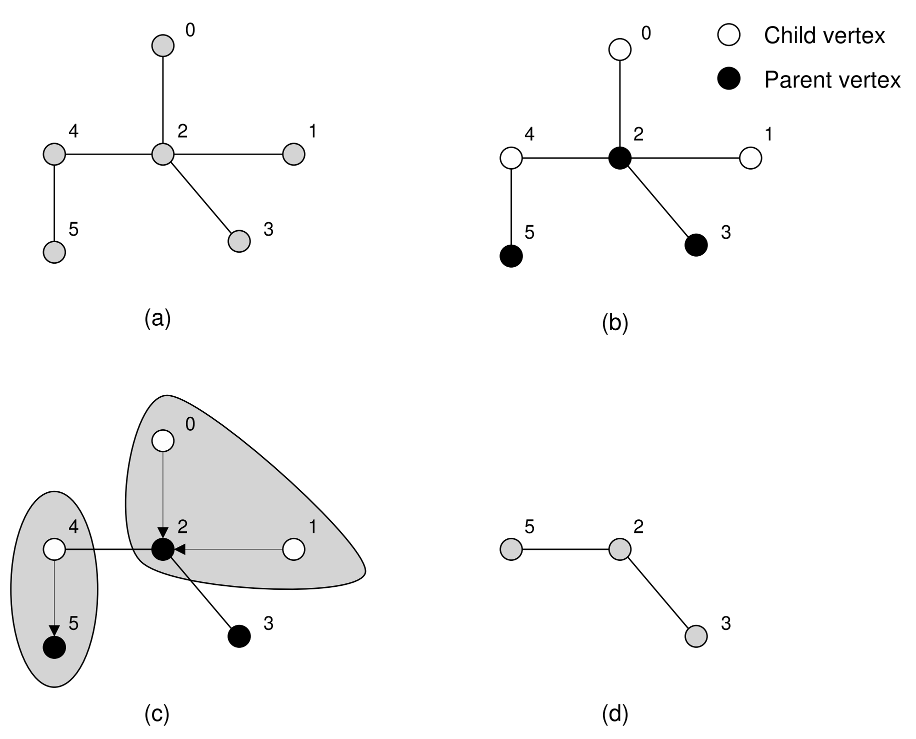


Figure 4.2: One step of the *random-mating* technique. (a) The initial graph. (b) Each vertex is assigned randomly the status of *parent* or *child*. (c) The contraction phase. (d) The resulting graph.

4.4.2 RPRAM Implementation

We now show how to implement each iteration of the above algorithm in RPRAM in $O(1)$ rounds using only $O(n)$ processors. We attach a processor to each vertex. The RPRAM algorithm then proceeds as follows:

1. Each vertex assigns itself the status of parent or child independently with probability 1/2.
2. Each child vertex scans its neighborhood to find a parent, if there exists one (in case of several parents, one is chosen arbitrarily), and updates the incident edges accordingly.
3. For each vertex having parent u, set its parent to $f(u)$.

Since an RPRAM processor can access an arbitrary number of cells, and therefore scan its entire neighborhood, at each round, it is easy to see that an iteration of the algorithm takes only $O(1)$ rounds. By remembering there are $O(\log n)$ such iterations with high probability, we obtain the result in Theorem 4.7.

Theorem 4.7 *Solving CC in* RPRAM *takes $O(n)$ processors and $O(\log n)$ rounds with high probability.*

By Theorem 4.4, this yields the following bound in *W-Stream*.

Corollary 4.4 *CC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

By the $p = \Omega(n/s)$ lower bound for CC in *W-Stream* (see Theorem 3.1), this upper bound is optimal up to a polylogarithmic factor. We notice that the same bound can be achieved deteministically by starting from the PRAM algorithm for CC in [99]. Moreover, this bound can be further improved to $O((n \log n)/s)$ passes by an *ad hoc W-Stream* algorithm, as shown in Section 3.1.

4.5 Minimum Spanning Tree

In this section, we will first describe the PRAM algorithm in [24] for computing a *minimum spanning tree* of an undirected graph, and then we will give an RPRAM implementation that leads to an optimal algorithm (up to a polylogarithmic factor) in *W-Stream* by using the simulation in Theorem 4.4. Please note that a formal definition of the *minimum spanning tree* of a graph has already been given in Section 3.3 (see Definition 3.2).

4.5.1 PRAM Algorithm

The algorithm is obtained by a modification of the random-mating CC algorithm presented in Section 4.4, to find a *minimum spanning tree* of a connected weighted undirected graph [24] (or a minimum spanning forest if the input graph is not connected). It is based on the property that given a subset V' of vertices, a minimum weight edge having one and only one endpoint in V' is in some MST. It takes $O(\log n)$ rounds with high probability and uses $O(m + n)$ processors. We will now describe the operations performed at each iteration. In the following, we will denote the parent vertex of a vertex u as $f(u)$. Notice that only Step 2 has changed from the CC algorithm.

1. Each vertex is assigned the status of parent or child independently with probability 1/2.

2. For each child vertex u, determine the minimum weight incident edge (u, v). If v is a parent vertex, then set $f(u) = v$, flag the edge (u, v) as belonging to the spanning tree, and replace each edge (u, w) by $(f(u), w)$ and each edge (w, u) by $(w, f(u))$.
3. For each vertex having parent u, set the parent to $f(u)$.

The algorithm terminates when there are no more edges left, and can be proven to solve the MST problem in $O(\log n)$ parallel rounds with high probability [24].

4.5.2 RPRAM Implementation

The three steps of the PRAM algorithm can be implemented on an RPRAM with $O(n)$ processors (each assigned to a vertex), as follows.

1. Each vertex assigns itself the status of parent or child independently with probability 1/2.
2. Each child vertex scans its neighborhood to find the adjacent parent vertex of minimum weight, if there exists one, and updates the incident edges accordingly.
3. For each vertex having parent u, set its parent to $f(u)$.

All the above steps can be executed in $O(1)$ rounds by an RPRAM with $O(n)$ processors, since each one of them can access an arbitrary number of memory cells, and therefore scan the entire neighborhood of the vertex it is assigned to, in a single round. Hence, we obtain the result stated in Theorem 4.8.

Theorem 4.8 *MST can be solved in* RPRAM *using $O(n)$ processors and $O(\log n)$ rounds with high probability.*

Assuming that edge weights can be encoded using $O(\log n)$ bits, we obtain the following bound in *W-Stream* by Theorem 4.4.

Corollary 4.5 *MST can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes.*

By the $p = \Omega(n/s)$ lower bound for MST in *W-Stream* (see Theorem 3.3), this upper bound is optimal up to a polylogarithmic factor. We note, however, that the bound can be further improved to $O((n \log n)/s)$ passes, by the deterministic *ad hoc W-Stream* algorithm presented in Section 3.3.

4.6 Biconnected Components

The *biconnected components* problem can be defined formally as follows:

Definition 4.4 (Biconnected Components) *Given an undirected graph $G = (V, E)$ with n nodes and m edges,* biconnected components (BCC) *is the problem of determining a function $c : V \to \{0, 1, \ldots, n-1\}$ such that $\forall u, v \in V$ $c(u) = c(v)$ if and only if u and v are biconnected in G. Two vertices of a graph are said to be biconnected if they cannot be disconnected by removing one vertex (and all the edges incident to it) in G.*

Tarjan and Vishkin [101] gave a PRAM algorithm that computes the *biconnected components* of an undirected graph in $O(\log n)$ time using $O(m+n)$ processors. We will first give an overview of their algorithm, then provide an RPRAM implementation that uses only $O(n)$ processors while preserving the time bounds, and thus can be turned using Theorem 4.4 in a *W-Stream* algorithm that runs in $O((n \log^2 n)/s)$ passes. Finally, we will show that by reimplementing some steps of the algorithm directly in *W-Stream*, without using any simulations, the number of passes can be further reduced to $O((n \log n)/s)$.

4.6.1 PRAM Algorithm

Given a graph G, the algorithm considers a graph G' such that vertices in G' correspond to edges in G and connected components in G' correspond to biconnected components in G. The algorithm first computes a rooted spanning tree T of G and then builds a subgraph G'' of G' having as vertices all the edges of T. The edges of G'' are chosen such that two vertices are in the same connected component of G'' if and only if the corresponding edges in G are in the same biconnected component. After computing the connected components of G'' the algorithm appends the remaining edges of G to their corresponding biconnected components. We now briefly sketch the five steps of the algorithm.

1. Build a rooted spanning tree T of G and compute for each vertex its preorder and postorder numbers together with the number of descendants. Also, label the vertices by their preorder numbers.

2. For each vertex u, compute two values, $low(u)$ and $high(u)$, as follows.

$$\begin{aligned} low(u) &= \min(\{u\} \cup \{low(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\}) \\ high(u) &= \max(\{u\} \cup \{high(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\}), \end{aligned}$$

   where $p(u)$ denotes the parent of vertex u. Hence, $low(u)$ denotes the lowest vertex that is either a descendant of u or adjacent to a descendant of u by an edge of $G \setminus T$, and analogously $high(u)$ denotes the highest vertex that is either a descendant of u or adjacent to a descendant of u by an edge of $G \setminus T$.

3. Add edges to G'' according to the following two rules. For all edges $(w, v) \in G \setminus T$ with $v + desc(v) \leq w$, add $((p(v), v), (p(w), w))$ to G'', and for all $(v, w) \in T$ with $p(w) = v$, $v \neq 1$, add $((p(v), v), (v, w))$ to G'' if $low(w) < v$ or $high(w) \geq v + desc(v)$, where $desc(v)$ denotes the number of descendants of vertex v.

4. Compute the connected components of G''.

5. Add the remaining edges of G to their biconnected components. Each edge $(v, w) \in G \setminus T$, with $v < w$, is assigned to the biconnected component of $(p(w), w)$.

4.6.2 RPRAM Implementation

We will now provide an RPRAM implementation of the five steps of the above algorithm that uses $O(\log n)$ parallel and $O(n)$ processors.

1. A spanning tree of the input graph can be computed using the RPRAM algorithm introduced in Section 4.5. Rooting the tree and computing for each vertex the preorder and postorder numbers as well as the number of descendants are performed using *list ranking*, *Euler tour of a tree* and *sorting* [101]. These operations take $O(\log n)$ time and $O(n)$ processors in PRAM, and thus also in RPRAM.

2. Since the second step takes $O(\log n)$ time using $O(n)$ processors in PRAM [101], the same bounds hold for RPRAM.

3. $O(n)$ processors and constant time suffice to implement this step in RPRAM, since each vertex can scan its entire neighborhood in a single parallel round.

4. For computing the connected components of G'', we may use the RPRAM algorithm introduced in Section 4.4 that takes $O(\log n)$ time and $O(n)$ processors.

5. Finally, we implement the last step of the algorithm in RPRAM in $O(1)$ time and $O(n)$ processors by scanning the neighborhood for all vertices and assigning the incident edges to the proper biconnected components.

Since we have implemented all the steps of the algorithm in RPRAM in $O(\log n)$ rounds and $O(n)$ processors, we obtain the following result (we remind that the both the CC algorithm of Section 4.4 and the MST algorithm of Section 4.5 take $O(\log n)$ rounds *with high probability*).

Theorem 4.9 *BCC can be solved in* RPRAM *using $O(n)$ processors in $O(\log n)$ rounds with high probability.*

By Theorem 4.4, this yields the following bound in *W-Stream*.

Corollary 4.6 *BCC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

We now show that better bounds can be achieved by maintaining the overall structure of the algorithm, but implementing some of its steps directly in *W-Stream*, without using any simulations.

4.6.3 A Faster W-Stream Implementation

We will now describe how to implement directly in *W-Stream* some of the steps of the parallel algorithm of Tarjan and Vishkin [101]. Notice that we have given constant time and $O(n)$ processors RPRAM descriptions for the third and the fifth step, thus by applying the simulation in Theorem 4.4 we obtain *W-Stream* implementations that run in $O((n \log n)/s)$ passes for these steps. For computing the *connected components* in the fourth step, we may use the algorithm of Section 3.2, that is deterministic and requires $O((n \log n)/s)$ passes. Therefore, to achieve a global bound of $O((n \log n)/s)$ passes, it suffices to give implementations that run in $O((n \log n)/s)$ passes for the first two steps. For the first step, we can compute a *spanning tree* with the algorithm described in Section 3.3 (notice that this too is a deterministic result). Rooting the tree and computing the preorder and postorder numbers together with the number of descendants can still be implemented in $O((n \log n)/s)$ passes using *list ranking*, *Euler tour of a tree* and *sorting*. Concerning the second step, the *low* and *high* values can easily be computed by processing $\Theta(s/\log n)$ vertices at each pass, according to the postorder numbers. Hence, the following theorem:

Theorem 4.10 *BCC can be solved in* W-Stream *in $O((n \log n)/s)$ passes in the worst case.*

We will now show that both the upper bounds of Corollary 4.6 and Theorem 4.10 are optimal up to a polylogarithmic factor.

Theorem 4.11 *Let $G = (V, E)$ be an undirected graph, with n vertices and m edges. Solving BCC on G requires $p = \Omega(n/s)$ passes in* W-Stream*, for a space restriction of s bits.*

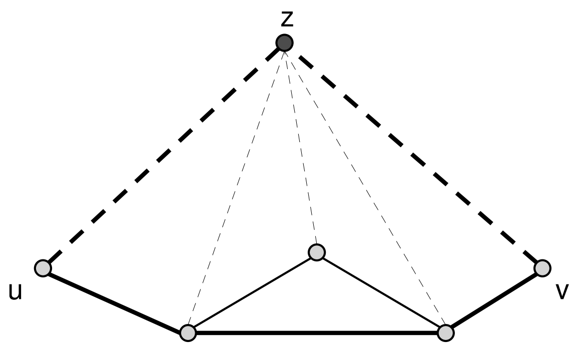


Figure 4.3: Reducing *connectivity* to *biconnectivity*: every two vertices u and v will be connected by two or more vertex-disjoint paths, if and only if the original graph was connected.

Proof. We first observe that *biconnectivity* (i.e., the decision problem of whether any two vertices in a graph are biconnected) can be reduced to *biconnected components*. *Graph connectivity* in turn can be reduced to *biconnectivity* in the following way. Add an extra vertex z and connect it to every vertex in V. The resulting graph G' will be biconnected if and only if the original graph was connected (between any two vertices $u, v \in V$ there will be at least two vertex disjoint paths: one in the original graph and the newly introduced path $\{(u,z),(z,v)\}$. The result then follows from the communication complexity-based lower bound for *connectivity* in *W-Stream* (see Theorem 3.10). $\square$

4.7 Maximal Independent Set

Given an undirected graph G, a *maximal independent set* is defined as follows.

Definition 4.5 (Maximal independent set) *Given an undirected graph $G = (V,E)$, an independent set is a subsetset $S \subseteq V$, such that for any two vertices in S, there is no edge in E connecting them. An independent set S is a* maximal independent set (MIS) *if it is such that every edge in E has at least one endpoint not in S, and every vertex not in S has at least one neighbor in S.*

We will now describe an efficient RPRAM algorithm for the *maximal independent set* problem, based on the PRAM algorithm proposed by Luby [83]. Using the simulation in Theorem 4.4, this leads to an efficient *W-Stream* implementation.

4.7.1 PRAM Algorithm

A *maximal independent set* S of a graph G is incrementally built through a series of iterations, where each iteration consists of the following three steps:

1. Compute a random subset I of the vertices in G, by including each vertex v with probability $1/(2 \cdot deg(v))$.

2. For each edge (u, v) in G, with $u, v \in I$, remove from I the vertex with the smallest degree.

3. Add to S the vertices in I, and then remove from G the vertices in I together with their neighbors.

The above steps are iterated until G gets empty. The algorithm uses $O(m+n)$ processors and $O(\log n)$ parallel rounds [83].

4.7.2 RPRAM Implementation

The three steps of the PRAM algorithm can be implemented on an RPRAM with $O(n)$ processors (each assigned to a vertex of the input graph) as follows:

1. Let each vertex compute its own degree, and select itself in I with the correct probability.

2. Let each vertex in I scan its neighborhood, and remove itself upon encountering a neighbor also in I with a larger degree.

3. Let each vertex not in I scan its neighborhood, and remove itself from G if at least one of its neighbors is in I.

Clearly, all three steps take constant time to run, since any RPRAM processor can access an arbitrary amount of memory cells at each round, and therefore scan its entire neighborhood in just one parallel round. Since the algorithm performs $O(\log n)$ iterations with high probability [83], by Theorem 4.4 we get the following result:

Theorem 4.12 *A* maximal independent set *can be found in* RPRAM *using $O(n)$ processors in $O(\log n)$ rounds with high probability.*

And, by Theorem 4.4, this yields the following upper bound in *W-Stream*.

Corollary 4.7 *A* maximal independent set *can be found in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

We will now show that the bound in Corollary 4.7 is optimal up to a polylogarithmic factor.

Theorem 4.13 *Finding a* maximal independent set *requires $\Omega(n/s)$ passes in* W-Stream *when the space restriction is s bits.*

Proof. The proof is based on a reduction from the bit vector disjointness communication complexity problem (see Section 2.3.1). The reduction proceeds as follows. A and B build a graph on $4n$ vertices v_i^j, where $i = 1, \cdots, n$ and $j = 1, \cdots, 4$. If $x_i = 0$, then A adds edges (v_i^1, v_i^2) and (v_i^3, v_i^4), whereas if $y_i = 0$, then Bob adds edges (v_i^1, v_i^3) and (v_i^2, v_i^4). The size of any MIS is $2n$ if $A \cdot B = 0$ and strictly greater otherwise. Since solving *bit vector disjointness* requires transmitting $\Omega(n)$ bits [80], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from A to B at each pass (see the proof of Theorem 3.9 for more details), we obtain $s = \Omega(n)$, which leads to $p = \Omega(n/s)$ lower bound. □

Finally, we remark that, to the best of our knowledge, no better *ad hoc W-Stream* algorithm is known for this problem.

4.8 Limits of the RPRAM Approach

In this section we prove that the increased power that RPRAM provides does not always help in reducing the number of processors from $O(m+n)$ to $O(n)$ in graph problems, and thus in obtaining *W-Stream* algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes. As an example, in Theorem 4.14 we prove that detecting cycles of length two in a directed graph takes $\Omega(m/s)$ passes.

Theorem 4.14 *Testing whether a directed graph with m edges contains a cycle of length two requires $p = \Omega(m/s)$ passes in* W-Stream.

Proof. As usual, we prove the lower bound by showing a reduction from the *bit vector disjointness* two-party communication complexity problem (see Section 2.3.1). The reduction proceeds as follows. A creates a stream containing an edge $e(i) = (x_i, y_i)$ for each i such that $x_i = 1$ and B creates a stream containing an edge $e^r(i) = (y_i, x_i)$ for each i such that $y_i = 1$, where $x_i = i \text{ div } \lceil\sqrt{m}\,\rceil$ and $y_i = i \bmod \lceil\sqrt{m}\,\rceil$. Let G be the directed graph induced by the union of the edges in the streams created by A and B. Clearly, there is a cycle of length two in G if and only if $A \cdot B > 0$. Since solving *bit vector disjointness* requires transmitting $\Omega(m)$ bits [80], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from A to B at each pass (see the proof of Theorem 3.9 for more details), we obtain $s = \Omega(m)$, which leads to $p = \Omega(m/s)$. □

Theorem 4.14 implies that testing whether a directed graph has a cycle of length two requires $\Omega(m/(n \log n))$ rounds on an RPRAM with n processors. On the other hand, this problem can easily be solved in one round on a PRAM using $O(m + n)$ processors, by just checking in parallel whether there is any edge (x, y) that also appears as (y, x) in the graph. This leads to an algorithm in *W-Stream* that runs in $O((m \log n)/s)$ passes by Theorem 4.1.

4.9 Conclusions

In this chapter, we have introduced a general technique for simulating parallel algorithms in the *W-Stream* model. We have shown that this simulation yields near optimal tradeoff algorithms for some problems, such as *sorting*, while it leads to suboptimal results for others. Most notably, this is the case for graph problems, that in many cases require $O(n + m)$ processors to be solved efficiently, leading to *W-Stream* simulations that run in $O((m \cdot \text{polylog } n)/s)$ passes, while near-optimal direct *W-Stream* algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes are known for many of them (for example, the *connected components* and *minimum spanning tree* algorithms presented in Chapter 3).

To overcome this problem, we have introduced an intermediate model, called RPRAM, that while being more powerful than the traditional PRAM model, can be simulated in *W-Stream* within the same asymptotic bounds. Adapting PRAM algorithms to RPRAM usually allows us to decrease the number of processors from $O(n + m)$ to $O(n)$, yielding *W-Stream* algorithms for many graph problems (e.g., *connected components*, *minimum spanning tree*, *biconnected components* and *maximal independent set*) that are optimal up to a polylogarithmic factor.

Finally, we have shown that there are nonetheless problems for which the extra power provided by the RPRAM model does not help in reducing the number of processors, and therefore in obtaining more eficient *W-Stream* algorithms. One example is detecting whether a directed graph contains a cycle of length 2, for which a communication complexity-based lower bound of $p = \Omega(m/s)$ has been proven to exist.

Chapter 5

Engineering Streaming Algorithms

5.1 Introduction

In this chapter, we contribute to bridge the gap between theoretical and experimental work concerning streaming algorithms for graph problems, by presenting an experimental investigation of recent *Semi-streaming* algorithms for a fundamental graph problem such as *spanner* construction (certainly, from a theoretical point of view, the most extensively studied graph problem in a streaming setting).

Graph spanners arise in many applications, including communication networks, computational biology, computational geometry, distributed computing, and robotics ([9, 13, 16, 29, 30, 37, 42, 47, 81, 82, 92, 93, 95]). Intuitively, a spanner of a given graph is a smaller subgraph which preserves approximate distances between all pairs of vertices. More formally,

Definition 5.1 ((α, β)-spanner) *Let $G = (V, E)$ be a graph, and let α, β be real values, with $\alpha \geq 1$ and $\beta \geq 0$. An* (α, β)-spanner *of G is a graph $S = (V, E')$ with $E' \subseteq E$ such that the following holds:*

$$\forall\ u, v \in V \qquad dist_S(u, v) \leq \alpha \cdot dist_G(u, v) + \beta.$$

An $(\alpha, 0)$-spanner is called a *purely multiplicative spanner*, or more simply an α-spanner, while a $(1, \beta)$-spanner is called a *purely additive spanner*.

The construction of spanners is a long-studied problem, in both static and dynamic settings. In a static setting the best known algorithms are those of Baswana and Sen [21] (randomized) and Roditty *et al.* [96] (deterministic), that compute a purely multiplicative $(2k-1, 0)$-spanner of size $O(k \cdot n^{1+\frac{1}{k}})$ of an undirected weighted graph (with n vertices and m edges) in time $O(m + n)$.

In the dynamic setting, where edges may be added to or deleted from the original graph, a few algorithms for updating the spanner have been proposed, in the case of unweighted graphs. Most notably, a randomized algorithm has been proposed by Baswana [19], for maintaining a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+\frac{1}{k}})$ in $\widetilde{O}(\frac{m}{n^{1+1/k}})$ amortized expected time for each edge insertion/deletion.

We remark that, based on the girth conjecture by Erdös [49], Bollobás [25] and Bondy *et al.* [26], there are graphs on n vertices for which any $(2k-1)$-spanner (for any $k \in \mathbb{N}$) requires $\Omega(n^{1+1/k})$ edges (the conjecture has been proven for $k = 1, 2, 3$ and 5). Therefore, the above algorithms essentially match the conjectured worst case lower bound, in terms of their spanner size *versus* stretch factor tradeoff.

As we will see in Section 5.2, in recent years a lot of attention has also been devoted to the study of efficient algorithms for spanner construction in streaming settings, mostly in the *Semi-streaming* model for unweighted graphs [12, 20, 48, 54], but also in *StreamSort* for the more general case of weighted graphs [20].
In this chapter, we will compare the experimental performances of three *Semi-streaming* algorithms, that run in a single pass and produce spanners of unweighted graphs, both to evaluate their efficiency in practice and as a complement to the theoretical analysis, that often focuses on worst-case scenarios rather than typical input instances.

The first algorithm we will take into consideration, designed by Ausiello, Franciosa and Italiano [12], produces $(3, 2)$-spanners of the input graph and, in addition to being deterministic, it does not require any *a priori* knowledge of the input graph (i.e., number of nodes or edges). The second and third algorithms, proposed respectively by Baswana [20] and by Elkin [48], produce purely multiplicative $(2k-1)$-spanners, for arbitrary integer k; however, both are randomized, and both require knowledge of the number of nodes of the input graph, for their randomization schemes to be properly initialized. All the algorithms we have considered are natural choices, since they have the best theoretical bounds and are likely to outperform other algorithms also in practice, due to their simpler data structures. The three streaming algorithms will also be compared with an internal memory algorithm for unweighted graphs proposed by Zwick [105], as a benchmark.

The chapter is organized as follows. Section 5.2 will present an overview of the most significant results concerning spanners in the *Semi-streaming* model. We will also provide a detailed description of the three very recent algorithms for spanner construction, whose experimental performances we will be comparing in the following sections. Section 5.3 will describe Zwick's off-line algorithm, that we will use as a benchmark against which to evaluate the performances of the streaming algorithms. Section 5.4 features a detailed description of our experimental setup: hardware platform, programming language, test instances and performance measures. Section 5.5 will present the experimental results we have obtained, and draw some conclusions about the performances of each algorithm, while Section 5.6 will summarize the results obtained in this chapter.

5.2 Spanner Algorithms in Semi-streaming

In the last few years, several authors have address the problem of the computation of graph spanners in the *Semi-streaming* model (formally introduced in Section 2.2.2). In [54], Feigenbaum *et al.* presented a randomized *Semi-streaming* algorithm for computing a multiplicative $(2k+1)$-spanner of an unweighted graph in one pass. The spanner has expected size $O(k \cdot n^{1+1/k})$, each edge is processed in expected $O(k^2 \cdot n^{1/k} \log n)$ time, using $O(k \cdot n^{1+1/k} \log n)$ memory words.
A recent result by Baswana [20] improves on the size/stretch tradeoff: a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+1/k})$ can be computed in a single pass over the edges in overall $O(m+n)$ time ($O(1)$ amortized per edge processing time), using $O(k \cdot n^{1+1/k})$ memory space. However, both the algorithm in [20] and the one in [54], in order to successfully apply their randomization technique, need to know in advance the total number of vertices n in the graph.
Elkin [48] has proposed a new one-pass randomized algorithm, that yields $(2k-1)$-spanners (of unweighted graphs) of size $O((k \cdot \log n)^{1-1/k} \cdot n^{1+1/k})$ (w.h.p.), with $O(1)$ expected processing time per edge (but drastically improved worst-case processing time per edge over the previous results). Elkin's algorithm also requires prior knowledge of n.
Ausiello, Franciosa and Italiano [12] have also recently proposed an algorithm for computing $(3,2)$-spanners of unweighted graphs in one pass. Their algorithm is deterministic and does not need to know n in advance. It produces spanners of size $O(n^{4/3})$ edges in constant amortized per edge processing time, and requires overall $O(n^{4/3} \log n)$ bits of working memory.

In the rest of this section, we will describe in detail the algorithms by Ausiello, Franciosa and Italiano (`AFI`), by Baswana (`B`), and by Elkin (`E`) that we will compare experimentally in the following sections.

5.2.1 The Algorithm by Ausiello, Franciosa and Italiano (`AFI`)

The *Semi-streaming* algorithm proposed by Ausiello, Franciosa and Italiano (`AFI`) [12] produces a $(3,2)$-spanner of an unweighted graph in one pass over the input stream. The input graph is assumed to be represented as an arbitrary sequence of pairs (x,y), representing edge endpoints.

The central idea behind algorithm `AFI` is the formation of *clusters*. The algorithm proceeds as follows: as edges are streamed in, they are stored in main memory as *free edges*. Whenever the size of the neighborhood $N(v)$ of a vertex v, defined as $N(v) = \{v\} \cup \{x \mid (v, x) \in E\}$, goes beyond a certain threshold value, this vertex becomes the *center* of a cluster, and all its adjacent nodes are marked as belonging to the cluster whose center is v (which we will denote as $C(v)$). Clusters are therefore *stars* around a center vertex; notice, however, that vertex v is included in $C(v)$ only if it hasn't already been included in some other cluster. The advantage of this scheme is that inter-cluster edges can be dropped (except those connecting a vertex to its center), and only one *bridge* edge between any two clusters, if at least one exists, has to be kept in order to obtain a $(3, 2)$-spanner.

More formally, given an undirected graph $G = (V, E)$ and a *clustering* (i.e., a set of disjoint clusters) $\Gamma = \{C(x_1), C(x_2), \ldots, C(x_k)\}$, with $C(x_i) \subseteq N(x_i)$, a *CC-subgraph on* Γ is defined as the subgraph $S = (V, E')$ of G such that E' is the union of the following edge sets:

- **cluster edges:** all edges (x, y) such that x is a center and $y \in C(x)$;
- **CC-bridge edges:** for each pair of clusters C_i, C_j, with $i \neq j$, one arbitrary edge $(x, y) \in E$ (if one exists) such that (x, y) is not a cluster edge, $x \in C_i$ and $y \in C_j$. We say that edge (x, y) *connects* clusters C_i and C_j;
- **free edges:** all edges $(x, y) \in E$ such that at either x or y is a free vertex.

Then the following result has been proven by Ausiello, Franciosa and Italiano:

Theorem 5.1 ([12]) *Given a graph $G = (V, E)$ and a clustering Γ, any CC-subgraph $S = (V, E')$ on Γ is a (3,2)-spanner of G.*

Figure 5.1 features a detailed description of `AFI`'s main procedure (namely, `CC-EdgeProcess`). The algorithm consists of invoking the procedure for each edge of the input stream. The algorithm will maintain the following simple data structures. For each vertex x, it will store:

- a flag indicating whether x belongs or not to a cluster. In case it is clustered the algorithm will also store a label $Cl(x)$ (i.e., an identifier of the cluster that contains x) and a label *center*(x) (i.e., the center of cluster $Cl(x)$);
- a set of edges $F(x)$, that contains the set of free edges (x, y) leading to each free vertex $y \in N(x)$. Set $F(x)$ may also possibly include some edge (x, z) such that z is no longer free. The value $|F(x)|$ is also maintained;

- if x is the center of a cluster C, the list of vertices belonging to C.

Bridge edges are stored in a matrix *Bridge*, whose entry $Bridge[i, j]$ contains the bridge connecting clusters C_i and C_j. Obviously, $Bridge[i, j]$ is empty in the case $(C_i \times C_j) \cap E = \emptyset$, but it is also possible that $Bridge[i, j]$ is empty if there is some edge in a set $F(x)$ that connects C_i and C_j. The algorithm will create a cluster only if it will contain at least $n_c^{1/3}$ vertices, where n_c is the number of vertices currently known to the algorithm. This implies that clusters created in the first steps of the edge stream processing can be very small, while clusters created later must be larger.

In more details, procedure `CC-EdgeProcess` processes each edge (x, y) as follows. It first checks whether edge (x, y) connects two clusters that are not yet connected by a CC-bridge, and updates matrix *Bridge* accordingly (cf. Figure 5.1, lines 1–5). In case (x, y) joins a center x and a free vertex y, it adds y to $C(x)$ (lines 6–9). Otherwise, if y is free, (x, y) is added to set $F(x)$ of edges that connect x to (possibly) free vertices (lines 10–11). When this set becomes too large, $F(x)$ is scanned to determine whether the other endpoint of each edge is still free, and if not, whether the corresponding edge constitutes a bridge between two clusters (lines 12–17). If, after this process, at least $n_c^{1/3}$ free vertices remain, then a new cluster centered at x (including x itself if it is free), is created (lines 18–23).

We point out that when vertices are included in a new cluster, as in line 19 (resp. 21), procedure `CC-EdgeProcess` does not update set $F(y)$ for all $y \in N(z)$ (resp. for all $y \in N(x)$), for performance reasons. This implies that sets $F(\cdot)$ may contain edges whose endpoints are no longer free, and as a consequence, the condition of line 12 does not ensure that a new cluster centered on x can be created. The solution proposed by the authors is to clean up set $F(x)$, and count the effective number of free vertices in $N(x)$, only when $|F(x)|$ is large enough to amortize the cost of the operation (lines 12–17).

`AFI` is deterministic and it does not need to know the number of nodes or edges of the input graph in advance. As proven by Ausiello, Franciosa and Italiano in [12], it computes a $(3, 2)$-spanner of size $O(n^{4/3})$ edges in $O(1)$ per edge amortized processing time ($O(n^{1/3})$ in the worst case). It requires overall $O(n^{4/3} \log n)$ working memory (in bits).

Implementation Details.

In our implementation, we modified the original algorithm of Figure 5.1 by parameterizing it with a user-defined *clustering threshold* c, that replaces the adaptive $n_c^{1/3}$ value used by the algorithm as a threshold to form clusters (where n_c is the number of vertices currently known to the algorithm). As we

```
    Procedure CC-EdgeProcess
    input: edge $(x, y)$
1.      if both $x$ and $y$ are clustered
2.          if $Cl(x) \neq Cl(y)$ and there is not a bridge connecting $Cl(x)$ and $Cl(y)$
3.              store $(x, y)$ into $Bridge$ as the bridge connecting $Cl(x)$ and $Cl(y)$
4.          else
5.              edge $(x, y)$ is discarded
6.      if $x$ is a center and $y$ is free (or analogously if $y$ is a center and $x$ is free)
7.          add $y$ to $C(x)$
8.          mark $y$ as clustered
9.          add $(x, y)$ to the set of cluster edges
10.     if $y$ is free
11.         add $(x, y)$ to $F(x)$
12.         if $|F(x)| \geq 2 \cdot n_c^{1/3}$
                /* clean up set $F(x)$ */
13.             foreach edge $(x, z) \in |F(x)|$
14.                 if $z$ is not free
15.                     remove $(x, z)$ from $F(x)$
16.                     if $x$ is clustered and there is not a bridge
                                                        [ connecting $Cl(x)$ and $Cl(z)$
17.                         store $(x, z)$ as the bridge connecting $Cl(x)$ and $Cl(z)$
                /* check whether a new cluster centered on $x$ can be created */
18.             if $|F(x)| \geq n_c^{1/3}$
19.                 create a new cluster $C$, centered on $x$,
                                                        [ containing all $z$ s.t. $(x, z) \in F(x)$
20.                 if $x$ is free
21.                     add $x$ to $C$
22.                 else
23.                     store one of the cluster edges $(x, z)$ into $Bridge$
                                                        [ as the bridge connecting $Cl(x)$ and $C$
24.     if $x$ is free
25.         same as above, with $x$ and $y$ swapped
```

Figure 5.1: Procedure `CC-EdgeProcess` from algorithm `AFI`.

will see in Section 5.5, this allows us to produce spanners of different size and stretch, leading to a more flexible algorithm in practice.

We have represented each vertex as a 32-bit label. The same label also serves as key to a hash table whose corresponding entry points to a record containing all the data structures required for that node. Whenever an edge is streamed in, it is first checked whether any of its endpoints is seen for the first time (in which case a new entry in the hash table is created), or a pointer to its data structures is retrieved.

The symmetric (since the input graph is undirected) matrix *Bridge* has been

implemented as a 64-bit key hash table, plus a list of bridge edges, which operates in the following way. Upon insertion of an edge in entry $Bridge[i, j]$, an item is inserted in the hash table, with a key obtained by concatenating the two 32-bit values i and j (if $i < j$), or *viceversa* (if $i > j$). The corresponding edge is also appended to the list of bridge edges. When checking if there is a bridge between clusters C_i and C_j, a 64-bit key is created as described above and the hash table is queried.
When the stream runs out, the spanner is output as a stream of edges. During this phase, since algorithm `AFI` may keep information about an edge in the data structures of both its endpoints (and these need not be consistent), some postprocessing has been added in order to avoid the writing of duplicate edges.

5.2.2 The Algorithm by Baswana (`B`)

Recently, Baswana has proposed a randomized streaming algorithm [20], which computes a $(2k-1)$-spanner of expected size $O(k \cdot n^{1+1/k})$ in one pass in $O(m+n)$ total time (i.e., amortized constant processing time per edge), using $O(k \cdot n^{1+1/k})$ memory space. We will denote this algorithm by `B`.

Baswana's algorithm is based on the idea of partitioning the vertices of the input graph into a small number of disjoint subsets, called *clusters*, each spanned by a small number of edges. As a result of this process, each vertex u will have its neighbors grouped into various clusters. Including in the spanner only one of the (possibly many) edges that connect u to a given cluster C, will guarantee that any edge $v \in C$ will be at distance at most 1 plus the diameter of C from u. Therefore, by bounding the diameter of the clusters, the stretch factor of the resulting spanner can also be bounded. This idea is implemented in the algorithm through a multilevel randomized clustering scheme, as follows: for a $(2k-1)$-spanner, k clustering levels are created. Each node in the input graph forms a singleton cluster at level 0, and each cluster present at level i will also be present at level $i+1$ with probability $n^{-1/k}$ (notice that in order to implement this randomized scheme, the number of vertices n must be known in advance). As edges are streamed in, they are added to the adjacency list of the lowest level endpoint (the algorithm keeps an adjacency list for each vertex, storing at most one edge from the vertex to each adjacent cluster). Whenever a vertex gets connected by an edge to a cluster which is present at a higher level, it will join the cluster. This "upward movement" of vertices proves crucial for computing a sparse spanner, since the higher the level the fewer the clusters (at level $k-1$ there will be $n^{1/k}$ expected clusters, and it would be possible to add an edge from a vertex at this level to each of them, in the worst case). In order to achieve constant amortized processing time per edge, edges incident to a given vertex are added to a temporary buffer, which is *pruned* (i.e., it is compared with the vertex's adjacency list, and edges incident to an already adjacent cluster are discarded) only when it is sufficiently large.

Figure 5.2 reports a detailed description of the three main procedures of algorithm `B`. It is assumed that the input graph will be given as a stream of edges in arbitrary order, where each edge consists of a pair of node ids in the range $(1, \ldots, n)$. The algorithm will first initialize its random clustering scheme through procedure `Init`, by including all vertices in the first clustering level (S_0), and promoting a vertex from one level to the next with probability $n^{-1/k}$ (cf. lines 1–3 of procedure `Init`). Notice that the number of vertices n of the input graph must be known by the procedure. The k levels of clustering will be stored in the C_i arrays (with $0 \leq i < k$), where $C_i(u)$ will store the center of the cluster in C_i that contains u. In the case that u is not clustered at level i, it will store 0. Initially, each cluster is initialized to a singleton set (lines 4–5 of `Init`).
In the following we will denote by $\ell(u)$ the highest level of the clustering in which u is currently present as a clustered vertex, and by $\ell_S(u)$ the highest level of the clustering in which u was present at the beginning of the algorithm (immediately after the execution of `Init`). After procedure `Init` has completed, procedure `ProcessEdge` will be called for each edge in the stream. The algorithm will use the following simple data structures. For each vertex $u \in V$ it will keep lists $E(u)$ and $Temp(u)$. $E(u)$ will store edges incident on u from clusters at level $\ell(u)$, while $Temp(u)$ will act as a buffer for these edges. First it is checked whether the current edges connects a vertex to a cluster at a higher level (lines 1–4 of procedure `ProcessEdge`). If this is indeed the case, then the vertex is promoted to the higher level (lines 5–7), its incident edges join the output spanner (set E_S denotes edges that will be added to the output spanner in the current iteration), and its lists of adjacent edges are flushed (lines 8–9). Otherwise, the current edge is just appended to the temporary buffer $Temp(u)$ (line 11). $Temp(u)$ will be purged once the number of edges in it exceeds those in $E(u)$, through procedure `Prune` (lines 12–13). This solution proves crucial for achieving constant amortized processing time per edge.
Procedure `Prune` first scans list $E(u)$, and sets to 1 entries in the boolean array $A[1..n]$ that correspond to clusters in C_i neighboring to u (lines 1–2). Then it scans $Temp(u)$, eliminating an edge if the corresponding cluster was already incident to u, or adding it to $E(u)$ if otherwise (lines 3–7). Finally, it resets all entries of the auxiliary boolean array A to 0 (lines 8–9).

Implementation Details.

Our implementation of algorithm `B` follows very closely the theoretical layout of Figure 5.2. The algorithm writes the output spanner to disk as a stream of

```
    Procedure Init
    input: number of clustering levels $k$, number of nodes $n$
1.      let $S_0 \leftarrow V$, $S_k = \emptyset$
2.      for $0 < i < k$
3.          $S_i$ contains each element of set $S_{i-1}$ independently with prob. $n^{-1/k}$
4.      for $0 \leq i \leq k$
5.          $C_i \leftarrow \{\{v\}|v \in S_i\}$
```

```
    Procedure ProcessEdge
    input: edge $(u, v)$
        /* assigning the edge to the endpoint at lower level */
1.      if $\ell(u) > \ell(v)$
2.          swap $(u, v)$
3.      $i \leftarrow \ell(u)$, $x \leftarrow C_i(v)$, $h \leftarrow \ell_S(x)$
        /* processing the edge */
4.      if $h > i$
5.          for $j = i + 1$ to $h$
6.              $C_j(u) \leftarrow x$
7.          $\ell(u) \leftarrow h$
8.          $E_S \leftarrow Temp(u) \cup E(u) \cup \{(u, v)\}$
9.          $Temp(u) \leftarrow \emptyset$, $E(u) \leftarrow \emptyset$
10.     else
11.         $Temp(u) \leftarrow Temp(u) \cup \{(u, v)\}$
12.         if $|Temp(u)| \geq |E(u)|$
13.             Prune $(u, i)$
```

```
    Procedure Prune
    input: vertex $u$, clustering level $i$
1.      foreach edge $(u, w) \in E(u)$
2.          $A[C_i(w)] \leftarrow 1$
3.      foreach edge $(u, v) \in Temp(u)$
4.          if $A[C_i(v)] = 0$ and $C_i(u) \neq C_i(v)$
5.              $A[C_i(v)] \leftarrow 1$
6.              $E(u) \leftarrow E(u) \cup \{(u, v)\}$
7.          $Temp(u) \leftarrow Temp(u) \backslash (u, v)$
8.      foreach edge $(u, w) \in E(u)$
9.          $A[C_i(w)] \leftarrow 0$
```

Figure 5.2: Procedures `Init`, `ProcessEdge` and `Prune` from algorithm `B`.

edges. However, unlike algorithm `AFI`, where no edges can be dropped from memory before the input stream has run out, we notice that in algorithm `B`, whenever a vertex is promoted to a higher level, its currently known incident edges will certainly enter the output spanner, and they are no longer needed by the algorithm (cf. lines 8–9 of procedure `ProcessEdge`). We seize this opportunity, in our practical implementation, to flush these edges directly to the output stream (even before the input stream has run out), thus freeing some internal memory.

5.2.3 The Algorithm by Elkin (`E`)

The randomized streaming algorithm proposed by Elkin [48] (denoted by `E`) computes a $(2k-1)$-spanner of size $O((k \log n)^{1-1/k} \cdot n^{1+1/k})$ (with high probability) in one pass, using $O((k \log n)^{1-1/k} \cdot n^{1+1/k})$ memory space and processing each edge in $O(1)$ expected time.

Algorithm `E` is based on a randomized vertex labelling scheme. In a preprocessing phase, the algorithm assigns to each vertex of the input graph a unique identifier $I(v) \in \{1, \dots, n\}$, and a *radius* $r(v) \in \{0, \dots, k-1\}$ from the *truncated geometric probability distribution* given by $\mathbb{P}(r = t) = p^t \cdot (1-p)$, for $t \in \{0, \dots, k-2\}$, and $\mathbb{P}(r = k-1) = p^{k-1}$, where $p = (\frac{k \log n}{n})^{1/k}$. Notice that this initialization phase requires prior knowledge of the number of vertices n. During its execution, the algorithm also maintains for every vertex v a label $P(v) \in \{1, 2, \dots, n \cdot k\}$. These labels are initialized as $I(v)$. A label P in the range $i \cdot n + 1 \leq P \leq (i+1) \cdot n$ is said to belong to *level* i (and we will write $L(P) = i$). To every label P it is also associated a *base value* $B(P)$, given by $B(P) = n$, if n divides P, and by $B(P) = P \bmod n$, otherwise. A label is said to be *selected* if $L(P) < r(B(P))$.

The algorithm maintains for each vertex v an edge set $Sp(v)$ (initially empty), that grows monotonically during the execution of the algorithm. When all edges have been processed, the union of these edge sets will constitute the output spanner. Set $Sp(v)$ can be thought of as being implicitly divided into two disjoint subsets: $T(v)$ in which the algorithm stores edges belonging to a *tree cover* of the input graph, and $X(v)$ in which it stores edges connecting the trees that make up the tree cover. Additionally, for every vertex v a table $M(v)$ is maintained, storing the base values of levels P such that there exists at least one neighbor z of v that was labelled by P at some point of the execution, and such that edge (v, z) was inserted into $X(v)$ at that point.

Edges are added to $Sp(v) = T(v) \cup X(v)$ by procedure `ReadEdge`, which is invoked on all edges of the input graph, as follows (line numbers refer to the pseudocode in Figure 5.3): upon seeing an edge (u, v), the algorithm determines the vertex with the greatest label (ties are broken by comparing the respective vertex identifiers $I(u)$ and $I(v)$) (line 1). Without loss of generality,

```
    procedure ReadEdge(edge $(u, v)$)
1.      let $u$ be the vertex s.t. $P(u) \succ P(v)$
2.      if ($P(u)$ is a selected label)
3.          $P(v) \leftarrow P(u) + n$
4.          $Sp(v) \leftarrow Sp(v) \cup \{(u, v)\}$
5.      else
6.          if ($B(P(u)) \notin M(v)$)
7.              $M(v) \leftarrow M(v) \cup \{B(P(u))\}$
8.              $Sp(v) \leftarrow Sp(v) \cup \{(u, v)\}$
```

Figure 5.3: Procedure ReadEdge from algorithm E.

let u be the vertex with greatest label: if $P(u)$ is selected, then (u, v) is inserted into $T(v)$, and $P(v)$ is set to $P(u) + n$ (lines 2–4). Otherwise, if $B(P(u))$ is not in $M(v)$, it is inserted into the table and (u, v) is added to $X(v)$ (lines 5–8).

Implementation Details.

Our implementation follows closely the theoretical description given by Elkin. The $M(v)$ tables have been implemented as hash tables, that allow membership queries and insertions to be performed in $O(1)$ amortized time. Since items are never removed from the $Sp(v)$ sets by the algorithm, our implementation immediately flushes to disk edges that join these sets, as a sequential output stream, thus avoiding any unnecessary storing of edges.

5.3 The Off-line Algorithm by Zwick (Z)

In this section we describe a simple off-line algorithm due to Zwick [105], which we denote by Z. Given an unweighted undirected graph with n vertices and m edges, it computes in $O(m+n)$ time a $(2k-1)$-spanner with $n^{1+1/k}$ edges for any integer $k \geq 2$. We use this algorithm as a benchmark in the performance analysis of AFI and B.

Algorithm.

Let $N_i(v)$ be the vertices that are at distance exactly i from v. The algorithm works by alternating the following two phases.

1. In the first phase, it chooses an arbitrary vertex v, starts from $N_0(v) = \{v\}$, and computes $N_i(v)$ for $i = 1, 2, 3, \ldots$, by finding all unreached

vertices that are one hop away from a vertex in $N_{i-1}(v)$, until:

$$|N_i(v)| \leq n^{1/k} \cdot |N_{i-1}(v)|$$

Notice that this must happen for some $i \leq k$, as otherwise $|N_k(v)| > n$, which is impossible.

2. In the second phase, the algorithm builds a shortest path tree from v to all the vertices at distance at most i from v, and adds the edges of this tree to the spanner. Then it removes from the graph the vertices at distance at most $i - 1$ from v, and all the edges touching them.

These two phases are repeated until no vertices remain.

Analysis.

It is easy to see that the total running time is linear, as each edge is explored at most twice. It is also not difficult to see that the subgraph constructed is a $(2k - 1)$-spanner of the original graph. Let $e = (u_1, u_2)$ be an edge in G. When e is removed from the graph, we have $u_1 \in N_{j_1}(v)$ and $u_2 \in N_{j_2}(v)$, for some vertex v, and for some indices $j_1, j_2 \leq k$. In fact, at most one of these indices can be k, so $j_1 + j_2 \leq 2k - 1$. (We also have $|j_1 - j_2| \leq 1$.) Thus, the shortest path tree added to the spanner at that stage contains a path between u_1 and u_2, through the root of the tree, whose length is at most $j_1 + j_2 \leq 2k - 1$. Simple calculations show that the number of edges in the spanner constructed is at most $n^{1+1/k}$ (let E_S be the number of edges and V_S the number of vertices added to the spanner during phase 2 of each iteration; then at each iteration it will hold that $E_S/V_S \leq n^{1/k}$).

Implementation Details.

Our implementation of algorithm Z merges together phases 1 and 2, in a *blocked* version of a *breadt-first search* (BFS) over the input graph: i.e., a BFS is started from an arbitrary vertex, and every time the search is advanced one level, a special separator item is inserted into the BFS vertex queue. If the condition of phase 1 is met at a specific level, then the edges connecting that level to the previous one are removed from the original graph, and flushed to disk as output (we remark that since the graph is unweighted the edges of subsequent levels form a shortest path tree). Otherwise, the BFS is halted and a new start node is arbitrarily selected.

5.4 Experimental Setup

We ran our experiments on a PC equipped with a 3 GHz Intel Pentium 4 dual core processor, 2 MB L2 cache, 2 GB RAM, using a 50 GB Ext3 partition on a 250 GB IDE hard disk running Linux Mandriva 2007 kernel 2.6.17. All algorithms were coded in C using a uniform programming style and common data structures. Programs were compiled with `gcc 4.1.1` with optimization flag `-O4`.

Performance Measures. The performance measures we considered are:

- *Estimates of average stretch, stretch variance, and maximum stretch*: Based on a sample of distances from 100 random sources to all other vertices in the original graph and in the spanner. These estimates proved to be very close to the exact values obtained by running an all-pairs shortest paths implementation, but much faster to compute.
- *Spanner size*: Number of edges in the spanner computed by the algorithm.
- *Running time*: Total time in seconds required to compute a spanner (over the whole stream).
- *Memory footprint*: Peak amount of memory in megabytes required by the algorithm during its execution.

Each measured value reported in our charts was obtained as the average of at least three independent trials on the same data point.

Test Sets. In our experiments, we considered the following synthetic graph families:

- *Power Law graphs*: Small world graphs based on the recursive matrix (R-MAT) model [32]. These graphs have the property that their degree distribution $f(d)$ follows a power law relationship (e.g., $f(d) \sim d^{-\gamma}$, for some constant γ): in other words, a small number of nodes act as hubs (having a large degree), while most nodes have a small degree. Power law graphs model a wide range of real-world networks, from Internet topology [51] to protein interaction networks [4].
- *Random graphs*: Based on the Erdös-Rényi $G_{n,p}$ model [50].
- *Clustered graphs*: Graphs made of clusters connected to form a line. Each cluster is a random $G_{n,p}$ graph. These graphs tend to have higher diameter than the power law and random graphs.

In all cases, graphs were presented to the algorithms as an on-line stream of edges in random order.

5.5 Experimental Results

In this section we present our experimental results. Since we have observed similar trends for all the graph families considered, we report the charts only for the power law graphs, which are the most relevant to real-world applications. The complete experimental package, which includes the C source codes of our implementations, and charts summarizing our experimental results for all the classes of graphs we have considered can be found at [1].

We will first discuss each algorithm separately, then study the differences between the various techniques in terms of both quality of solutions and time/space requirements.

5.5.1 Algorithm `AFI`

The charts of Figure 5.4 and Figure 5.5 show how spanner size, average stretch, running time, and memory footprint depend upon the value set for the clustering threshold on a power law graph with $n = 8,192$ and $m = 13,003,600$. Observe that the number of edges in the spanner initially drops down with the clustering threshold c, reaches a minimum around $c = 50$, and then grows up again for higher values of c. Intuitively, this can be explained by the fact that for small values of c the algorithm forms many clusters, and therefore there will be many CC-bridge edges (i.e., inter-cluster edges, see Section 5.2.1). On the other hand, if the threshold is high there will be many free vertices, and sets $F(\cdot)$ can be larger. The running time and the memory footprint follow the same trend, while the smaller the spanner, the higher the stretch factor. Notice that the value of the clustering threshold that minimizes the spanner size minimizes also the running time and the space usage. Interestingly, while this optimal value may depend upon n and m, the adaptive choice of $c = n_c^{1/3}$ made in the original algorithm, plotted as dashed horizontal lines in the charts of figures 5.4 and 5.5, seems to be very close to it. We also notice that the spanner size can be much smaller in practice than the theoretical bound $k \cdot n^{1+1/k}$ as shown in Figure 5.4 (a), suggesting that multiplicative factors in `AFI`'s $O(n^{1+1/k})$ bound on the number of spanner edges may be small.

5.5.2 Algorithm `B`

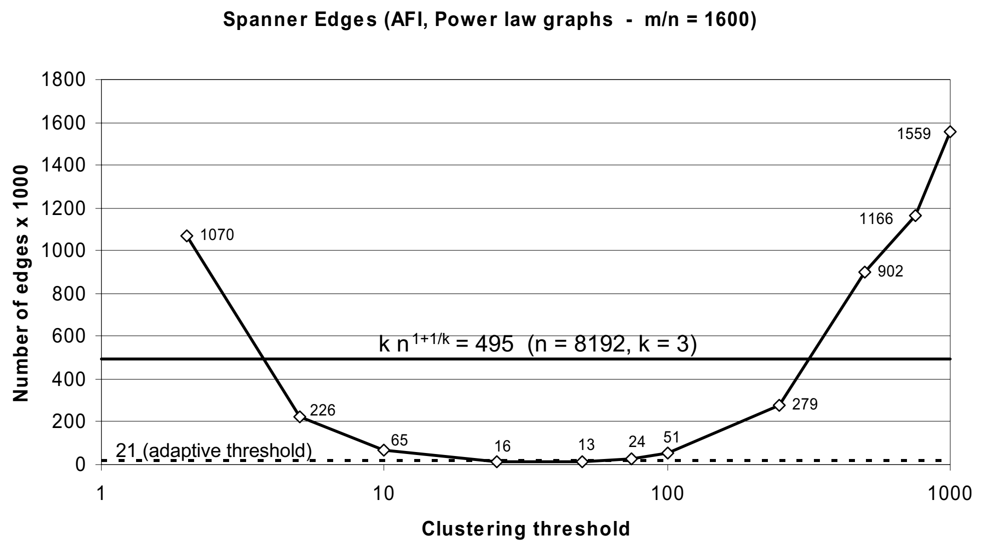
In Figure 5.6 and Figure 5.7 we show how spanner size, average stretch, running time, and memory footprint depend upon parameter k on a power law graph with $n = 8,192$ and $m = 13,003,600$. We first notice that the number of edges in the spanner follows the theoretical trend of $k \cdot n^{1+1/k}$ for small values of k, but it quickly gets flattened out for $k > 10$. Similarly to what we have seen for `AFI`, the smaller the spanner, the higher the average stretch. However, differently from `AFI`, the running time is minimized for the value of k that maximizes the spanner size, i.e., $k = 2$. Notice that, while increasing k beyond 100 does not yield any spanner size or stretch benefits, it raises the memory consumption considerably due to the higher number of clustering levels the algorithm needs to maintain. On all the test sets we considered, the algorithm seems to be of practical value only for small values of k.

5.5.3 Algorithm `E`

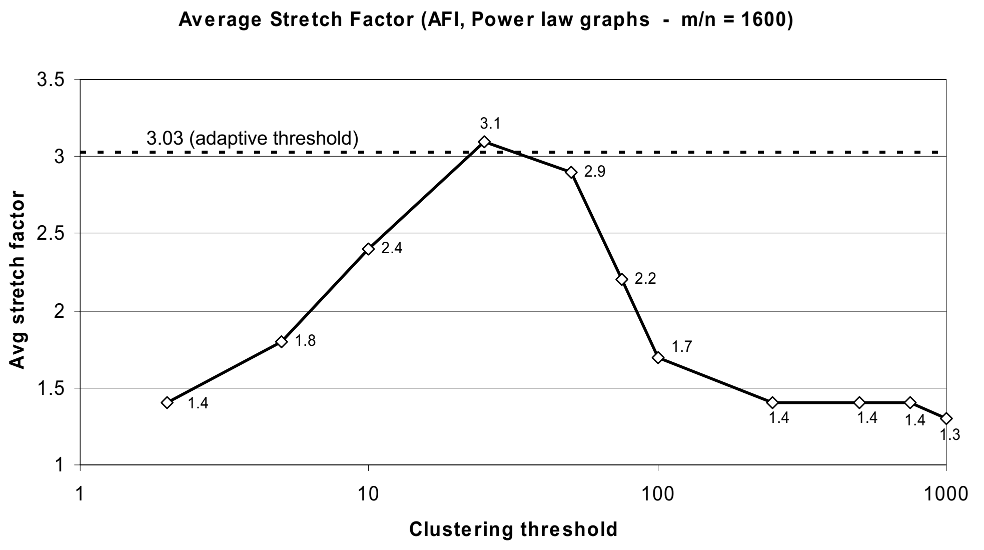
Figure 5.8 and Figure 5.9 show how spanner size, average stretch, running time, and memory footprint depend upon parameter k on a power law graph with $n = 8,192$ and $m = 13,003,600$. We notice that the size of the spanners obtained by algorithm `E` follows roughly the theoretical trend $(k \log n)^{1-1/k} \cdot n^{1+1/k}$. In particular, the number of edges is decreasing for small values of k and reaches a minimum around $k = 25$. Differently from `B`, however, it increases again for larger values of k. The stretch factor follows the opposite trend, leading to the somewhat ironical situation where pushing k beyond a certain threshold results in a reduction of the stretch factor. This can be explained by the fact that larger spanners tend naturally to have smaller stretch. Another relevant aspect of `E` shown in Figure 5.9 is that the running time and the memory usage are maximum for $k = 2$ and decrease with k. Differently from `B`, we noticed that algorithm `E` tends to be inefficient for small values of k.

5.5.4 Algorithm `Z`

Figure 5.10 and Figure 5.11 show how spanner size, average stretch, running time, and memory footprint depend upon parameter k on a power law graph with $n = 8,192$ and $m = 13,003,600$, for algorithm `Z`. We notice that all four performance measures exhibit very little variability. Spanner size for $k = 2$ is roughly equal to the value obtainable through algorithm `AFI` with adaptive clustering threshold, and it settles to a slightly lower value (overall best) by $k = 10$, remaining constant from there onwards. The average stretch factor exhibits only slight variations as the spanner size settles to its final value. Memory usage is practically constant throughout, and it is a lot higher than the other algorithms, which is not at all surprising, considering that Zwick's algorithm has to keep the entire input graph in main memory. Running time is also roughly constant.

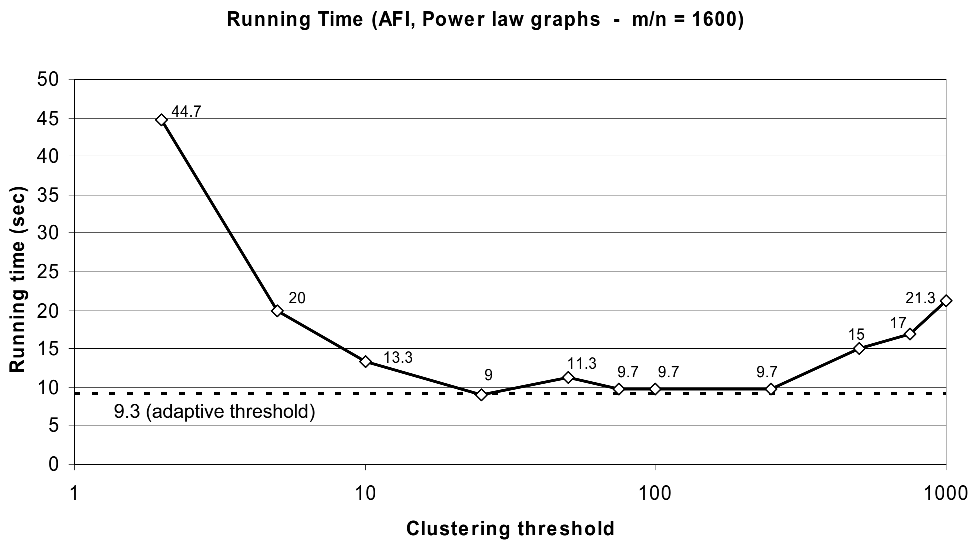


(a)

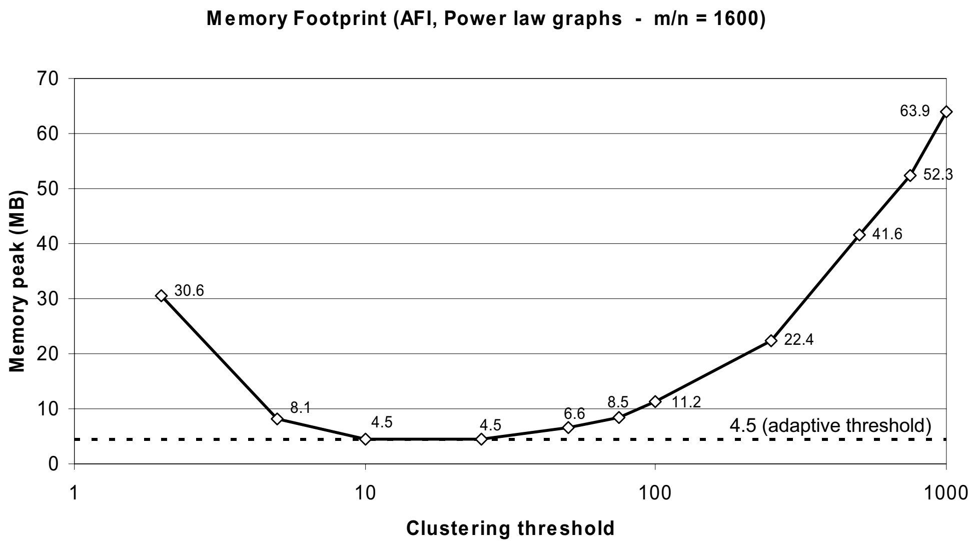


(b)

Figure 5.4: Analysis of algorithm AFI on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of the clustering threshold: (a) number of edges in the spanner; (b) average stretch estimate.

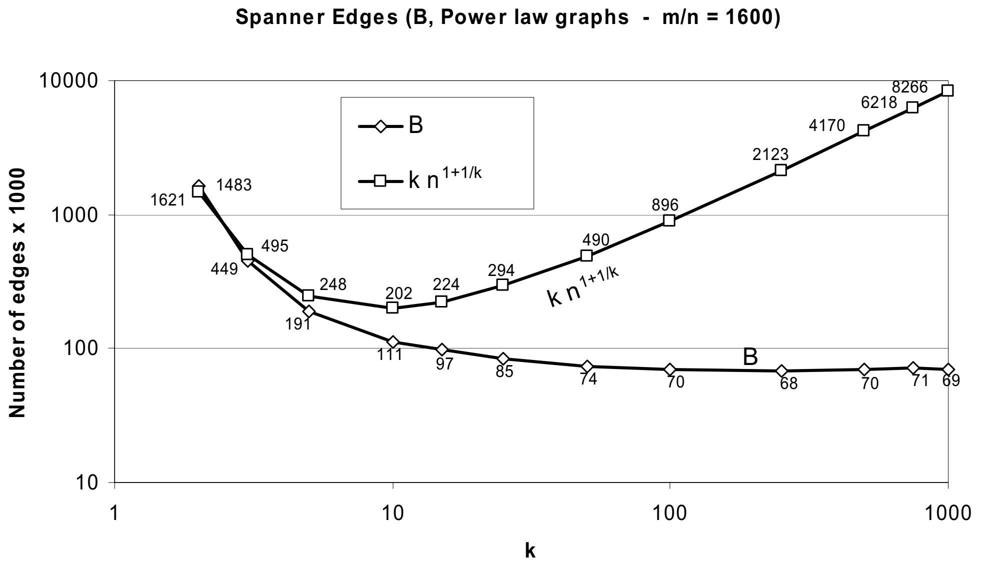


(a)

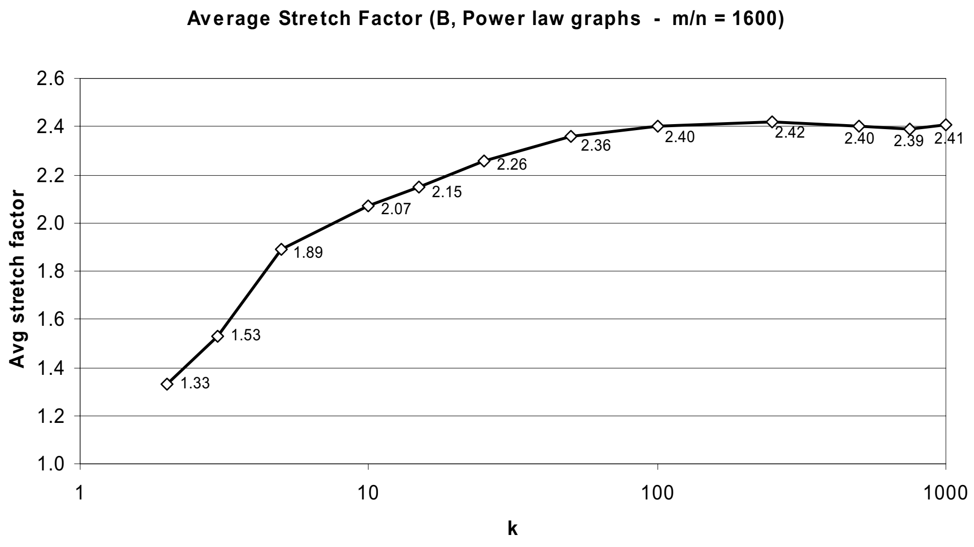


(b)

Figure 5.5: Analysis of algorithm `AFI` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of the clustering threshold: (a) running time in seconds; (b) memory peak in megabytes.

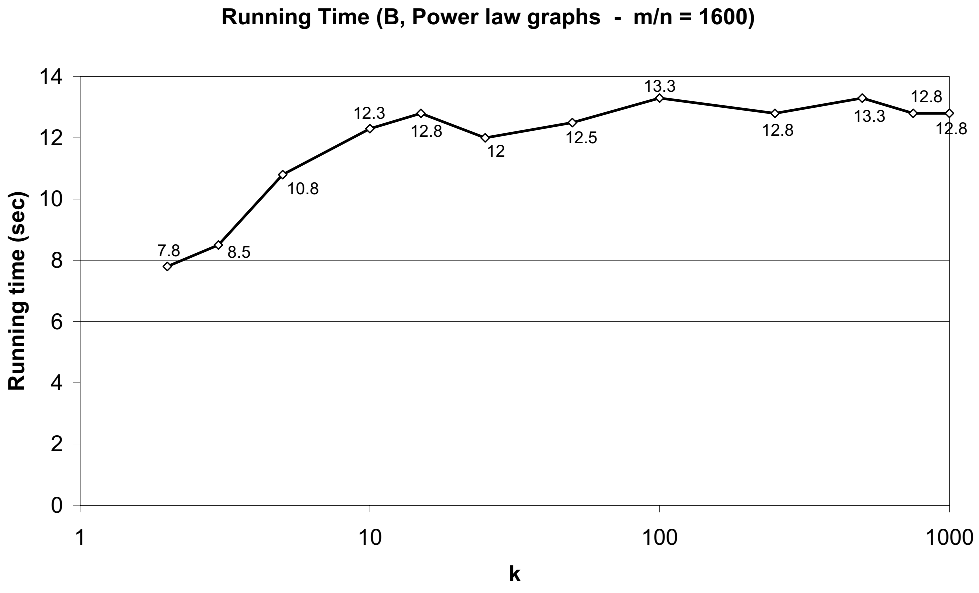


(a)

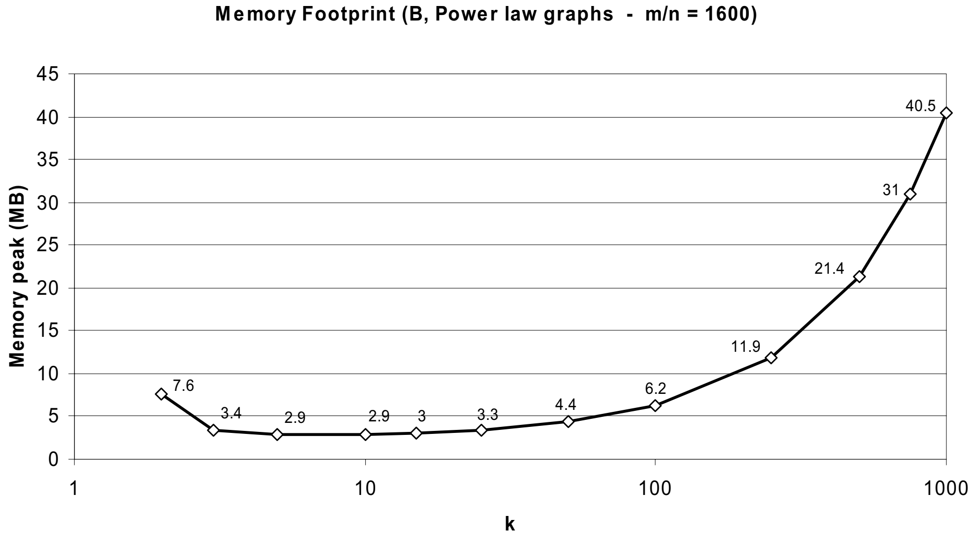


(b)

Figure 5.6: Analysis of algorithm B on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) number of edges in the spanner; (b) average stretch estimate.

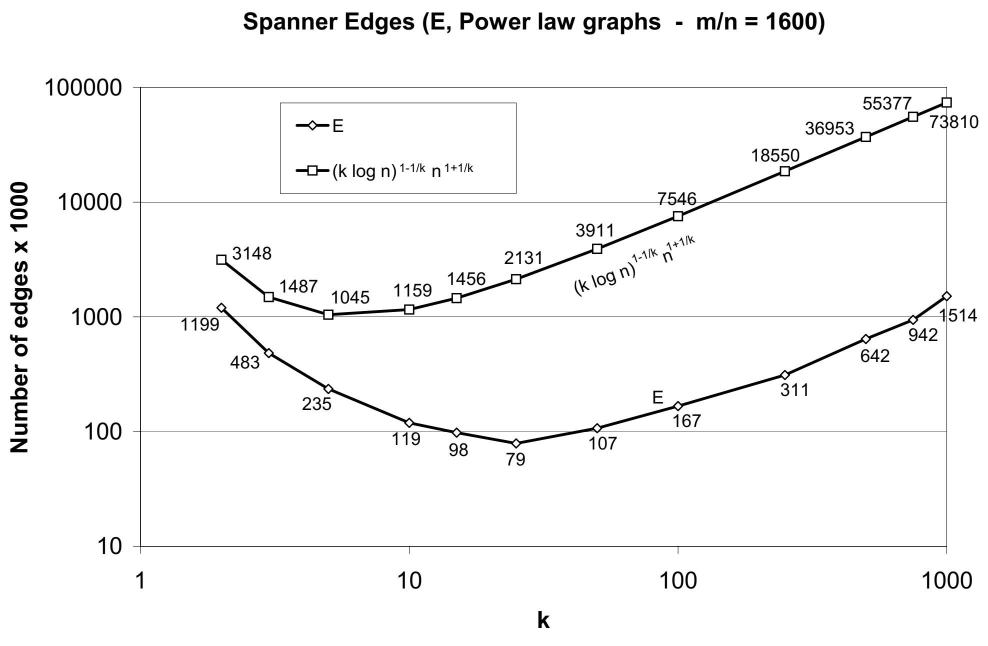


(a)

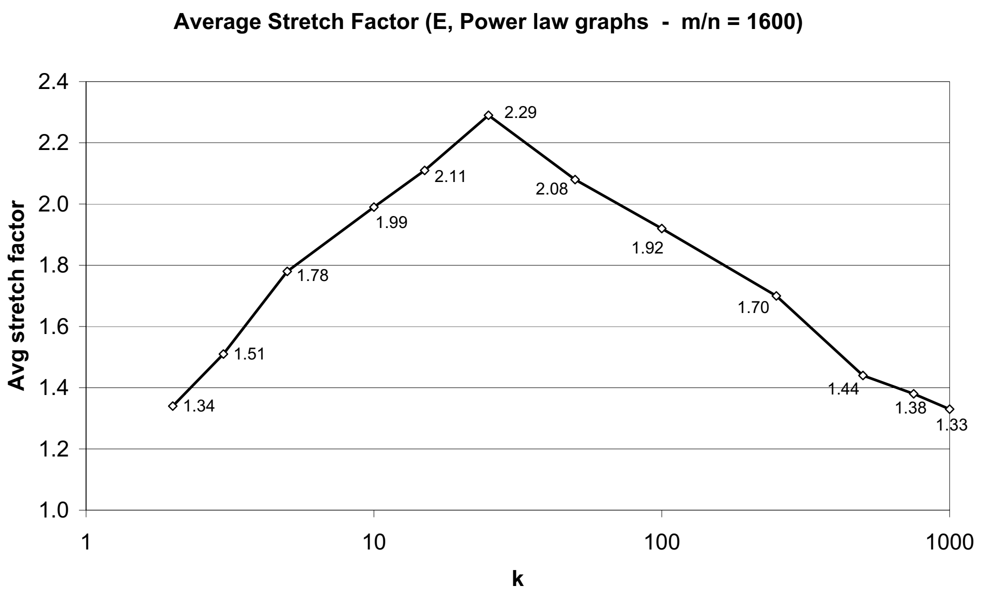


(b)

Figure 5.7: Analysis of algorithm B on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) running time in seconds; (b) memory peak in megabytes.

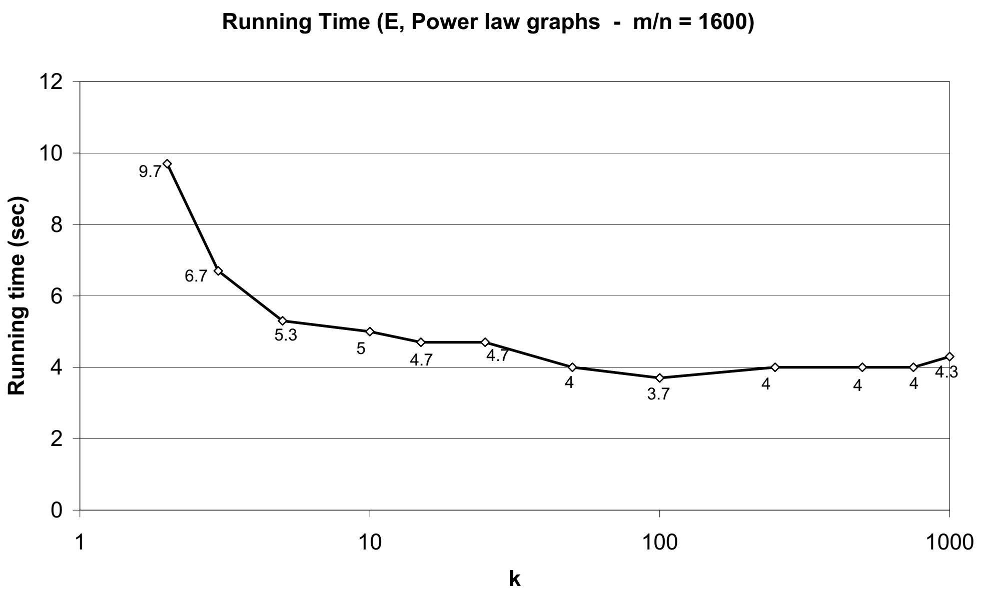


(a)

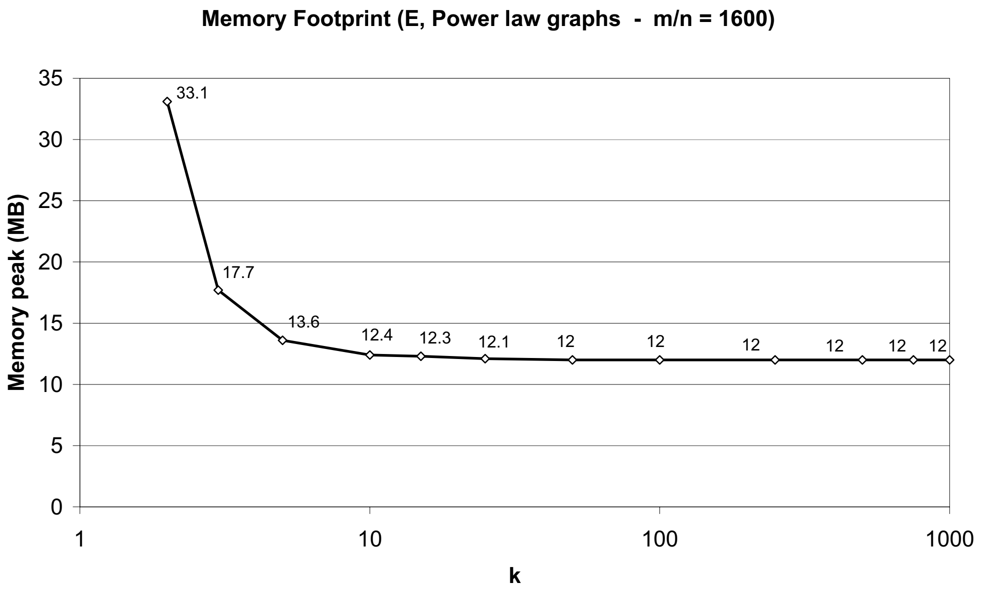


(b)

Figure 5.8: Analysis of algorithm E on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) number of edges in the spanner; (b) average stretch estimate.

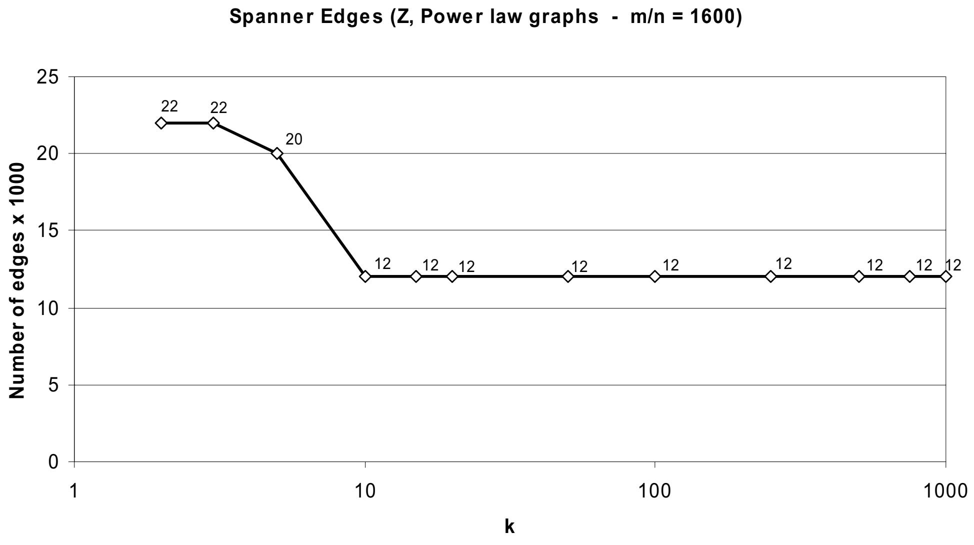


(a)

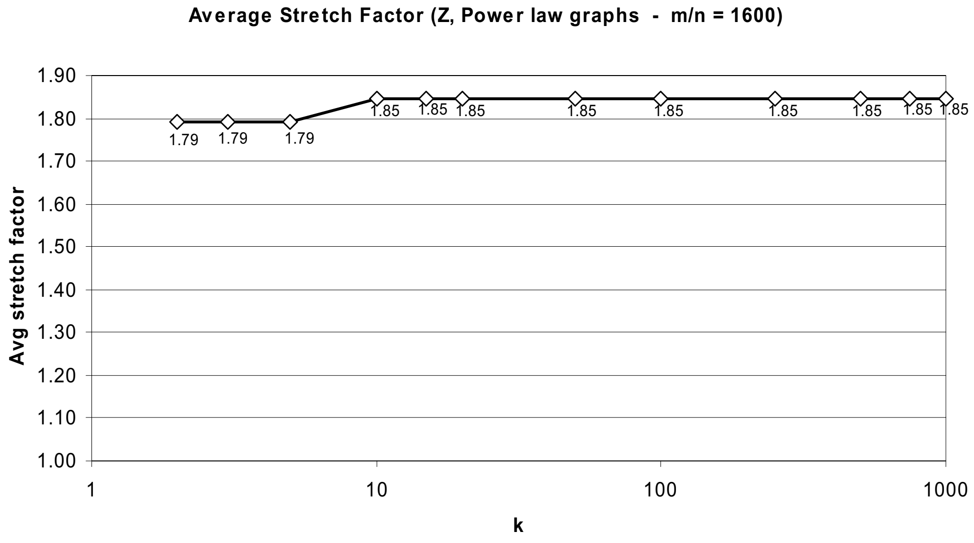


(b)

Figure 5.9: Analysis of algorithm E on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) running time in seconds; (b) memory peak in megabytes.

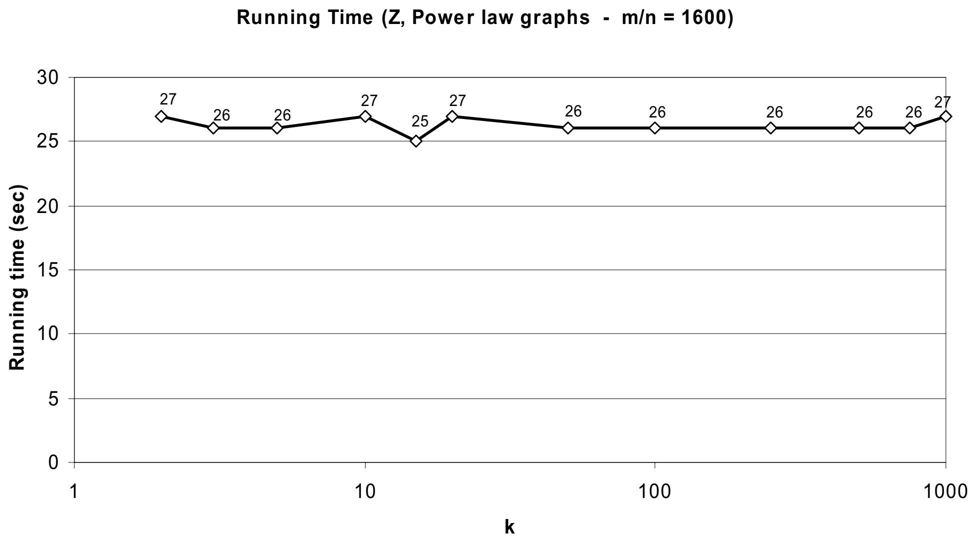


(a)

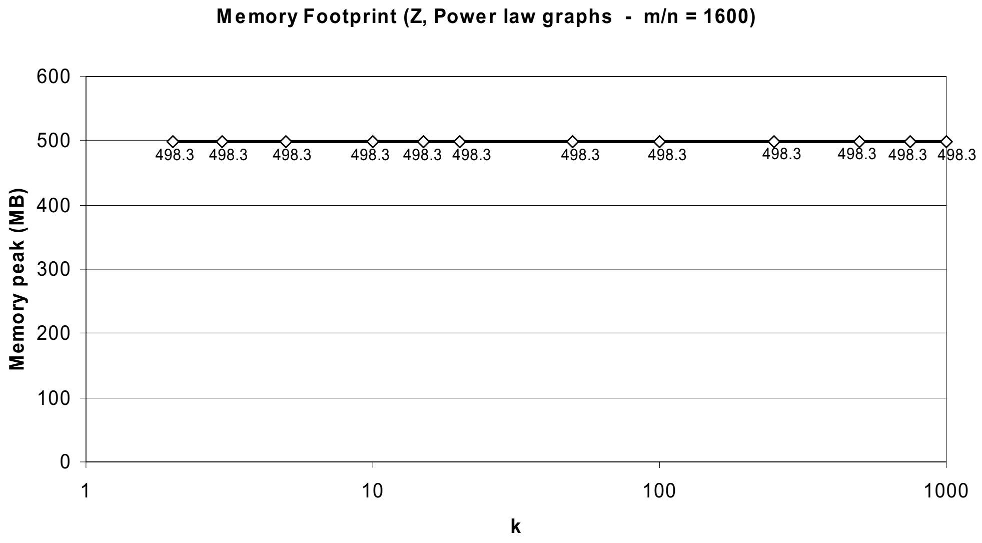


(b)

Figure 5.10: Analysis of algorithm Z on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) number of edges in the spanner; (b) average stretch estimate.



(a)



(b)

Figure 5.11: Analysis of algorithm Z on a power law graph with $n = 8,192$ and $m = 13,003,600$ for different values of k: (a) running time in seconds; (b) memory peak in megabytes.

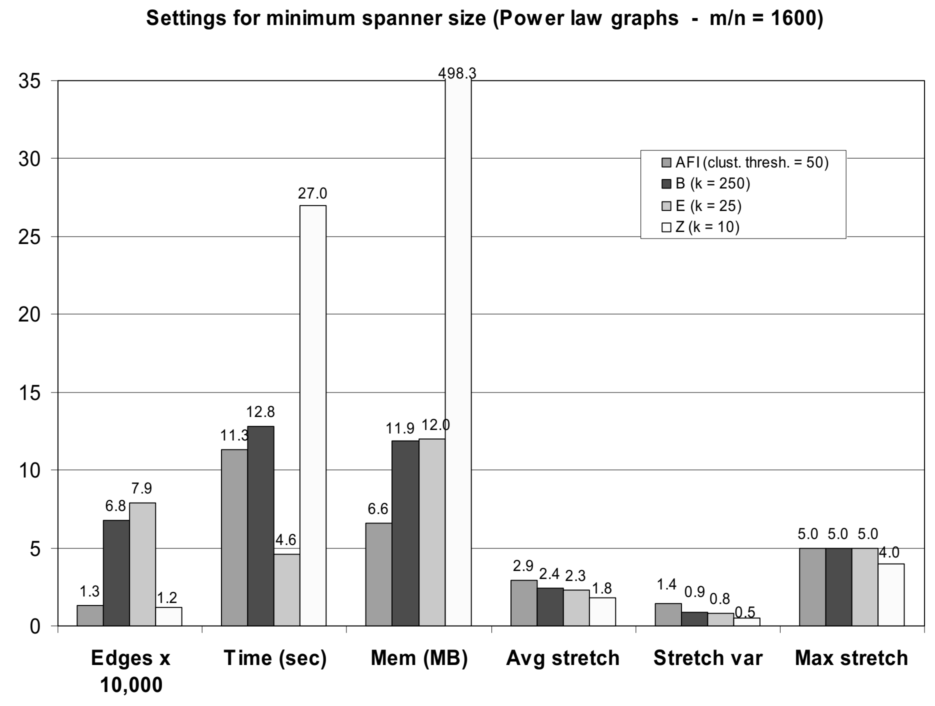
5.5.5 Overall Comparison

Figure 5.12 shows an overall comparison of algorithms `AFI`, `B`, `E`, and `Z` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for two choices of parameters k (for `B`, `E`, and `Z`) and c (for `AFI`): the setting that minimizes the spanner size (a) and the setting that minimizes the stretch factor (b). We observe that, to minimize the spanner size, `E` was the fastest and `AFI` was the most efficient in terms of memory requirements. In this scenario, they seemed to be generally preferable to `B`. Conversely, `B` was slightly faster and used substantially less memory than `AFI` and `E` in the stretch factor minimization setting. In both cases, `AFI` was able to find much smaller spanners than `B` and `E`, with very little loss in the average stretch. All streaming algorithms we considered were much faster and used substantially less memory than `Z`, which needs to keep the whole graph in internal memory. They were also able to find spanners with smaller stretch than `Z`, and this is somewhat surprising given that `Z` is an off-line algorithm.

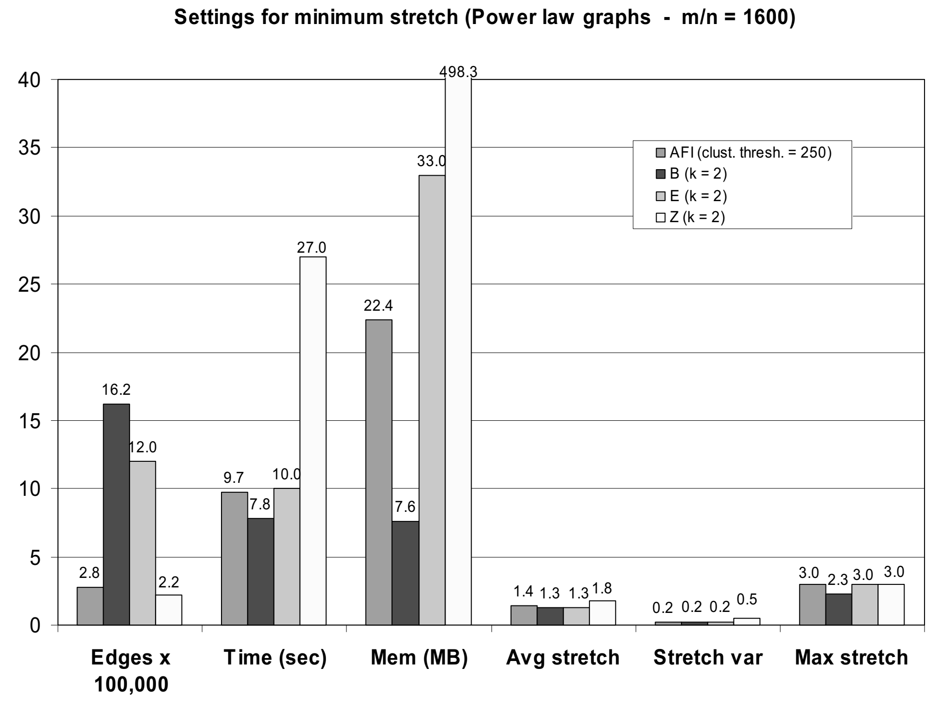
Figure 5.13 focuses on the quality of the spanners found by `AFI`, `B`, `E`, and `Z` on two power law graphs with different edge densities. The charts plot size and stretch of the spanners that can be achieved by considering the full range of parameters k and c. Ideally, we would like to find a spanner with the smallest stretch and the smallest size simultaneously. We first observe that the curves related to algorithms `B` and `E` on both sparse and dense graphs are mostly overlapping, suggesting that the spanners found by the two algorithms have very similar features. This may be a consequence of the fact that both algorithms build upon the techniques of Baswana and Sen [21], labelling vertices by numbers and using these labels to decide whether incoming edges have to be inserted in the spanner or not. In the case of the graph with $n = 8,192$ and $m = 13,003,600$ ($m/n = 1600$), `Z` yielded the best spanners, and `AFI` found spanners of substantially better quality than `B` and `E` (a): in particular, it was able to construct spanners by the same stretch factor, but roughly 4-5 times smaller. For the sparser power law graph with $n = 262,144$ and $m = 13,927,365$ ($m/n = 53$) the differences between the algorithms were instead negligible (b); `Z`, however, managed to sparsify the (already sparse) input graph to a higher extent than the other algorithms, obviously at the expenses of a worse stretch factor.

To summarize, our experiments suggest that:

- Algorithms `AFI`, `B`, and `E` are likely to be very effective in practice, as they find spanners of size and stretch much smaller than the theoretical bounds, and are competitive with off-line algorithms. In addition to being intrinsically space-efficient, surprisingly they can also be substantially faster than internal-memory algorithms such as `Z`.

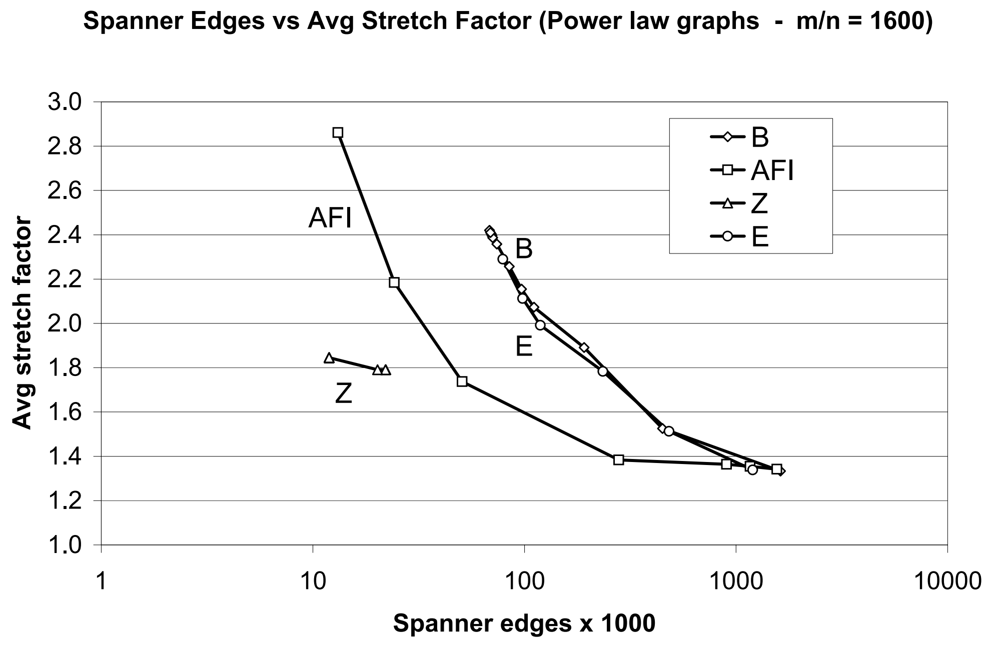


(a)

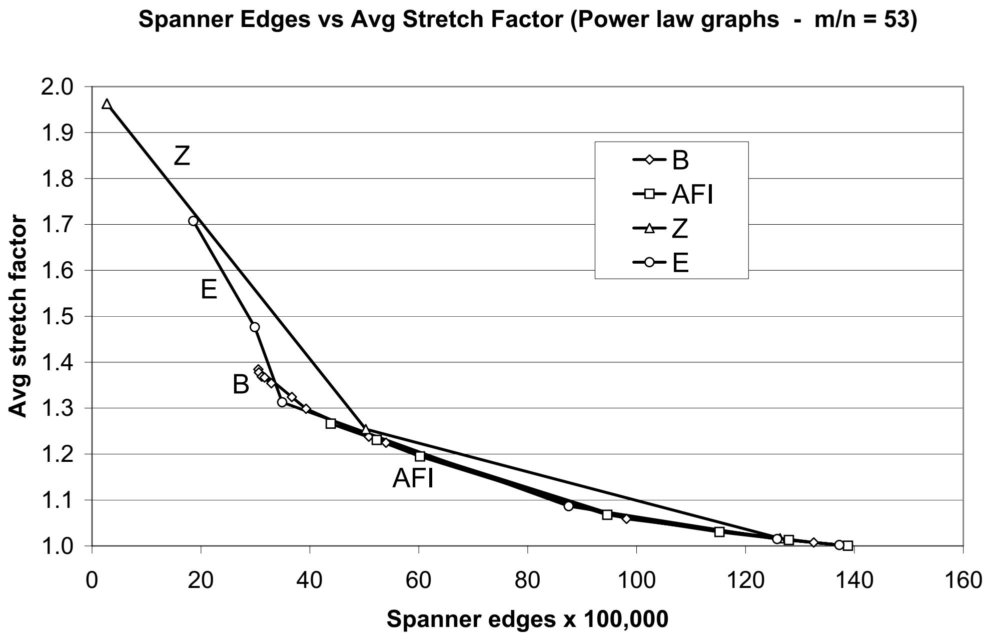


(b)

Figure 5.12: Comparison of algorithms `AFI`, `B`, `E`, and `Z` on a power law graph with $n = 8,192$ and $m = 13,003,600$ for the values of k (for `B`, `E`, and `Z`) and of the clustering threshold (for `AFI`) that yield the smallest spanner (a) and the smallest stretch (b).



(a)



(b)

Figure 5.13: Comparison of the solution quality achieved by algorithms `B`, `AFI`, `Z`, and `E` on a power law graph with $n = 8,192$ and $m = 13,003,600$ (a) and on a power law graph with $n = 262,144$ and $m = 13,927,365$ (b) for different values of k (for `B`, `Z`, and `E`) and of the clustering threshold (for `AFI`).

- Algorithms `B` and `E` produce spanners of comparable quality, but `E` seems to be faster in the case where the spanner size is to be minimized, while `B` appears to be more space-efficient when the goal is to minimize the stretch factor.
- Setting large values of k in `B` and `E` does not produce sparser spanners in all test sets we have considered, so small values of k, for which `AFI` was designed, seem to be the case of interest in practice.
- `AFI`, in addition to being deterministic and not requiring prior knowledge of the number of vertices in the graph, seems to have the additional benefit of producing spanners of substantially better quality than `B` and `E`, while still using a comparable amount of time and space resources.

5.6 Conclusions

In this chapter, we have compared the experimental performances of three one-pass *Semi-streaming* algorithms for constructing spanners of unweighted graphs. We have put our implementations to the test on a variety of graph classes (Erdös-Rényi random graphs, power law graphs, clustered graphs), measuring several performance parameters, such as spanner size, average stretch factor, maximum stretch factor, stretch variance, running time and peak memory usage. We have also implemented an internal memory algorithm for the same problem, to serve as a benchmark.

Our experiments indicate that all three algorithms are likely to be very efficient in practice, as they have produced spanners of size and stretch much smaller than the worst-case theoretical bounds, on all test instances we have considered, and are in fact competitive with off-line algorithms, while enjoying a much wider range of applicability, due to their greatly reduced memory requirements.

Chapter 6

Conclusions and Further Directions

In this dissertation we have focused on streaming algorithms for graph problems.

In Chapter 3 we have devised algorithms for fundamental graph problems such as *connected Components*, *minimum spanning tree* and *multiple-sources shortest paths* in the *W-Stream* model (a variant of the *classical streaming* model that allows sequential reading and writing of intermediate streams). Our algorithms achieve a smooth tradeoff between the number of passes and the working memory they require. We have also proved a number of hardness and separation results between *classical streaming* and *W-Stream*, and we have adapted a communication complexity-based technique for obtaining lower bounds in *classical streaming* to the *W-Stream* setting, proving our algorithms for *connected Components* and *minimum spanning tree* to be optimal up to a logarithmic factor. Finally, we have shown that some memory/passes tradeoffs are possible even in the more restrictive *classical streaming* model, for simpler variants of natural graph problems, or even for fundamental graph problems such as *Undirected s-t connectivity*, at least for sparse graphs.

In Chapter 4 we have devised reduction techniques for obtaining efficient *W-Stream* algorithms from parallel ones. We have first introduced a general simulation technique of PRAM algorithms that, while yielding near optimal *W-Stream* algorithms for some problems (e.g., *sorting*), leads to suboptimal results for others (most notably, graph problems). We have then introduced an intermediate model (RPRAM), that while being more powerful than the traditional PRAM model, can be simulated in *W-Stream* within the same asymptotic bounds. By reimplementing PRAM graph algorithms in RPRAM, we have then been able to solve efficiently (i.e., within polylogarithmic factors away from the optimum) graph problems such as *biconnected components* and *maximal independent set* in *W-Stream*.

In Chapter 5 we have reported an experimental study comparing three very recent *Semi-streaming* algorithms for constructing spanners of unweighted graphs, all with optimal theoretical bounds. Our experiments indicate that all three algorithms are very efficient in practice, obtaining spanners of much higher quality than suggested by the worst-case theoretical bounds for all the test instances we have considered, and being competitive with off-line algorithms for the same problem, while enjoying a much wider range of applicability due to their greatly reduced memory requirements.

To conclude, we outline some open problems and directions for future research, which we consider a natural continuation of the work of this thesis:

- Our *multiple-sources shortest paths* algorithm takes $O((n \cdot \text{polylog}\, n)/\sqrt{s})$ passes, while the best known lower bound for this problem is $p = \Omega(n/s)$, derived from the lower bound for *undirected connectivity*. Obviously, it would be very interesting if we could narrow this gap, either by designing a more efficient algorithm, of by proving a tighter lower bound for this problem. Also, the algorithm presented in this thesis is a randomized montecarlo one, and a deterministic version would be of great interest.

- Addressing new graph problems, such as *depth first search*, which is considered hard in all the streaming models, including the (relatively) powerful *StreamSort* model.

- Devising general tradeoff algorithms for natural graph problems in *classical streaming*, or proving that they are not possible, would of course be of great theoretical and practical interest.

- Finally, much remains to be done in the experimental evaluation of streaming algorithms for graph problems, an area in which our investigation on *semi-streaming* algorithms for graph spanners constitutes merely an initial step.

Bibliography

[1] Experimental package. Available at `http://www.dis.uniroma1.it/~ribichini/spanner/`.

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

[3] Aggarwal, G., Datar, M., Rajagopalan, S., and Ruhl, M. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (2004).

[4] Albert, R. Scale-free networks in cell biology. *Journal of Cell Science 118* (2005), 4947–4957.

[5] Aleliunas, R., Karp, R., Lipton, R., Lovást, L., and Rackoff, C. Random walks, universal traversal sequences, and the complexity of maze problems. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)* (1979), pp. 218–223.

[6] Alon, N., Gibbons, P., Matias, Y., and Szegety, M. Tracking join and self join sizes in limited storage. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)* (1999), pp. 10–20.

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

[8] Alt, H., Geffert, V., and Mehlhorn, K. A lower bound for the nondeterministic space complexity of context free recognition. *Information Processing Letters 42* (1992), 25–27.

[9] Althofer, I., Das, G., Dobkin, D. P., Joseph, D., and Soares, J. On sparse spanners of weighted graphs. *Discrete & Computational Geometry 9* (1993), 81–100.

[10] Anderson, R., and Miller, G. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters 33*, 5 (1990), 269–273.

[11] Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the 21st Symposium on Principles of Database Systems (PODS'02)* (2002), pp. 221–232.

[12] Ausiello, G., Demetrescu, C., Franciosa, P., Italiano, G., and Ribichini, A. Small stretch spanners in the streaming model: new algorithms and experiments. In *Proc. 15th Annual European Symposium on Algorithms (ESA'07)* (2007), pp. 605–617.

[13] Awerbuch, B. Complexity of network synchronization. *Journal of the ACM 32*, 4 (Oct. 1985), 804–823.

[14] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. Models and issues in data stream systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS)* (2002), pp. 1–16.

[15] Babcock, B., Datar, M., and Motwani, R. Sampling from a moving window over streaming data. In *Proceedings of 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)* (2002).

[16] Bandelt, H.-J., and Dress, A. Reconstructing the shape of a tree from observed dissimilarity data. *Adv. Appl. Math. 7* (1986), 309–343.

[17] Bar-Yossef, Z., Jayram, T., Kumar, R., and Sivakumar, D. Information statistics approach to data stream and communication complexity. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'02)* (2002).

[18] Bar-Yossef, Z., Kumar, R., and Sivakumar, D. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002), San Francisco, California, pp. 623-632* (2002).

[19] Baswana, S. Dynamic algorithms for graph spanners. In *Algorithms - ESA 2006, 14th Annual European Symposium on Algorithms* (2006), pp. 76–87.

[20] Baswana, S. Faster streaming algorithms for graph spanners, 2006. `http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0611023`.

[21] Baswana, S., and Sen, S. A simple linear time algorithm for computing $(2k-1)$-spanner of $O(n^{1+1/k})$ size for weighted graphs. In *Proc. 30th Int. Coll. on Automata, Languages and Programming (ICALP)* (Berlin, 2003), vol. 2719 of *LNCS*, Springer, pp. 384–396.

[22] Beame, P., Jayram, T., and Rudra, A. Lower bounds for randomized read/write stream algorithms. In *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC'07)* (2007), pp. 689–698.

[23] Bhuvanagiri, L., Ganguly, S., Kesh, D., and Saha, C. Simpler algorithm for estimating frequency moments of data streams. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)* (2006), pp. 623–632.

[24] Blelloch, G., and Maggs, B. Parallel algorithms. In *The Computer Science and Engineering Handbook*. 1997, pp. 277–315.

[25] Bollobás, B. *Extremal graph theory*. Academic Press, 1978.

[26] Bondy, J., and Simonovits, M. Cycles of even length in graphs. *Journal of Combinatorial Theory, Series B, 16* (1974), 97–105.

[27] Broder, A., Karlin, A., Raghavan, P., and Upfal, E. Trading space for time in undirected *s-t* connectivity. *SIAM Journal on Computing 23* (1994), 324–334.

[28] Buriol, L., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohler, C. Counting triangles in data streams. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems (PODS'06)* (2006), pp. 253–262.

[29] Cai, L. NP-completeness of minimum spanner problems. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science 48*, 2 (1994), 187–194.

[30] Cai, L., and Keil, J. M. Degree-bounded spanners. *Parallel Processing Letters 3* (1993), 457–468.

[31] Chakrabarti, A., Khot, S., and Sun, X. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *Proceedings of the IEEE Conference on Computational Complexity* (2003), pp. 107–117.

[32] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining* (2004).

[33] Charikar, M., O'Callaghan, L., and Panigrahy, R. Better streaming algorithms for clustering problems. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03)* (2003).

[34] Chaudhuri, S., and Motwani, R. On sampling and relational operators. *IEEE Data Engineering Bulletin 22*, 4 (1999), 41–46.

[35] Chaudhuri, S., Motwani, R., and Narasayya, V. Random sampling for histogram construction: How much is enough? In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1998), pp. 436–447.

[36] Chaudhuri, S., Motwani, R., and Narasayya, V. On random sampling over joins. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1999), ACM Press, pp. 263–274.

[37] Chew, L. P. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences 39*, 2 (Oct. 1989), 205–219.

[38] Chiang, Y., Goodrich, M., Grove, E., Tamassia, R., Vemgroff, D., and Vitter, J. External-memory graph algorithms. In *Proc. 6th Annual ACM-SIAM Symposium on Dicrete Algorithms (SODA'95)* (1995), pp. 139–149.

[39] Coppersmith, D., and Kumar, R. An improved data stream algorithm for frequency moments. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Dicrete Algorithms (SODA'04)* (2004), pp. 151–156.

[40] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[41] Cormode, G., and Muthukrishnan, S. Estimating dominance norms on multiple data streams. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)* (2003), pp. 148–160.

[42] Das, G., and Joseph, D. Which triangulations approximate the complete graph? In *Proc. Int. Symp. on Optimal Algorithms* (Berlin, May 29–June 2 1989), H. Djidjev, Ed., vol. 401 of *LNCS*, Springer, pp. 168–192.

[43] Datar, M., Gionis, A., Indyk, P., and Motwani, R. Maintaining stream statistics over sliding windows. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2002).

[44] Demetrescu, C., Escoffier, B., Moruz, G., and Ribichini, A. Adapting parallel algorithms to the W-Stream model, with applications to graph problems. In *Proc. 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS'07)* (2007), pp. 194–205.

[45] Demetrescu, C., Finocchi, R., and Ribichini, A. Trading off space for passes in graph streaming problems. In *Proc. 17th Annual ACM-SIAM Symposium of Discrete Algorithms (SODA'06)* (2006), pp. 714–723.

[46] Demetrescu, C., and Italiano, G. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada* (2001), pp. 260–267.

[47] Dobkin, D., Friedman, S. J., and Supowit, K. J. Delaunay graphs are almost as good as complete graphs. *Discrete & Computational Geometry 5* (1990), 399–407. See also *28th Symp. Found. Comp. Sci.*, 1987, pp. 20–26.

[48] Elkin, M. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. In *Proc. of International Colloq. on Automata, Languages, and Programming, ICALP'07 Wroclaw, Poland* (2007), pp. 716–727.

[49] Erdös, P. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963), Publ. House Czechoslovak Acad. Sci., Prague* (1964), pp. 29–36.

[50] Erdös, P., and Rényi, A. On random graphs. *Publ. Math. Debrecen 6* (1959), 290–291.

[51] Faloutsos, M., Faloutsos, P., and Faloutsos, C. On power-law relationships of the internet topology. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1999), ACM Press, pp. 251–262.

[52] Feige, U. A spectrum of time-space trade-offs for undirected s-t connectivity. *Journal of Computer and System Sciences 54*, 2 (1997), 305–316.

[53] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. On graph problems in a semi-streaming model. In *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP)* (2004).

[54] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. Graph distances in the streaming model: the value of space. In *Proceedings of the 16th ACM/SIAM Symposium on Discrete Algorithms (SODA)* (2005), pp. 745–754.

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

[56] Frahlin, G., Indyk, P., and Sohler, C. Sampling in dynamic data streams and applications. In *Proceedings of the 21st ACM Symposium on Computational Geometry* (2005), pp. 79–88.

[57] Frahlin, G., and Sohler, C. Coresets in dynamic geometric data streams. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)* (2005).

[58] Gilbert, A., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC'02)* (2002), pp. 389–398.

[59] Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Quicksand: Quick summary and analysis of network data. Tech. rep., DIMACS Technical Report 2001-43, 2001.

[60] Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)* (2001), pp. 79–88.

[61] Gilbert, A., Kotidis, Y., Muthukrishnan, S., and Strauss, M. How to summarize the universe: dynamic maintenance of quantiles. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)* (2002), pp. 454–465.

[62] Golab, L., and Ozsu, M. Data stream management issues: a survey. Tech. rep., School of Computer Science, University of Waterloo, TR CS-2003-08, 2003.

[63] Greene, D., and Knuth, D. *Mathematics for the analysis of algorithms.* Birkhäuser, 1982.

[64] Grigni, M., and Sipser, M. Monotone separation of logarithmic space from logarithmic depth. *Journal of Computer and System Sciences 50* (1995), 433–437.

[65] Grohe, M., Hernich, A., and Schweikardt, N. Randomized computations on large data sets: tight lower bounds. In *Proc. 25th ACM Symposium on Principles of Database Systems (PODS '06)* (2006), ACM Press, pp. 243–252.

[66] Grohe, M., Koch, C., and Schweikardt, N. Tight lower bounds for query processing on streaming and external memory data. In *Proc. 32nd Int. Colloquium on Automata, Languages and Programming (ICALP'05)* (2005), vol. 3580 of *Lecture Notes in Computer Science*, pp. 1076–1088.

[67] Grohe, M., and Schweikardt, N. Lower bounds for sorting with few random accesses to external memory. In *Proc. 24th ACM Symposium on Principles of Database Systems (PODS '05)* (2005), ACM Press, pp. 238–249.

[68] Guha, S., Indyk, P., Muthukrishnan, S., and Strauss, M. Histogramming data streams with fast per-item processing. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP'02)* (2002), pp. 681–692.

[69] Guha, S., Koudas, N., and Shim, K. Data streams and histograms. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC'01)* (2001), pp. 471–475.

[70] Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. Clustering data streams. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)* (2000), pp. 359–366.

[71] Henzinger, M., and King, V. Fully dynamic biconnectivity and transitive closure. In *Proc. 36th IEEE Symposium on Foundations of Computer Science (FOCS'95)* (1995), pp. 664–672.

[72] Henzinger, M., Raghavan, P., and Rajagopalan, S. Computing on data streams. *In "External Memory algorithms", DIMACS series in Discrete Mathematics and Theoretical Computer Science 50* (1999), 107–118.

[73] Indyk, P. Algorithms for dynamic geometric problems over data streams. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC'04)* (2000), pp. 373–380.

[74] Indyk, P. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00)* (2000), pp. 189–197.

[75] Indyk, P., and Woodruff, D. Tight lower bounds for the distinct elements problem. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS'03)* (2003), pp. 283–288.

[76] Indyk, P., and Woodruff, D. Optimal approximations of the frequency moments of data streams. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05)* (2005), pp. 202–208.

[77] Jájá, J. *An introduction to parallel algorithms.* Addison-Wesley, 1992.

[78] Jowhari, H., and Ghodsi, M. New streaming algorithms for counting triangles in graphs. In *Proceedings of the 11th Annual International Conference on Computing and Combinatorics (COCOON'05)* (2005), pp. 710–716.

[79] King, V. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)* (1999), pp. 81–99.

[80] Kushilevitz, E., and Nisan, N. *Communication Complexity.* Cambridge Univ. Press, 1997.

[81] Liestman, A. L., and Shermer, T. Additive graph spanners. *Networks: An International Journal 23* (1993), 343–364.

[82] Liestman, A. L., and Shermer, T. Grid spanners. *Networks: An International Journal 23* (1993), 122–133.

[83] Luby, M. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing 15*, 4 (1986), 1036–1053.

[84] Matias, Y., Vitter, J., and Wang, M. Dynamic maintenance of wavelet-based histograms. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)* (San Francisco, CA, USA, 2000), Morgan Kaufmann Publishers Inc., pp. 101–110.

[85] McGregor, A. Finding graph matchings in data streams. In *Proc. 8th Int. Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'05)* (2005), pp. 170–181.

[86] Morris, R. Counting large numbers of events in small registers. *Commun. ACM 21*, 10 (1978), 840–842.

[87] Munro, I., and Paterson, M. Selection and sorting with limited storage. *Theoretical Computer Science 12* (1980), 315–323.

[88] Muthukrishnan, S. Data streams: algorithms and applications. Tech. rep., 2003. Available at `http://athos.rutgers.edu/~muthu/stream-1-1.ps`.

[89] Muthukrishnan, S., and Strauss, M. Maintenance of multidimensional histograms. In *Proceedings of the FSTTCS* (2003), pp. 352–362.

[90] Muthukrishnan, S., and Strauss, M. Rangesum histograms. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)* (2003).

[91] Muthukrishnan, S., and Strauss, M. Approximate histogram and wavelet summaries of streaming data. Tech. rep., DIMACS Technical Report 2004-52, 2004.

[92] Peleg, D., and Shäffer, A. Graph spanners. *Journal of Graph Theory 13* (1989), 99–116.

[93] Peleg, D., and Ullman, J. D. An optimal synchronizer for the hypercube. *SIAM Journal on Computing 18*, 4 (Aug. 1989), 740–747.

[94] Reif, J. Optimal parallel algorithms for integer sorting and graph connectivity. Tech. Rep. TR 08-85, Aiken Computation Laboratory, Harvard University, Cambridge, 1985.

[95] Richards, D., and Liestman, A. L. Degree-constrained pyramid spanners. *JPDC: Journal of Parallel and Distributed Computing 25* (1995), 1–6.

[96] Roditty, L., Thorup, M., and Zwick, U. Deterministic constructions of approximate distance oracles and spanners. In *Proc. 32nd Int. Coll. on Automata, Languages and Programming* (2005), pp. 261–272.

[97] Ruhl, M. *Efficient Algorithms for New Computational Models*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2003.

[98] Saks, M., and Sun, X. Space lower bounds for distance approximation in the data stream model. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02)* (2002), pp. 360–369.

[99] Shiloach, Y., and Vishkin, U. An $o(\log n)$ parallel connectivity algorithm. *J. Algorithms 31*, 1 (1982), 57–67.

[100] Sullivan, M., and Heybey, A. Tribeca: A system for managing large databases of network traffic. In *Proceedings USENIX Annual Technical Conference* (1998).

[101] Tarjan, R., and Vishkin, U. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science (FOCS'84)* (1984), pp. 12–20.

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

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

[104] Vitter, J. S. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1 (1985), 37–57.

[105] Zwick, U. Personal communication.

[106] Zwick, U. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM 49*, 3 (2002), 289–317.