Sublinear Time Algorithms for Graph Problems

Sixue Liu

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Robert E. Tarjan

September 2021

Abstract

Processing massive graphs is becoming a more-and-more essential task in big-data analysis nowadays. Many graph problems require running time linear in the input size on the traditional RAM model. But linear time is not good enough in many real-life applications. In this thesis, we focus on theoretical analysis of solving fundamental graph problems on practical big-data platforms such as streaming and parallel computation models, with an emphasis on time efficiency and simplicity.

In the first part of this thesis, we study the connected components problem in the parallel computation model. Our first result includes several simple parallel algorithms with a state-of-the-art running time, which are significantly easier to implement comparing to the existing ones. Our second result includes simpler algorithms for connected components and spanning forest that are faster than any previous algorithms when the graph has small diameter – which is usually true in many graphs generated from real-life instances – albeit in a much weaker computation model.

For the second part of this thesis, we study the graph matching problem. We first give a simple algorithm for approximate bipartite matching on streaming and massively parallel computation models that improve upon the existing ones on the number of passes or space usage. Then we present the first semi-streaming algorithm for exact maximum weight bipartite matching that breaks the linear-pass barrier whenever the graph is not super dense.

Our approaches not only leverage the combinatorial properties in the graph, but also reformulate the graph problem and use continuous optimization techniques. We hope the new general techniques developed in this thesis can find further applications in algorithm design emerged in big-data analysis.

Acknowledgements

First of all, I want to thank my advisor Bob Tarjan for his patience and support. I believe, working with a professor like Bob for four years is like a dream not only for me, but for most computer science students in the world. His enthusiasm for hard-core research has not aged a day. On the other hand, Bob gave me the most freedom in conducting research: he always let me choose the topic that interests me the most. I am grateful for every day in Princeton due to this.

I would like to thank Matt Weinberg, Sepehr Assadi, Bernard Chazelle, and Omri Weinstein for being a member of my FPO committee. Not only for the helpful feedback to my presentation and dissertation, but also for the lecture, discussion, or chat happened during these four years. Special thanks to Matt for helping me publishing my first single-author paper, to Sepehr for teaching me his knowledge in a new area, and to Omri for the constructive suggestions to my paper.

I would like to thank my coauthors. The discussion with Peilin Zhong is always nice: having someone from the same high school makes the theory life easier. I learned a lot from Sepehr's way to present ideas. I thank Hengjie Zhang and Zhao Song for endless discussions on my last project in Princeton.

I want to thank many people that I meet in Princeton. I thank Yuping Luo, Aarti Gupta, and other people for helping me settle down in Princeton. I thank Noga Alon and Huacheng Yu for many insightful discussions, which encourage me to solve even harder problems. I also want to thank Elaine Shi for hosting me in CMU for a wonderful semester.

I want to thank Andrew Yao and Periklis Papakonstantinou in Tsinghua for opening the gate of theoretical computer science for me.

Finally, I want to thank my parents for their unconditional love and thank my wife Angie for the years we have been together.

To my parents and my wife.

Contents

Chapter 1

Introduction

With the rapid growth of the internet, many optimization problems in recent big-data applications often exceed the capacity of traditional computing platforms. We study the theoretical foundation of big-data analysis by designing algorithms on modern computing platforms to solve large-scale optimization problems under various resource constraints, with an emphasis on time efficiency and simplicity.

The datasets in most applications are originated from or can be modeled as graphs, and this thesis focuses on solving graph problems. Usually, the graph is too large to be stored in a single memory, resulting in a very slow random access to the data. To capture this, one important computational model is *streaming*: we want to solve the problem using very limited space while minimizing the number of passes of scanning the entire data. Another framework to solve large-scale graph problems is to store the graph in a distributed manner and use the computational power of the distributed processors, which is called *parallel computation*. These two areas continue to be very active partly due to the success of the cloud computing platforms from Google, Microsoft, Amazon, and so on.

In this thesis, we focus on two fundamental graph problems: computing *connected components* and *matching*. We are interested in *sublinear-time* algorithms: computing connected components and other, more complicated problems on the classic RAM model require at least running time linear in the input graph size. (In the streaming model, sublinear-time refers to sublinear passes of scanning the graph edges, see the next subsection.) For each problem, we first demonstrate the beauty and power of parallel and streaming computation by giving new, simple, yet efficient algorithm for this problem. Indeed, these problems already have elegant solutions on the RAM model, and simplicity makes the algorithms more preferable in practice. After that, we present

more complicated algorithms for each problem in the same model with better running time.

The rest of the introduction is outlined as follows. In §1.1, we introduce the main computational models in this thesis. In §1.2, we summarize our contributions with pointer to each detailed chapter. In §1.3, we introduce the background for the connected components problem, on two major parallel computation models. In §1.4, we introduce the background for the graph matching problem in the streaming model, where both approximation and exact matching algorithms are mentioned.

1.1 Computational Models

Through out the thesis, we use n to denote the number of vertices in the input graph and m to denote the number of edges in the input graph. We use d to denote the maximal diameter of any component in the input graph. We use log to the base-2 logarithm. We use $\widetilde{O}(f(n))$ to denote $f(n) \cdot \text{poly}(\log n)$ for any function $f(n)$ of n.

We start with a model called COMBINING CRCW (concurrent read, concurrent write) PRAM (parallel random-access machine) [8]. Such a machine consists of a large common memory and a number of processors, each with a small private memory. In one step, each processor can do one unit of local computation, read one word of common memory, or write into one word of common memory. The processors operate in lockstep. Concurrent reads and writes are allowed, with write conflicts resolved in favor of the smallest value written. A weaker model is called ARBITRARY CRCW PRAM, where when concurrent writes to the same cell, an arbitrary one wins but we do not know which. We discuss other variants of the PRAM model in the next chapter. We measure the efficiency of an algorithm primarily by the number of concurrent steps and secondarily by the *work*, defined to be the number of steps times the number of processors.

Another major model we consider is the MPC (massively parallel computing) model [27]. This is a model of distributed computation based on the BSP (bulk synchronous parallel) model [159]. The MPC model is more powerful than our PRAM model but is a realistic model of cloud computing platforms. An MPC machine consists of a number of processors, each with a private memory. There is no common global memory; the processors communicate with each other by sending messages. Computation proceeds in globally synchronized steps. In one step, each processor receives the messages sent to it in the previous step, does some amount of local computation, and then sends a message or messages to one or more other processors. The total size of the messages sent or received for each processor in any step cannot exceed the space of the processor.

In streaming computation, we mainly focus on the *semi-streaming* model, where each edge is

revealed along with its weight one-by-one in an online fashion and according to an adversarial order, and the algorithm is allowed to make one or more passes over the stream using $\widetilde{O}(n)$ space. The order of edges in different passes *need not* to be the same.

1.2 Our Contributions

This thesis contains two parts, which are described below. A more detailed introduction for each part can be found in the next two subsections.

In Part I, we consider parallel algorithms for the connected components problem.

- In Chapter 2, we give several very simple connected components algorithms on the COMBINING CRCW PRAM and MPC model. Among our new algorithms, two of them have a running time of $O(\log n)$, which matches the best time bounds in the literature in terms of functions of n, but ours are *significantly simpler*. The other algorithms we proposed here have running time of $O(\log^2 n)$ or $O(d)$. All our algorithms are very easy to implement. The material in this chapter is based on the paper with Robert E. Tarjan [117].

- In Chapter 3, we have three new algorithms on an ARBITRARY CRCW PRAM. The first two algorithms are for connected components and spanning forest, respectively. Their running time are $O(\log d \cdot \log\log_{m/n} n)$ with probability approaching 1. The third algorithm for connected components runs in $O(\log d + \log\log_{m/n} n)$ time. Before our work, there are only MPC algorithms with the same time bounds. Our algorithms are *simpler* yet on a much *weaker* model. There are strong lower bounds on problems easier than connected components on the ARBITRARY CRCW PRAM, but not on an MPC. Additionally, the MPC model ignores the total work, our result on the more fine-grained PRAM model captures the inherent complexities of the problems. The material in this chapter is based on the paper with Peilin Zhong and Robert E. Tarjan [10].

In Part II, we consier streaming algorithms for the bipartite matching problem.

- In Chapter 4, we design a simple and generic auction algorithm that reduces the problem of finding a $(1-\epsilon)$-approximate bipartite matching to that of finding $O(1/\epsilon^2)$ *maximal* matchings in *adaptively* chosen subgraphs of the input. Despite its simplicity, this technique gives a powerful tool for boosting approximation ratio of algorithms for the bipartite matching problem from the 2-approximation of maximal matching to $(1-\epsilon)$-approximation in different settings.

For instance, we obtain the following algorithms for the bipartite matching problem as a corollary:

  - A deterministic $O(1/\epsilon^2)$-pass $O(n)$-space algorithm in the graph streaming model; this improves the pass complexity of the state-of-the-art algorithms by $O(\log\log(1/\epsilon))$ and the space complexity by $O(1/\epsilon)$ to achieve optimal space bounds with no dependence on ϵ.
  - A randomized $O(1/\epsilon^2 \cdot \log\log n)$-round $O(n)$ memory algorithm in the MPC model; the round-complexity of the algorithm improves upon the state-of-the-art by an $(1/\epsilon)^{O(1/\epsilon)}$ factor while maintaining the same memory per machine.

  The material in this chapter is based on the paper with Sepehr Assadi and Robert E. Tarjan [21].

- In Chapter 5, we present a new streaming algorithm for the maximum weight bipartite matching problem that uses $\widetilde{O}(n)$ space and $\widetilde{O}(\sqrt{m})$ passes, which breaks the n-pass barrier. All the previous streaming algorithms either require $\Omega(n \log n)$ passes or only find an approximate solution. As a byproduct, this also yields an $\widetilde{O}(\sqrt{m})$-pass semi-streaming algorithm for the minimum vertex cover problem in bipartite graph. Our streaming algorithm combines various ideas from different fields, most notably the construction of *space-efficient* interior point method (IPM), SDD system solvers, the isolation lemma, and LP duality. To the best of our knowledge, this is the first work that implements the SDD solvers and IPMs in the streaming model in $\widetilde{O}(n)$ spaces for graph matrices; previous IPM algorithms only focus on optimizing the running time, regardless of the space usage. The material in this chapter is based on the manuscript with Zhao Song and Hengjie Zhang [116].

1.3 Parallel Algorithms for Connected Components

Computing the connected components of an undirected graph is a fundamental problem in algorithmic graph theory, with many applications. Using graph search [152], one can find the connected components of an n-vertex, m-edge graph in $O(m)$ time, which is best possible for a sequential algorithm. But linear time is not fast enough for big-data applications in which the problem graph is of internet scale or even bigger. To find the components of such large graphs in practice requires the use of parallelism.

Beginning in the 1970's, theoreticians developed a series of more-and-more efficient parallel algorithms. Their model of computation was some variant of the PRAM (parallel random access machine) model. Shiloach and Vishkin [143] gave an $O(\log n)$-time PRAM algorithm in 1982. The algorithm was later simplified by Awerbuch and Shiloach [24]. These algorithms are deterministic and run on an ARBITRARY CRCW PRAM with $O(m)$ processors. There are simpler algorithms and algorithms that do less work but use randomization. Gazit [67] combined an elegant randomized algorithm of Reif [138] with graph size reduction to obtain an $O(\log n)$-time, $O(m/\log n)$-processor CRCW PRAM algorithm. This line of work culminated in the $O(\log n)$-time, $O(m/\log n)$-processor EREW (exclusive-read, exclusive-write) PRAM algorithms of Halperin and Zwick [79, 80], the second of which computes spanning trees of the components as well as the components themselves. On an EREW PRAM, finding connected components takes $\Omega(\log n)$ time [47], so these algorithms minimize both time and work. The $\Omega(\log n)$ lower bound also holds for the CREW (concurrent-read, exclusive-write) PRAM with randomization, a model slightly weaker than the CRCW PRAM [53]. For the PRIORITY CRCW PRAM (write resolution by processor priority), a time bound of $\Omega(\log n/\log\log n)$ holds if there are $\text{poly}(n)$ processors and unbounded space, or if there is $\text{poly}(n)$ space and any number of processors [25].

The Halperin-Zwick algorithms use sophisticated techniques. Practitioners charged with actually finding the connected components of huge graphs have implemented much simpler algorithms. Indeed, such simple algorithms often perform well in practice [71, 77, 85, 144, 151]. The computational power of current platforms and the characteristics of the problem graphs may partially explain such good performance. Current parallel computing platforms such as MapReduce, Hadoop, Spark, and others have capabilities significantly beyond those modeled by the PRAM [50]. A more powerful model, the MPC (massively parallel computing) model [27] is intended to capture these capabilities. In this model, each processor can have $\text{poly}(n)$ (typically sublinear in n) private memory, and the local computational power is unbounded. A PRAM algorithm can usually be simulated on an MPC with asymptotically the same round complexity, and it is widely believed that the MPC model admits algorithms faster than the PRAM model [99, 75, 23]. On the MPC model, and indeed on the weaker COMBINING CRCW PRAM model, we present very simple, practical algorithms that run in $O(\log n)$ time [117].

In practice, many graphs in applications have components of small diameter, perhaps polylogarithmic in n. This observations lead to the question of whether one can find connected components faster on graphs of small diameter, perhaps by exploiting the power of the MPC model. Andoni et al. [10] answered this question "yes" by giving an MPC algorithm that finds connected

components in $O(\log d \log \log_{m/n} n)$ time, where d is the largest diameter of a component. Very recently, this time bound was improved to $O(\log d + \log \log_{m/n} n)$ by Behnezhad et al. [28]. Both of these algorithms are complicated and use the extra power of the MPC model, in particular, the ability to sort and compute prefix sums in $O(1)$ communication rounds. These operations require $\Omega(\log n / \log \log n)$ time on a CRCW PRAM even with poly(n) processors [25].

1.4 Streaming Algorithms for Graph Matching

A *matching* in a graph $G = (V, E)$ is any set $M \subseteq E$ of edges that do not share any endpoints. Finding matchings - in particular in *bipartite* graphs, namely, the *bipartite matching* problem - has been studied extensively in the literature, starting as early as the work of König [105] over a century ago, and continues to be an excellent testbed for development of fundamental algorithmic tools and ideas [61, 3, 57, 72, 57, 91, 78, 54, 93, 54, 5, 13, 20, 22, 59, 30, 43]. Broadly speaking, there have been two conceptual approaches to designing efficient matching algorithms - the traditional one is to leverage the *combinatorial* properties in the graph [153]; the other is reformulating the graph problem as a specific linear program [49], and using *continuous optimization* techniques (e.g., interior point method [101] and SDD system solver [149]) to design efficient algorithms.

Many of the works in the literature consider the *semi-streaming* model, where each edge is revealed along with its weight one-by-one in an online fashion and according to an adversarial order, and the algorithm is allowed to make one or more passes over the stream using $\widetilde{O}(n)$ space. For algorithms that only make one pass over the edges stream, researchers make continuous progress on pushing the constant approximation ratio above $1/2$, which is under the assumption that the edges are arrived in a uniform random order [93, 13, 59, 30]. The random-order assumption makes the problem easier (at least algorithmically). A more general setting is multi-pass streaming with adversarial edge arriving. Under this setting, the first streaming algorithm that beats the $1/2$-approximation of bipartite cardinality matching is [61], giving a $2/3 \cdot (1 - \epsilon)$-approximation in $1/\epsilon \cdot \log(1/\epsilon)$ passes. The first to achieve a $(1-\epsilon)$-approximation is [124], which takes $(1/\epsilon)^{1/\epsilon}$ passes. Since then, there is a long line of research in proving upper bounds and lower bounds on the number of passes to compute a maximum matching in the streaming model [3, 57, 72, 57, 91, 54, 5, 20, 22, 21]. Notably, [3, 5] use linear programming and duality theory.

A natural idea to find an approximate matching is to iteratively sample a small subset of edges and use these edges to refine the current matching. These algorithms are called *sampling-based* algorithms. In [5], Ahn and Guha show that by adaptively sampling $\widetilde{O}(n)$ edges in each iteration,

one can either obtain a certificate that the sampled edges admit a desirable matching, or these edges can be used to refine the solution of a specific LP. The LP is a nonstandard relaxation of the matching problem, and will eventually be used to produce a good approximate matching. The algorithm of Ahn and Guha can compute a $(1-\epsilon)$-approximate matching for weighted (not necessarily bipartite) graph in $\widetilde{O}(1/\epsilon)$ passes and $\widetilde{O}(n \operatorname{poly}(1/\epsilon))$ space. However, the degree of $\operatorname{poly}(1/\epsilon)$ in the space usage can be very large, making their algorithm inapplicable for small (non-constant) $\epsilon = o(1/\log n)$ in the semi-streaming model.

Finding a $(1-\epsilon)$-approximate maximum matching with no space dependence on ϵ requires different methods. Inspired by the well-studied *water filling* process in online algorithms (see [52] and the references therein), Kapralov proposes an algorithm that generalizes the water filling process to multiple passes [91]. This algorithm works in the vertex arrival semi-streaming model, where a vertex and all of its incident edges arrive in the stream together. The observation is that the water filling from pass $(k-1)$ to pass k follows the same manner as that in the first pass (with a more careful double-counting method), then solving differential equations gives a $(1-1/\sqrt{2\pi k})$-approximate matching in k passes. Kapralov's algorithm removes the $\operatorname{poly}(\log n)$ factor in the number of passes comparing to [3], giving a $(1-\epsilon)$-approximate maximum matching in $O(1/\epsilon^2)$ passes, albeit in a stronger vertex arrival model.

Part I

Parallel Algorithms for Connected Components

Chapter 2

Simple Connected Components Algorithms

2.1 Algorithmic Framework

Given an undirected graph with vertex set $[n] = \{1, 2, ..., n\}$ and m edges, we wish to compute its connected components via a concurrent algorithm. More precisely, for each component we want to label all its vertices with a unique vertex in the component, so that two vertices are in the same component if and only if they have the same label. To state bounds simply, we assume $n > 2$ and $m > 0$. We denote an edge by the unordered pair of its ends.

As our computing model we use a COMBINING CRCW (concurrent read, concurrent write) PRAM (parallel random-access machine) [8]. Such a machine consists of a large common memory and a number of processors, each with a small private memory. In one step, each processor can do one unit of local computation, read one word of common memory, or write into one word of common memory. The processors operate in lockstep. Concurrent reads and writes are allowed, with write conflicts resolved in favor of the smallest value written. We discuss weaker variants of the PRAM model in §2.4. We measure the efficiency of an algorithm primarily by the number of concurrent steps and secondarily by the *work*, defined to be the number of steps times the number of processors.

Our algorithms are also easy to implement on the MPC (massively parallel computing) model [27]. This is a model of distributed computation based on the BSP (bulk synchronous parallel) model [159]. The MPC model is more powerful than our PRAM model but is a realistic model of cloud computing platforms. An MPC machine consists of a number of processors, each with a

private memory. There is no common global memory; the processors communicate with each other by sending messages. Computation proceeds in globally synchronized steps. In one step, each processor receives the messages sent to it in the previous step, does some amount of local computation, and then sends a message or messages to one or more other processors. The total size of the messages sent or received for each processor in any step cannot exceed the space of the processor.

The MPC model is quite powerful, but even in this model there is a non-constant conditional lower bound on the number of steps needed to compute connected components.

Theorem 2.1.1 ([28]). *Any MPC algorithm with $n^{1-\Omega(1)}$ space per processor that with high probability computes each connected component of any given graph with diameter $d \geq \log^2 n$ requires $\Omega(\log d)$ steps, unless the* 2-Cycle *conjecture is wrong.*[1]

In this chapter, we specialize the MPC model to the connected components problem as follows. There is one processor per edge and one per vertex. A processor can only send a message to another processor once it knows about that processor. Initially a vertex knows only about itself, and an edge knows only about its two ends. Thus in the first concurrent step only edges can send messages, and only to their ends. A vertex or edge knows about another vertex or edge once it has received a message containing the vertex or edge. This model ignores contention resulting from many messages being sent to the same processor, and it allows a processor to send many messages in one step.

It is easy to solve the problem in $O(\log d)$ steps if messages can be arbitrarily large: send each edge end to the other end, and then repeatedly send from each vertex all the vertices it knows to all its incident vertices [137]. If there is a large component, however, this algorithm is not practical, for at least two reasons: it requires huge memory at each vertex, and the number of messages sent in the last step can be quadratic in n. Hence we restrict the local memory of a vertex or edge to hold only a small constant number of vertices and edges. We also restrict messages to hold only a small constant number of vertices and edges, along with an indication of the message type, such as a label request or a label update. Our goal is a simple algorithm with a step bound of $O(\log n)$. (We discuss the harder goal of achieving a step bound of $O(\log d)$ in §2.4.)

We consider algorithms that maintain a label for each vertex u, initially u itself. The algorithm updates labels step-by-step until none changes, after which all vertices in a component have the same label, which is one of the vertices in the component. At any given time the current labels define a digraph (directed graph) of out-degree one whose arcs lead from vertices to their labels. We call this the *label digraph*. If these arcs form no cycles other than loops (arcs of the form (u, u)),

[1] The 2-Cycle conjecture [142, 23] asserts that any MPC algorithm that uses $n^{1-\Omega(1)}$ space per processor requires $\Omega(\log n)$ steps to distinguish one cycle of size n from two cycles of sizes $n/2$ with high probability.

then this digraph is a forest of trees rooted at the self-labeled vertices: the parent of u is its label unless this label is u, in which case u is a root. We call this the *label forest*. Each tree in the forest is a *label tree*.

All our algorithms maintain the label digraph as a forest; that is, they maintain acyclicity except for self-labels. (We know of only two previous algorithms that do not maintain the label digraph as a forest: see §2.4.) Henceforth we call the label of a vertex u its *parent* and denote it by $u.p$, and we call a label tree just a *tree*. A non-root vertex is a *child* of its parent. A vertex is a *leaf* if it is a child but it has no children of its own. A tree is *flat* if the root is the parent of every child in the tree, and is a *singleton* if the root is the only vertex. (Some authors call a flat tree a *star*.) The *depth* of a tree vertex x is the number of arcs on the path from x to the root of the tree: a root has depth zero, a child of a root has depth one. The *depth of a tree* is the maximum of the depths of its vertices.

When changing labels, a simple way to guarantee acyclicity is to replace a parent only by a smaller vertex. We call this *minimum labeling*. (An equivalent alternative, *maximum labeling*, is to replace a parent only by a larger vertex). A minimum labeling algorithm stops with each vertex labeled by the smallest vertex in its component. All our algorithms do minimum labeling. They also maintain the invariant that all vertices in a tree are in the same component, as do all algorithms known to us. That is, they never create a tree containing vertices from two or more components. Our specialization of the MPC model to the connected components problem maintains this invariant. At the end of the computation there is one flat tree per component, whose root is the minimum vertex in the component.

2.2 Algorithms

We consider algorithms that consist of initialization followed by a main loop that updates parents and repeats until no parent changes. Initialization consists of setting the parent of each vertex equal to itself. The following pseudocode does initialization:

```
initialize:
    for each vertex v do v.p = v
```

Since initialization is the same for all our algorithms, we omit it in the descriptions below and focus on the main loop. Each iteration of the loop does a *connect* step, which updates parents

using current edges, one or more *shortcut* steps, each of which updates parents using old parents, and possibly an *alter* step, which alters the edges of the graph. When discussing and analyzing our algorithms, we use the following terminology. By the *graph* we mean the input graph, or the graph formed from the input graph by the *alter* steps done so far, if the algorithm does *alter* steps. An *edge* is an edge of the graph; a *component* is a connected component of the graph. All *alter* steps preserve components. By the *forest* we mean the forest whose child-parent arcs are the pairs $(v, v.p)$ with $v \neq v.p$; a *tree* is a tree in this forest. The terms *parent*, *child*, *ancestor*, *descendant*, *root*, *leaf* refer to the forest.

Our algorithms maintain the following *connectivity invariant*: v and $v.p$ are in the same component, as are v and w if $\{v, w\}$ is an edge. If $\{v, w\}$ is an edge, replacing the parent of v by any ancestor of w preserves the invariant, as does replacing the parent of w by any ancestor of v. This gives us many ways of doing a connect step. We focus on two. The first is *direct-connect*, which for each edge $\{v, w\}$, uses the minimum of v and w as a candidate for the new parent of the other. The other is *parent-connect*, which uses the minimum of the old parents of v and w as a candidate for the new parent of the old parent of the other. We express these methods in pseudocode below. We have written all our pseudocode so that it is correct and produces unambiguous results even if the loops run sequentially rather than concurrently and the vertices and edges are processed in arbitrary order. We say more about this issue below.

```
direct-connect:
    for each edge {v, w} do
        if v > w then
            v.p = min{v.p, w}
        else w.p = min{w.p, v}
```

The pseudocode for *parent-connect* begins by computing $v.o$, the old parent of v, for each vertex v. It then uses these old parents to compute the new parent $v.p$ of each vertex v. The new parent of a vertex x is the minimum $w.o < x.o$ such that there is an edge $\{v, w\}$ with $v.o = x$, if there is such an edge; if not, the parent of x does not change.

```
parent-connect:
    for each vertex v do
        v.o = v.p
    for each edge {v, w} do
        if v.o > w.o then
            v.o.p = min{v.o.p, w.o}
        else w.o.p = min{w.o.p, v.o}
```

If all reads and comparisons occur before all writes, and all writes occur concurrently, the following simpler pseudocode has the same semantics as that for *parent-connect*:

```
for each edge {v, w} do
    if v.p > w.p then
        v.p.p = min{v.p.p, w.p}
    else w.p.p = min{w.p.p, v.p}
```

On the other hand, if this simpler loop is executed sequentially, then in general the results depend on the order in which the edges are processed. Suppose for example there are two edges $\{x, y\}$ and $\{v, w\}$ such that $x.p = v$ and $v.p = z$. In *parent-connect*, $w.p$ is a candidate to be the new parent of z. That is, after the connect, the new parent of z will be no greater than the old parent of w. But in the simpler loop, if $\{x, y\}$ is processed before $\{v, w\}$, the processing of $\{x, y\}$ might change the parent of v to a vertex other than z, thereby making $w.p$ no longer a candidate for the new parent of z. Even though we are primarily interested in global concurrency, we want our algorithms and bounds to be correct in the more realistic setting in which the edges are processed one group at a time, with the group size determined by the number of available processors.

On a COMBINING CRCW PRAM, we can use the simple loop for *parent-connect*, since there is global concurrency. Each processor for an edge $\{v, w\}$ reads $v.p$ and $w.p$. If $v.p > w.p$, it reads $v.p.p$, tests if $w.p < v.p.p$; and, if so, writes $w.p$ to $v.p.p$; if $v.p \leq w.p$, it reads $w.p.p$, tests if $v.p < w.p.p$; and, if so, writes $v.p$ to $w.p.p$. All the reads occur before all the writes, and all the writes occur concurrently, with a write of smallest value succeeding if there is a conflict.

In the MPC model, each vertex stores its parent. To execute *parent-connect*, each processor for an edge $\{v, w\}$ requests $v.p$ and $w.p$. If $v.p > w.p$, it sends $w.p$ to $v.p$; otherwise, it sends $v.p$ to $w.p$. Each vertex then updates its parent to be the minimum of its old value and the smallest of

the received values. All our other loops can be similarly implemented on a COMBINING CRCW PRAM or in the MPC model.

A connect step of either kind can move a subtree from one tree to another. We can prevent this by restricting connection so that it only updates parents of roots. The following pseudocode implements such restrictions of *direct-connect* and *parent-connect*, which we call *direct-root-connect* and *parent-root-connect*, respectively:

direct-root-connect:

for each vertex v **do**

$v.o = v.p$

for each edge $\{v, w\}$ **do**

if $v > w$ and $v = v.o$ **then**

$v.p = \min\{v.p, w\}$

else if $w = w.o$ **then**

$w.p = \min\{w.p, v\}$

parent-root-connect:

for each vertex v **do**

$v.o = v.p$

for each edge $\{v, w\}$ **do**

if $v.o > w.o$ and $v.o = v.o.o$ **then**

$v.o.p = \min\{v.o.p, w.o\}$

else if $w.o = w.o.o$ **then**

$w.o.p = \min\{w.o.p, v.o\}$

In *direct-root-connect* (as in *parent-connect*) we need to save the old parents to get a correct sequential implementation, so that the root test is correct even if the parent has been changed by processing another edge during the same iteration of the loop over the edges. If we truly have global concurrency, simpler pseudocode suffices, as for *parent-connect*.

Shortcutting is the key to obtaining a logarithmic step bound. Shortcutting replaces the parent of each vertex by its grandparent. The following pseudocode implements shortcutting:

```
shortcut:
    for each vertex v do
        v.o = v.p
    for each vertex v do
        v.p = v.o.o
```

In the case of *shortcut*, the simpler loop "for each vertex v do $v.p = v.p.p$" produces correct results and preserves our time bounds, even though sequential execution of the code produces different results depending on the order in which the vertices are processed. All we need is that the new parent of a vertex is no greater than its old grandparent. Thus the simpler loop might well be a better choice in practice.

Edge alteration deletes each edge $\{v, w\}$ and replaces it by $\{v.p, w.p\}$ if $v.p \neq w.p$. The following pseudocode implements alteration:

```
alter:
    for each edge {v, w} do
        if v.p = w.p then
            delete {v, w}
        else replace {v, w} by {v.p, w.p}
```

We shall study in detail four algorithms, whose main loops are given below:

Algorithm S: **repeat** {*parent-connect*; **repeat** *shortcut* **until** no $v.p$ changes} **until** no $v.p$ changes

Algorithm R: **repeat** {*parent-root-connect*; *shortcut*} **until** no $v.p$ changes

Algorithm RA: **repeat** {*direct-root-connect*; *shortcut*; *alter*} **until** no $v.p$ changes

Algorithm A: **repeat** {*direct-connect*; *shortcut*; *alter*} **until** no $v.p$ changes

In algorithm S, the inner loop "**repeat** *shortcut* **until** no $v.p$ changes" terminates after a shortcut that changes no parents. Similarly, in each of the algorithms the outer **repeat** loop terminates after

an iteration that changes no parents.

Many algorithms fall within our framework. We focus on these four because they are simple and natural and we can prove good bounds for them. Algorithm S simplifies the algorithm of Hirschberg, Chandra, and Sarwate [82]. Theirs was the first algorithm to run in a polylogarithmic number of steps, specifically $O(\log^2 n)$. Algorithm R simplifies the algorithm of Shiloach and Vishkin, which runs in $O(\log n)$ steps. We discuss in detail the relationship of our algorithms to theirs, as well as other related work, in §2.4.

Some observations guided our choice of algorithms to study. There is a tension between connect steps and shortcut steps: the former combine trees but generally produce deeper trees; the latter make trees shallower. Algorithm S completely flattens all the existing trees between each connect step. This guarantees monotonicity: the *parent-connect* steps in S change the parents only of roots. As we shall see, completely flattening the trees also guarantees that each tree is connected to another tree in at most two iterations of the outer loop. Algorithm S has an $O(\log^2 n)$ step bound, one log factor coming from the at most $\log n$ iterations of the inner loop needed to flatten all trees, the other coming from the at most $2 \log n$ iterations of the outer loop needed to reduce the number of trees in each component to 1. Unfortunately, the $O(\log^2 n)$ bound is tight to within a constant factor.

To obtain a step bound of $\log n$, we must reduce the number of shortcuts per round to $O(1)$. Algorithm R does this. It uses *root-connect*, making it monotonic. Algorithm RA is a related algorithm that does edge alteration, allowing it to use *direct-root-connect* instead of the more-complicated *root-connect*. Both R and RA have an $O(\log n)$ step bound.

It is natural to wonder whether monotonicity is needed to get a polylogarithmic step bound. Algorithm A answers this question negatively. It is algorithm RA with *direct-connect* replacing *direct-root-connect*. We shall prove an $O(\log^2 n)$ step bound for algorithm A. We do not know if this bound is tight.

Each of our four algorithms is distinct: for each pair of algorithms, there is a graph on which the two algorithms make different parent changes, as one can verify case-by-case. Use of *direct-connect* or *direct-root-connect* requires edge alteration to obtain a correct algorithm. Although different, algorithms R and RA behave similarly, and we use the same techniques to obtain bounds for both of them. Algorithm S is equivalent to the algorithm formed by replacing *parent-connect* by *parent-root-connect*, and to the algorithm formed by replacing *parent-connect* by *direct-connect* and adding *alter* to the end of the main loop: all three algorithms make the same parent changes.

We conclude this section with a proof that our algorithms are correct. We begin by establishing some properties of edge alteration. We call an iteration of the main loop (the outer loop in algorithm

S) a *round*. We call a vertex *bare* if it is not an edge end and *clad* if it is. Only alter steps can change vertices from clad to bare or vice-versa. A clad vertex becomes bare if it loses all its incident edges during an alter step. A bare leaf cannot become clad, nor can a bare vertex all of whose descendants are bare, because no connect step can give it a new (clad) descendant.

To prove correctness, we need the following key result.

Lemma 2.2.1. *After k rounds of algorithm A or RA, each vertex that is clad or a root has a path in the graph to the smallest vertex in its component. If the algorithm is A, there is such a path with at most* $\max\{0, d-k\}$ *edges.*

Proof. The proof is by induction on k. The lemma is true for $k = 0$. Let $k > 0$, let x be a clad vertex or a root at the end of round k, and let w be the smallest vertex in the same component as x. If $x = w$, the lemma holds for x and k. Suppose $x > w$. If x is a root just before the alteration in round k, let $u = x$; otherwise, let u be a vertex such that $u.p = x$ and u must have been clad just before the alteration in round k. Such a u must exist since x is clad at the end of round k. Since $u \geq x > w$, $u \neq w$. By the induction hypothesis, there is a path in the graph from u to w at the beginning of round k, and if the algorithm is A this path contains at most $\max\{0, d-k+1\}$ edges. Let $\{v, w\}$ be the last edge on this path. The alteration in round k converts this path to a path from x to w. Since $\{v, w\}$ exists at the beginning of round k, $v.p = w$ after the connect in round k. Thus the alter in round k deletes $\{v, w\}$, so if the algorithm is A, the path from x to w contains at most $\max\{0, d-k\}$ edges. □

Lemma 2.2.2. *In algorithms A and RA,* (i) *an alter makes all leaves bare;* (ii) *once a vertex is a leaf, it stays a leaf;* (iii) *once a leaf is bare, it stays bare; and* (iv) *every non-root grandparent is clad.*

Proof. Part (i) is immediate from the definition of alter. A vertex becomes a leaf either in a connect or in a shortcut. A shortcut does not make a leaf into a non-leaf. Any leaf existing at the beginning of a connect is bare by (i) and hence cannot be made a non-leaf by the connect. Neither a connect nor a shortcut can make a bare vertex clad. Thus (ii) and (iii) hold.

We prove (iv) by induction on the number of steps. No vertex is a grandparent initially, so (iv) holds initially. Let x be a vertex that is not a non-root grandparent before a connect; that is, x is either a root or has no grandchildren. If x acquires a parent or child during the connect, then it must be clad, since *direct-connect* only changes parents and children of clad vertices, so the lemma holds for x after the connect. Since all children of x (if any) are leaves, they cannot become non-leaves during the connect by (i), so x cannot become a grandparent as a result of one of its children

acquiring a child. Thus the connect preserves (iv). A vertex that is a non-root grandparent after a shortcut is also a non-root grandparent before the shortcut, so a shortcut preserves (iv). Suppose x is a non-root grandparent before an alter. Then x is a great-great grandparent before the shortcut preceding the alter, so it has a non-root grandchild y that is clad. The shortcut makes y a child of x. By Lemma 2.2.1 there is a path in the graph from y to the smallest vertex in its component. The alter transforms this path into a path from x to the smallest vertex. Since x is not a root, it is not the smallest vertex, so the path from x contains at least one edge, making x clad. □

Lemma 2.2.3. *In all our algorithms, a leaf stays a leaf.*

Proof. For A or RA, this is part (iii) of Lemma 2.2.2. For the other algorithms, a connect or shortcut cannot make a leaf into a parent. □

Theorem 2.2.4. *All our algorithms are correct.*

Proof. For each algorithm, a proof by induction on the number of parent changes and alterations (if any) shows that v and $v.p$ are in the same component of the input graph for each vertex v, as are v and w for each edge $\{v, w\}$. Thus all vertices in any tree are in the same component.

Parents only decrease, so the parent function always defines a forest. There are at most $n(n-1)/2$ parent changes. In the following paragraph, we prove that there is at least one parent change in any round except the last one, so each algorithm terminates in at most $n(n-1)/2+1$ rounds.

Consider the state just before a round. If there is at least one non-flat tree, then either the connect step or the first shortcut step will change a parent. Suppose all trees are flat but the vertices in some component are in two or more trees. Let T be the tree containing the smallest vertex in the component and T' another tree in the component. It is immediate for algorithms S and R and follows from Lemma 2.2.1 for A and RA that there is a path in the graph from the root of T' to the root of T. Thus there is an edge $\{v, w\}$ with w but not v in T. Since all trees are flat, $v.p$ and $w.p$ are roots. In algorithms A and RA, v and w are roots by part (i) of Lemma 2.2.2. In algorithms S and R, the connect step will make $w.p$ the parent of $v.p$; in algorithms A and RA, the connect step will make w the parent of v. We conclude that the algorithm can only stop when all trees are flat and there is one tree per component. □

2.3 Efficiency

On a graph that consists of a path of n vertices, all our algorithms take $\Omega(\log n)$ steps, since this graph has one component of diameter n, and the general lower bound of Theorem 2.1.1 applies. We

prove worst-case step bounds of $O(\min\{d, \log n\} \log n)$ for S, $O(\min\{d, \log^2 n\})$ for A, and $O(\log n)$ for R and RA. We also show that algorithm S can take $\Omega(\log^2 n)$ steps, and algorithms R and RA can take $\Omega(\log n)$ steps even on graphs with constant diameter d.

2.3.1 Analysis of S

Theorem 2.3.1. *Algorithm S takes $O(\min\{d, \log n\} \log n)$ steps.*

Proof. Since the maximum depth of any tree is $n-1$, and a shortcut reduces the depth of a depth-k tree to $\lceil k/2 \rceil$, the inner loop stops in $O(\log n)$ iterations. Let u be the smallest vertex in some component. We prove by induction on k that after k rounds every vertex in the component at distance k or less from vertex u has parent u, which implies that the algorithm stops in at most $d+1$ rounds. This is true initially. Let v be a vertex at distance k from u. If $v.p \neq u$ just before round k, there is an edge $\{v, w\}$ such that $w.p = u$ by the induction hypothesis. After the connect step in round k, $v.p = u$.

To obtain an $O(\log n)$ bound on the number of rounds, we prove that if there are two or more trees in a component, two rounds reduce the number of such trees by at least a factor of two. Call a root *minimal* if edges incident to its tree connect it only with trees having higher roots. A connect step makes each non-minimal root into a non-root. Let x be a minimal root, and let $\{v, w\}$ be an edge with v in the tree rooted at x and w in a tree rooted at $y > x$. If the connect step in the round does not make y a child of x, then $y.p < x$ after the round, causing x to become a non-root in the next round.

Suppose there are $k > 1$ roots in a component at the beginning of a round. Among the minimal roots, suppose there are j that get a new child as a result of the connect step in the round. At least $\max\{k-j, j\} \geq k/2$ roots are non-roots after two rounds. □

We show by example that S can take $\Omega(\log^2 n)$ steps.

If there is a tree of depth 2^k just before the inner loop in a round, this loop will take at least k steps. If every round produces a *new* tree of depth 2^k, and the algorithm takes k rounds, the total number of steps will be $\Omega(k^2)$. Our bad example is based on this observation. It consists of k components such that running algorithm S for i rounds on the i-th component produces a tree that contains a path of length 2^k.

At the beginning of a round of the algorithm, we define the *implied graph* to be the graph whose vertices are the roots and whose edges are the pairs $\{v.p, w.p\}$ such that $\{v, w\}$ is a graph edge and $v.p \neq w.p$. Restricted to the roots, the subsequent behavior of algorithm S on the original graph is

the same as its behavior on the implied graph. We describe a way to produce a given implied graph in a given number of rounds.

To produce a given implied graph G with vertex set $[n]$ in one round, we start with the *generator* $g(G)$ of G, defined to be the graph with vertex set $[2n]$, edges $\{v, v+n\}$ and $\{v+n, v\}$ for each $v \in [n]$, and an edge $\{v+n, w+n\}$ for each edge $\{v, w\}$ in G. A round of algorithm S on $g(G)$ does the following: the connect step makes $v+n$ a child of v for $v \in [n]$, and the shortcuts do nothing. The resulting implied graph has vertex set $[n]$ and an edge $\{v, w\}$ for each edge $\{v, w\}$ in G; that is, it is G.

To produce a given implied graph G in i rounds, we start with $g^i(G)$. An induction on the number of rounds shows that after i rounds of algorithm S, G is the implied graph.

Let k be a positive integer, and let P be the path of vertices $1, 2, ..., 2^k + 1$ and edges $\{i, i+1\}$ for $i \in [2^k]$. Our bad example is the disjoint union of $P, g(P), g^2(P), ..., g^{k-1}(P)$, with the vertices renumbered so that each component has distinct vertices and the order within each component is preserved. Algorithm S takes $\Omega(k^2)$ steps on this graph. The number of vertices is $n = (2^k+1)(2^k - 1) = 2^{2k} - 1$. The number of edges is $2^{2k} - k - 1$, since the graph is a set of k trees. Thus the number of steps is $\Omega(\log^2 n)$.

2.3.2 Analysis of A

In analyzing A, R, and RA, we assume that the graph is connected: this is without loss of generality since A, R, and RA operate independently and concurrently on each component, and each round does $O(1)$ steps.

Theorem 2.3.2. *Algorithm A takes $O(d)$ steps.*

Proof. By Lemma 2.2.1, after d rounds there are no edges and only one tree. By Lemma 2.2.2, this tree has depth at most two. Thus the algorithm stops after at most $d + 2$ rounds. □

A simple example shows that Theorem 2.3.2 is false for S, R, and RA. Consider the graph whose edges are $\{i, i+1\}$ for $i \in [n-1]$ and $\{i, n\}$ for $i \in [n-1]$. After the first connection step, there is one tree: 1 is the parent of 2 and n, and i is the parent of $i+1$ for $i \in [2, n-2]$. Subsequent connection steps do nothing; the tree only becomes flat after $\Omega(\log n)$ shortcuts.

To obtain an $O(\log^2 n)$ step bound for algorithm A, we show that $O(\log n)$ rounds reduce the number of non-leaf vertices by at least a factor of 2, from which an overall $O(\log^2 n)$ step bound follows.

It is convenient to shift our attention from rounds to passes. A *pass* is the interval from the beginning of one shortcut to the beginning of the next. Pass 1 begins with the shortcut in round 1 and ends with the connect in round 2. We need one additional definition. A vertex is *deep* if it is a non-root with at least one child and all its children are leaves.

Lemma 2.3.3. *A vertex that is deep at the beginning of a pass is a leaf at the end of the pass.*

Proof. Let x be a vertex that is deep at the beginning of a pass. The shortcut in the pass makes x a leaf. Once a vertex is a leaf, it stays a leaf by Lemma 2.2.2. □

Lemma 2.3.4. *Suppose there are at least two roots at the beginning of a pass, and that x is a root all of whose children are bare leaves. Then x is not a root after the pass.*

Proof. By Lemma 2.2.1, at the beginning of the pass there is an edge $\{v, w\}$ with v but not w in the tree with root x. Since all children of x are bare leaves, $v = x$. The edge $\{x, w\}$ existed at the beginning of the connect just before the pass. Since this connect did not make x a non-root, $w > x$, and since w is not a child of x after the connect, $w.p < x$ after it. The alter in the pass replaces $\{x, w\}$ by $\{x, w.p\}$. The connect in the pass then makes x a non-root. □

We need one more idea, which we borrow from the analysis of *path halving*, a method used in disjoint set union algorithms that shortcuts a single path [154]. For any vertex v, we define the *level* of v to be $v.l = \lfloor \log(v - v.p) \rfloor$ unless $v.p = v$, in which case $v.l = 0$. The level of a vertex is at least 0, less than $\log n$, and non-decreasing. The following lemma quantifies the effect of a shortcut on a sufficiently long path in a tree.

Lemma 2.3.5. *Assume $n \geq 4$. Consider a tree path P of $k \geq 4 \log n$ vertices. A shortcut increases the sum of the levels of the vertices on P by at least $k/4$.*

Proof. Let u, v, and w be three consecutive vertices on P, with v the parent of u and w the parent of v. Let i and j be the levels of u and v, respectively. A shortcut increases the level of u from i to $\lfloor \log(u - w) \rfloor = \lfloor \log(u - v + v - w) \rfloor \geq \lfloor \log(2^i + 2^j) \rfloor$. If $i < j$, this increases the level of u by at least $j - i$; if $i = j$, it increases the level of u by one.

Let $x_1, x_2, ..., x_{k-1}$ be the vertices on P from largest to smallest (deepest to shallowest), excluding the last one. For each $i \in [k-2]$, let $\Delta_i = x_{i+1}.l - x_i.l$. The sum of the Δ_i's is $\Sigma = x_{k-1}.l - x_1.l \geq -\log n$ since the sum telescopes. Let k_+, k_0, and k_-, respectively, be the number of positive, zero, and negative Δ_i's, and let Σ_+ and Σ_- be the sum of the positive Δ_i's and the sum of the negative Δ_i's, respectively. By the previous paragraph, the sum of the levels of the vertices on P increases by at least $\Sigma_+ + k_0$.

From $\Sigma = \Sigma_+ + \Sigma_-$ we obtain $\Sigma_+ \leq -\Sigma_- - \log n$. Since the Δ_i's are integers, $\Sigma_+ \geq k_+$ and $-\Sigma_- \geq k_-$. Thus $2(\Sigma_+ + k_0) \geq 2\Sigma_+ + k_0 \geq k_+ + k_0 + k_- - \log n = k - 2 - \log n \geq k/2$, since $n \geq 4$ implies $k/2 \geq 2\log n \geq \log n + 2$. Dividing by two gives $\Sigma_+ + k_0 \geq k/4$. □

We combine Lemmas 2.3.3, 2.3.4, and 2.3.5 to obtain the desired result.

Lemma 2.3.6. *Assume $n \geq 4$. Suppose that at the beginning of pass i there are at least two roots and more than $k/2$ but at most k non-leaves. After $O(\log n)$ passes there are at most $k/2$ non-leaves or at most one root.*

Proof. Assume the hypotheses of the lemma are true. Call a vertex *fresh* if it is a non-leaf at the beginning of pass i and *stale* otherwise. After pass i, all the stale leaves are bare by Lemma 2.2.2. Suppose the hypotheses of the lemma hold at the beginning of some pass after pass i. At least one of the following four cases occurs:

1. There are at least $k/8$ clad leaves. One pass makes all clad leaves bare by Lemma 2.2.2. Since each clad leaf is fresh and can only become bare once, there are at most $k/(k/8) = 8$ passes in which this case can occur.

2. There are at least $k/8$ roots of flat trees, all of whose children are bare. One pass makes all but one such roots non-roots by Lemma 2.3.4. There are at most $k/(k/8) = 8$ passes in which this case can occur.

3. There are at least $k/(32\log n)$ deep vertices. This pass makes all these vertices into leaves by Lemma 2.3.3. There are at most $k/(k/(32\log n)) = 32\log n$ passes in which this case can occur.

4. None of the first three cases occurs. Since Cases 1 and 2 do not occur, there are at most $k/4$ roots of flat trees. Thus there are at least $k/4$ non-leaves in non-flat trees. Since Case 3 does not occur, there are at most $k/(32\log n)$ deep vertices. From each of these deep vertices there is a tree path to a root. Every non-leaf in a non-flat tree is on one or more such paths. Find a deep vertex whose tree path is longest, and delete this path. This may break the tree containing the path into several trees, but this does not matter: it does not increase the number of deep vertices, although it may convert some deep vertices into roots, which only improves the bound. Repeat this processor until at least $k/8$ non-leaves are on deleted paths. Each path contains at least $(k/8)/(k/(32\log n)) = 4\log n$ vertices. By Lemma 2.3.5, the shortcut increases the sum of the levels of the vertices on these paths by at least $k/32$. There are at most $(k\log n)/(k/32) = 32\log n$ passes in which this case can occur.

We conclude that after at most $16 + 64 \log n$ passes, either there are at most $k/2$ non-leaves or at most one root. □

Theorem 2.3.7. *Algorithm A takes $O(\log^2 n)$ steps.*

Proof. The theorem is immediate from Lemma 2.3.6, since once there is a single root the algorithm stops after $O(\log n)$ steps. □

We do not know whether the bound in Theorem 2.3.7 is tight. We conjecture that it is not, and that algorithm A takes $O(\log n)$ steps. We *are* able to prove an $O(\log n)$ bound for the monotone algorithms R and RA, which we do in the next section.

An algorithm similar to algorithm A is algorithm P, which replaces *direct-connect* in A by *parent-connect* and deletes *alter*. The following pseudocode implements the main loop of this algorithm:

Algorithm P: **repeat** {*parent-connect*; *shortcut*} **until** no $v.p$ changes

We conjecture that algorithm P, too, has an $O(\log n)$ step bound, but we are unable to prove even an $O(\log^2 n)$ bound, the problem being that this algorithm can leave flat trees unchanged for a non-constant number of rounds.

2.3.3 Analysis of R and RA

Algorithms R and RA can also leave flat trees unchanged for a non-constant number of rounds, so the analysis of §2.3.2 also fails for these algorithms. But we can obtain an even better bound than that of Theorem 2.3.7 by using a different analytical technique, that of Awerbuch and Shiloach [24], extended to cover a constant number of rounds rather than just one. This analysis requires the algorithm to be monotonic.

We call a tree *passive* in a round if it exists both at the beginning and at the end of the round; that is, the round does not change it. A passive tree is flat, but a flat tree need not to be passive. We call a tree *active* in a round if it exists at the end of the round but not at the beginning. An active tree contains at least two vertices and has depth at least one.

We say a connect *links* trees T and T' if it makes the root of one of them a child of a vertex in the other. If the connect makes the root of T a child of a vertex in T', we say the connect *links T to T'*.

Lemma 2.3.8. *If trees T and T' are passive in round k, then there is no edge with one end in T and the other end in T', and the connect in round $k+1$ does not link T and T'.*

Proof. If T and T' were linked in round $k+1$, there would be an edge connecting them that caused the link. In algorithm RA, Lemma 2.2.2 implies that any such edge connects the roots of T and T' at the beginning of round k. Thus in either algorithm T and T' would have been linked in round k, contradicting their passivity in round k. □

If T exists at the end of round k, its *constituent trees* at the end of round $j \leq k$ are the trees existing at the end of round j whose vertices are in T. Since algorithms R and RA are monotone, these trees partition the vertices of T.

Lemma 2.3.9. *Let T be an active tree in round k. Then for $j \leq k$ at least one of the constituent trees of T in round j is active.*

Proof. The proof is by induction on j for j decreasing. The lemma holds for $j = k$ by assumption. Suppose it holds for $j > 0$. If the constituent trees of T in round $j-1$ were all passive, the connect step in round j would change none of them by Lemma 2.3.8. Neither would the shortcut in round j, contradicting the existence of an active constituent tree in round j. □

Since algorithm RA is slightly simpler to analyze than R, we first analyze RA, and then discuss the changes needed to make the analysis apply to R. We measure progress using the potential function of Awerbuch and Shiloach, modified so that it is non-increasing and passive trees have zero potential. In RA we define the *individual potential* of a tree T at the end of round k to be zero if T is passive in round k, or two plus the maximum of zero and the maximum depth of an arc end in T if T is active in round k. If T exists at the end of round k and $j \leq k$, we define the *potential* $\Phi_j(T)$ of T at the end of round j to be the sum of the individual potentials of its constituent trees at the end of round j. We define the *total potential* at the end of round k to be the sum of the potentials of the trees existing at the end of round k.

We shall prove that the total potential decreases by a constant factor in a constant number of rounds. This will give us an $O(\log n)$ bound on the number of rounds. It suffices to consider each active tree individually.

Lemma 2.3.10. *Let T be active in round $k > 1$ of RA. Then $\Phi_{k-1}(T) \geq \Phi_k(T)$. If $\Phi_{k-1}(T) \geq 4$ then $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$.*

Proof. Let $t \geq 1$ be the number of active constituent trees of T in round $k-1$, and let $\ell = \Phi_{k-1}(T) - 2t$. Then $\ell \geq 0$. Consider the tree S formed from the constituent trees of T in round $k-1$ by the connect step in round k. The shortcut in round k transforms S into T. By Lemma 2.3.8, along any path in S there cannot be consecutive vertices from two different passive trees. By Lemma 2.2.2,

at the beginning of round k no edge end is a leaf, so the deepest edge end in S has depth at most $\ell+2t$: its path to the root contains at most $\ell+t$ vertices in active constituent trees and at most $t+1$ vertices (all roots) in passive constituent trees. The shortcut of S in round k reduces the maximum depth of an arc end to at most $\lceil \ell/2 \rceil + t$. The alter in round k reduces this maximum depth by at least one, to at most $\lceil \ell/2 \rceil + t - 1$. Thus $\Phi_k(T) \leq \lceil \ell/2 \rceil + t - 1$. Since $\Phi_{k-1}(T) = \ell + 2t$ and $t \geq 1$, $\Phi_{k-1}(T) \geq \Phi_k(T)$, giving the first part of the lemma.

We prove the second part of the lemma by induction on $\Phi_{k-1}(T) = \ell + 2t$. If $\Phi_{k-1}(T) = 4$, then $t = 2$ and $\ell = 0$, or $t = 1$ and $\ell = 2$, so $\Phi_k(T) \leq \lceil \ell/2 \rceil + t + 1 = 3$. If $\Phi_{k-1}(T) = 5$, then $t = 2$ and $\ell = 1$, or $t = 1$ and $\ell = 3$, so $\Phi_k(T) \leq 4$. In both cases the second part of the lemma is true. Each increase of ℓ by two or t by one increases $\Phi_{k-1}(T)$ by two and increases the upper bound of $\lceil \ell/2 \rceil + t + 1$ on $\Phi_k(T)$ by one, which preserves the inequality $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$. □

Lemma 2.3.10 gives a potential drop for any active tree T such that $\Phi_{k-1}(T) \geq 4$. To obtain a potential drop if $\Phi_{k-1}(T) < 4$, we need to consider two rounds if $\Phi_{k-1}(T) = 3$ and three rounds if $\Phi_{k-1}(T) = 2$.

Lemma 2.3.11. *Let T be an active tree in round $k > 2$ of RA such that $\Phi_{k-1}(T) = 3$. Then $\Phi_{k-2}(T) \geq (4/3)\Phi_k(T)$.*

Proof. The lemma holds if T has at least two active constituent trees in round $k-2$ or one with an edge end of depth two or more, since then $\Phi_{k-2}(T) \geq 4$. Suppose neither of these cases occurs. Then T has one active constituent tree, say T_2, in round $k-2$. By Lemma 2.3.8, no pair of passive constituent trees of T in round $k-2$ is connected by an edge. Thus they are all connected by an edge with the root or a child of the root of T_2. Let T_1 be the active constituent tree of T in round $k-1$. The root of T_2 is the root or a child of the root of T_1. It follows that each passive constituent tree of T in round $k-1$ has an edge connecting its root with that of T_1. Hence the tree containing the vertices of T_1 formed by the connect step in round k has no edge ends of depth greater than two, which implies that T has no edge ends of positive depth, making its potential two. □

Lemma 2.3.12. *Let T be an active tree in round $k > 3$ of RA such that $\Phi_{k-1}(T) = 2$. Then $\Phi_{k-3}(T) \geq (3/2)\Phi_k(T)$.*

Proof. The lemma holds if there are at least two active constituent trees of T in round $k-2$ or in round $k-3$, or there is a constituent tree in one of these rounds with an edge end of positive depth: two active trees have total potential at least four, and an active tree with an edge end of depth at least one has a potential of at least three.

Suppose not. Let T_3, T_2, and T_1 be the active constituent trees of T in rounds $k-3$, $k-2$, and $k-1$, respectively. Since the constituent trees of T in round $k-3$ other than T_3 are passive, no edge connects any pair of them by Lemma 2.3.8. Since all their vertices are in T, each such tree must have an edge connecting its root with that of T_3. The connect step in round $k-2$ makes the minimum vertex in T the root of T_2. Tree T_2 has depth at most two by Lemma 2.2.2, since only its root can be an edge end. The connect step in round $k-1$ links each passive constituent tree of T in round $k-2$ to T_2, forming a tree S_1 containing all the vertices of T and of depth at most two. The shortcut in round $k-1$ transforms S_1 to T_1, which is flat since S_1 has depth at most two. But since T_1 is flat and contains all the vertices in T, it must be passive in round k, a contradiction. □

Having covered all the cases, we are ready to put them together. Let $a = (3/2)^{1/3} = 1.1447+$, and let $|T|$ be the number of vertices in tree T.

Lemma 2.3.13. *Let T be an active tree in round k of RA. Then $\Phi_k(T) \leq (3/2)|T|/a^{k-3}$.*

Proof. The proof is by induction on k. Since T contains at least two vertices and has potential at most $|T|+1$, it has potential at most $(3/2)|T|$. This gives the lemma for $k \leq 3$. Let $k > 3$ and suppose the lemma holds for smaller values. We consider three cases. If T contains an edge end of depth at least two, then $\Phi_k(T) \leq (4/5)\Phi_{k-1}(T) \leq (4/5)(3/2)|T|/a^{k-4} \leq (3/2)|T|/a^{k-3}$ by Lemma 2.3.10, the induction hypothesis, the linearity of the total potential, and the inequality $a \leq 5/4$. If the maximum depth of an edge end in T is one, then $\Phi_k(T) \leq (3/4)\Phi_{k-2}(T) \leq (3/4)(3/2)|T|/a^{k-5} \leq (3/2)|T|/a^{k-3}$ by Lemma 2.3.11, the induction hypothesis, the linearity of the total potential, and the inequality $a \leq (4/3)^{1/2}$. If no vertex in T other than the root is an edge end, then $\Phi_k(T) \leq (2/3)\Phi_{k-3}(T) \leq (2/3)(3/2)|T|/a^{k-6} \leq (3/2)|T|/a^{k-3}$ by Lemma 2.3.12, the induction hypothesis, the linearity of the total potential, and $a = (3/2)^{1/3}$. □

Theorem 2.3.14. *Algorithm RA takes $O(\log n)$ steps.*

Proof. By Lemma 2.3.13, if k is such that $a^{k-3} > (3/2)n$, then no tree can be active in round k. Only the last round has no active trees. □

We can use the same approach to prove an $O(\log n)$ step bound for algorithm R, but the details are more complicated. Algorithm RA has the advantage over R that the alter step in effect does extra flattening. If T is active in round k and T_1 is the only active constituent tree of T in round $k-1$, it is possible for the depth of T_1 to be one and that of T to be two. This is not a problem in RA, because the alter decreases the depth of each edge end by one. But in R we need to give extra potential to trees of depth one to make the potential function non-increasing.

In R we define the individual potential of a tree T at the end of round k to be zero if T is passive in round k, or the depth of T if T is active in round k and not flat, or two if T is active in round k and flat. As in RA, if T exists at the end of round k and $j \leq k$, we define the potential $\Phi_j(T)$ of T at the end of round j to be the sum of the potentials of its constituent trees at the end of round j, and we define the total potential at the end of round k to be the sum of the potentials of the trees existing at the end of round k. We prove analogues of Lemmas 2.3.10-2.3.13 and Theorem 2.3.14 for R.

Lemma 2.3.15. *Let T be active in round $k > 1$ of R. Then $\Phi_{k-1}(T) \geq \Phi_k(T)$, and if $\Phi_{k-1}(T) \geq 4$, then $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$.*

Proof. Let ℓ be the sum of the depths of the active constituent trees of T in round $k-1$, let t be the number of these trees, and let f be the number of these trees that are flat. Then $\ell \geq t$ and $\Phi_{k-1}(T) = \ell + f \geq \ell$. Let S be the tree formed from the constituent trees of T by the connect step in round k. This step in round k does not make a leaf into a non-leaf, nor does it make a root of a passive tree the parent of another such root by Lemma 2.3.8. It follows that any path in S contains at most $\ell + 2$ vertices: at most $\ell - t$ non-leaf vertices of active constituent trees, at most $t + 1$ roots of passive constituent trees, and at most one leaf of some constituent tree. The shortcut in round k transforms S into T, so the depth of T is at most $\lceil \ell/2 \rceil + 1$, as is its potential: if $\ell = 1$, $\lceil \ell/2 \rceil + 1 = 2$, which is the potential of a flat active tree.

If $\ell = 1$, there is one active constituent tree, and it is flat, so $\Phi_{k-1}(T) = 2 = \Phi_k(T)$, making the lemma true. If $\ell \geq 2$, $\Phi_{k-1}(T) \geq \ell \geq \lceil \ell/2 \rceil + 1 \geq \Phi_k(T)$. If $\ell = 4$, $\Phi_{k-1}(T) \geq 4$ and $\Phi_k(T) \leq 3$. If $\ell = 5$, $\Phi_{k-1}(T) \geq 5$ and $\Phi_k(T) \leq 4$. Thus the lemma is true if $\ell \leq 5$. Each increase of ℓ by two increases the lower bound of ℓ on $\Phi_{k-1}(T)$ by two and increases the upper bound of $\lceil \ell/2 \rceil + 1$ on $\Phi_k(T)$ by one, which preserves the inequality $\Phi_{k-1}(T) \geq (5/4)\Phi_k(T)$, so the lemma holds for all ℓ by induction. □

Lemma 2.3.15 gives a potential drop for any active tree T such that $\Phi_{k-1}(T) \geq 4$. To obtain a potential drop if $\Phi_{k-1}(T) < 4$, we need to consider two rounds if $\Phi_{k-1}(T) = 3$ and five rounds if $\Phi_{k-1}(T) = 2$.

Lemma 2.3.16. *Let T be an active tree in round $k > 2$ of R such that $\Phi_{k-1}(T) = 3$. Then $\Phi_{k-2}(T) \geq (4/3)\Phi_k(T)$.*

Proof. If the constituent trees of T in round $k-1$ or in round $k-2$ include at least two active trees, or one active tree of depth at least four, or T has depth at most two, the lemma holds.

Suppose not. Let T_2 and T_1 be the unique active constituent trees of T in rounds $k-2$ and $k-1$, respectively. Let S_1 and S be the trees containing the vertices of T_2 and T_1 formed by the connect steps in rounds $k-2$ and $k-1$, respectively. Trees T_2, T_1, and T all have depth three, and S_1 and S have depth five. No edge connects two passive constituent trees of T in round $k-2$, so each such tree is connected to T_2 by an edge. Call such a tree *primary* if it has an edge connecting it with the root or a child of the root of T_2, and *secondary* otherwise.

Since S_1 has depth five and T_2 has depth three, their roots must be different, so the root of S_1 is the minimum of the roots of the primary trees. Each vertex in T_2 that is the end of an edge whose other end is in a secondary tree has depth at least two in T_2, depth at least three in S_1, and depth at least two in T_1, which is formed from S_1 by the shortcut in round $k-1$. Since T_1 has depth three and S has depth five, their roots must be different. But then the root of S must be the root of one of the secondary trees, which is impossible since the root of T_1 is not the parent of any of the edge ends connecting T_1 with the secondary trees. □

Lemma 2.3.17. *Let T be an active tree in round $k > 5$ of R such that $\Phi_{k-1}(T) = 2$. Then $\Phi_{k-5}(T) \geq (3/2)\Phi_k(T)$.*

Proof. If for some j between $k-5$ and $k-1$ inclusive the constituent trees of T in round j include at least two active trees, or one active tree of depth at least three, then $\Phi_j(T) \geq 3$, so the lemma holds.

Suppose not. Then the constituent trees of T in each round from $k-5$ to k include exactly one active tree, of depth one or two. For j between 1 and 5 inclusive let T_j be the active constituent tree of T in round $k-j$, for j between 1 and 4 inclusive let S_j be the tree containing the vertices of T_{j+1} formed by the connect step in round $k-j$, and let S be the tree containing the vertices of T_1 formed by the connect in round k. For j from 1 to 4 inclusive, the shortcut in round $k-j$ transforms S_j into T_j, and the shortcut in round k transforms S into T.

No edge connects two passive constituent tree of T in round $k-5$, so each such tree has an edge connecting it with T_5. Call such a tree T *primary* if it has an edge connecting it with the root of T_t or to a child of the root of T_5, *secondary* otherwise. Since T_5 has depth at most two, each secondary tree has an edge connecting it with a grandchild of the root of T_5.

We consider two cases: the roots of T_5 and T_4 are the same, or they are different. In the former case, the roots of all primary trees are greater than the root of T_5, and the connect in round $k-4$ makes all of them children of the root of T_5. In the latter case, the root of T_4 is the minimum of the roots of the primary trees, and each such tree other than the one of minimum root is linked to

T_5 in round $k-4$ or to T_4 in round $k-3$.

Now consider the secondary trees. If the roots of T_5 and T_4 are the same, then after the shortcut in round $k-4$ each secondary tree has an edge connecting it with the root or a child of the root of T_4. By the argument in the preceding paragraph, each such tree will be linked with T_4 in round $k-3$ or with T_3 in round $k-2$. If the roots of T_5 and T_4 are different, none of the secondary trees has an edge connecting it with the root or a child of the root of T_4 at the end of round $k-4$. In this case the roots of T_4 and T_3 must be the same, so after the shortcut in round $k-3$ each secondary tree has an edge connecting it with the root or a child of the root of T_3. Each such tree will be linked with T_3 in round $k-2$ or with T_2 in round $k-1$. Furthermore the roots of T_2 and T_1 must be the same.

It follows that there is only one constituent tree of T in round $k-1$, and this tree is flat. But this tree must be T, making T passive in round k, a contradiction. □

Let $b = (3/2)^{1/5} = 1.0844+$.

Lemma 2.3.18. *Let T be an active tree in round k. Then $\Phi_k(T) \leq (3/2)|T|/b^{k-5}$.*

Proof. The proof is by induction on k. Since T contains at least two vertices and has potential at most $|T|+1$, it has potential at most $(3/2)|T|$. This gives the lemma for $k \leq 5$.

Suppose $k > 5$ and suppose the lemma holds for smaller values. We consider three cases. If the depth of T exceeds three, then $\Phi_k(T) \leq (4/5)\Phi_{k-1}(T) \leq (4/5)(3/2)|T|/b^{k-6} \leq (3/2)|T|/b^{k-5}$ by Lemma 2.3.15, the induction hypothesis, the linearity of the total potential, and the inequality $b \leq 5/4$.

If the depth of T is three, then $\Phi_k(T) \leq (3/4)\Phi_{k-2}(T) \leq (3/4)(3/2)|T|/b^{k-7} \leq (3/2)|T|/b^{k-5}$ by Lemma 2.3.16, the induction hypothesis, the linearity of the total potential, and the inequality $b \leq (4/3)^2$.

If the depth of T is at most two, then $\Phi_k(T) \leq (2/3)\Phi_{k-5}(T) \leq (2/3)(3/2)|T|/b^{k-10} \leq (3/2)|T|/b^{k-5}$ by Lemma 2.3.17, the induction hypothesis, the linearity of the total potential, and $b \leq (3/2)^{1/5}$. □

Theorem 2.3.19. *Algorithm R takes $O(\log n)$ steps.*

Proof. By Lemma 2.3.18, if k is such that $b^{k-5} > (3/2)n$, then no tree can be active in round k. Only the last round has no active trees. □

For the variants of R and RA that do two shortcuts in each round instead of just one, we can simplify the analysis and improve the constants. For RA, we let the potential of an active tree be

the maximum depth of an edge end (or zero if there are no edge ends) plus one. The potential of an active tree drops from the previous round by at least a factor of two unless it has only one active constituent tree in the previous round and that tree has potential one. The proof of Lemma 2.3.12 gives a potential reduction of at least a factor of two in at most three rounds in this case. Lemma 2.3.13 holds with a replaced by $2^{1/3} = 1.2599+$. For R, we let the potential of an active tree be its depth. The potential of an active tree drops from the previous round by at least a factor of at least two unless it has only one active constituent tree in the previous round and that tree is flat. The proof of Lemma 2.3.17 gives a potential reduction of at least a factor of two in at most three rounds, and Lemma 2.3.18 holds with b replaced by $2^{1/3}$, the same constant as for the two-shortcut variant of RA.

This analysis suggests that doing two shortcuts per round rather than one might improve the practical performance of R and RA, especially since a shortcut needs only n processors, but a connect step needs m. Exactly how many shortcuts to do per round is a question for experiments to resolve. Our analysis of algorithm S suggests that doing a non-constant number of shortcuts per round is likely to degrade performance. We have analyzed the one-shortcut-per-round algorithms R and RA, even though their analysis is more complicated than the corresponding two-shortcut-per-round algorithms, because this analysis applies to the corresponding algorithms with any constant number of shortcuts per round, and our goal is to determine the *simplest* algorithms with an $O(\log n)$ step bound.

2.4 Related Work

In this section we review previous work related to ours. We have presented our results first, since they provide insights into the related work. As far as we can tell, all our algorithms are novel and simpler than previous algorithms, although they are based on some of the previous algorithms.

Two different communities have worked on concurrent connected components algorithms, in two overlapping eras. First, theoretical computer scientists developed provably efficient algorithms for various versions of the PRAM model. This work began in the late 1970's and reached a natural conclusion in the work of Halperin and Zwick [79, 80], who gave $O(\log n)$-step, $O(m)$-work randomized algorithms for the EREW (exclusive read, exclusive write) PRAM. Their second algorithm finds spanning trees of the components. The EREW PRAM is the weakest variant of the PRAM model, and finding connected components in this model requires $\Omega(\log n)$ steps [47]. To solve the problem sequentially takes $O(m)$ time, so the Halperin-Zwick algorithms minimize both the number

of steps and the total work (number of steps times the number of processors). Whether there is a deterministic EREW PRAM algorithm with the same efficiency remains an open problem.

Halperin and Zwick's paper [80] contains a table listing results preceding theirs, and we refer the reader to their paper for these results. Our interest is in simple algorithms for a more powerful computational model, so we content ourselves here with discussing simple labeling algorithms related to ours. (The Halperin-Zwick algorithms and many of the preceding ones are *not* simple.) First we review variants of the PRAM model and how they relate to our algorithmic framework.

The three main variants of the PRAM model, in increasing order of strength, are EREW, CREW (concurrent read, exclusive write), and CRCW (concurrent read, concurrent write). The CRCW PRAM has four standard versions that differ in how they handle write conflicts: (i) COMMON: all writes to the same location at the same time must be of the same value; (ii) ARBITRARY: among concurrent writes to the same location, an arbitrary one succeeds; (iii) PRIORITY: among concurrent writes to the same location, the one done by the highest-priority processor succeeds; (iv) COMBINING: values written concurrently to a given location are combined using some symmetric function. As discussed in §2.1, our algorithms can be implemented on a COMBINING CRCW PRAM, with minimization as the combining function.

An early and important theoretical result is the $O(\log^2 n)$-step CREW PRAM algorithm of Hirschberg, Chandra, and Sarwate [82]. Algorithm S is a simplification of their algorithm. They represent the graph by an adjacency matrix, but it is easy to translate their basic algorithm into our framework. Their algorithm alternates connect steps with repeated shortcuts. To do connection they use a variant of *parent-connect* that we call *strong-parent-connect*. It concurrently sets $x.p$ for each vertex x equal to the minimum $w.p \neq x$ such that there is an edge $\{v, w\}$ with $v.p = x$; if there is no such edge, $x.p$ does not change.

The following pseudocode implements this method in our framework:

```
strong-parent-connect:
    for each vertex v do
        v.n = ∞
    for each edge {v, w} do
        if v.p ≠ w.p then
            v.p.n = min{v.p.n, w.p}
            w.p.n = min{w.p.n, v.p}
        for each vertex v do
            if v.n ≠ ∞ then
                v.p = v.n
```

This version of connection can make a larger vertex the parent of a smaller one. Thus their algorithm does not do minimum labeling. Furthermore it can create parent cycles of length two, which Hirschberg et al. eliminate in a cleanup step at the end of each round. To do the cleanup it suffices to concurrently set $v.p = v$ for each vertex such that $v.p > v$ and $v.p.p = v$. Their algorithm is one of two we have found in the literature that can create parent cycles.

Although they do not say this, we think the reason Hirschberg et al. used *strong-parent-connect* was to guarantee that each tree links with another tree in each round. This gives them an $O(\log n)$ bound on the number of rounds and an $O(\log^2 n)$ bound on the number of steps, since there are $O(\log n)$ shortcuts per round. Our simpler algorithm S uses *parent-connect* in place of *strong-parent-connect*, making it a minimum labeling algorithm and eliminating the cleanup step. Although *parent-connect* does not guarantee that each tree links with another tree every round, it does guarantee such linking every two rounds, giving us the same $O(\log^2 n)$ step bound as Hirschberg et al. See the proof of Theorem 2.3.1.

The first $O(\log n)$-step PRAM algorithm was that of Shiloach and Vishkin [143]. It runs on an ARBITRARY CRCW PRAM, as do the other algorithms we discuss, except as noted.

The following is a version of their algorithm SV in our framework:

```
Algorithm SV:
    repeat
        {shortcut; arb-parent-root-connect; stagnant-parent-root-connect; shortcut}
    until no v.p changes

    arb-parent-root-connect:
        for each vertex v do
            v.o = v.p
        for each edge {v, w} do
            if v.o > w.o and v.o = v.o.o then
                v.o.p = w.o
            else if w.o = w.o.o then
                w.o.p = v.o

    stagnant-parent-root-connect:
        for each vertex v do
            v.o = v.p
        for each edge {v, w} do
            if v.o ≠ w.o then
                if v.o is a stagnant root then
                    v.o.p = w.o
                else if w.o is a stagnant root then
                    w.o.p = v.o
```

Whereas *parent-root-connect* updates the parent of each root x to be the *minimum* y such that there is an edge $\{v, w\}$ with $x = v.o$ and $y = w.o < v.o$ if there is such an edge, *arb-parent-root-connect* replaces the parent of each such root by an *arbitrary* such y. Arbitrary resolution of write conflicts suffices to implement the latter method, but not the former.

Shiloach and Vishkin define a root to be *stagnant* if its tree is not changed by the first two steps of the main loop (the first shortcut and the *arb-parent-root-connect*). Their algorithm has additional steps to keep track of stagnant roots. Method *stagnant-parent-root-connect* updates the parent of each stagnant root x to be an arbitrary y such that there is an edge $\{v, w\}$ with $x = v.o$

and $y = w.o \neq v.o$ if there is such an edge. The definition of "stagnant" implies that no two stagnant trees are connected by an edge. Algorithm SV does not do minimum labeling, since *stagnant-parent-root-connect* can make a larger vertex the parent of a smaller one. Nevertheless, the algorithm creates no cycles, although the proof of this is not straightforward, nor is the efficiency analysis. Algorithm R is algorithm SV with the third and fourth steps of the main loop deleted and the second step modified to resolve concurrent writes by minimum value instead of arbitrarily. Shiloach and Vishkin state that one shortcut can be deleted from their algorithm without affecting its asymptotic efficiency. They included the third step for two reasons: (i) their analysis examines one round at a time, requiring that every tree change in every round, and (ii) if the third step is deleted, the algorithm can take $\Omega(n)$ steps on a graph that is a tree with edges $\{i, n\}$ for $i \in [n-1]$. This example strongly suggests that to obtain a simpler algorithm one needs to use a more powerful model of computation, as we have done by using minimization to resolve write conflicts. Awerbuch and Shiloach presented a slightly simpler $O(\log n)$-step algorithm and gave a simpler efficiency analysis [24]. Our analysis of algorithms R and RA in §2.3.3 uses a variant of their potential function. Their algorithm is algorithm SV with the first shortcut deleted and the two connect steps modified to update only parents of roots of flat trees. The computation needed to keep track of flat tree roots is simpler than that needed in algorithm SV to keep track of stagnant roots. An even simpler but randomized $O(\log n)$-step algorithm was proposed by Reif [138]:

```
Algorithm Reif:
    repeat
        {for each vertex flip a coin; random-parent-connect; shortcut}
    until no v.p changes

    random-parent-connect:
        for each vertex v do
            v.o = v.p
        for each edge {v, w} do
            if v.o flipped heads and w.o flipped tails then
                v.o.p = w.o
            else if w.o flipped heads and v.o flipped tails then
                w.o.p = v.o
```

Reif's algorithm keeps the trees flat, making the algorithm monotone, although it does not do minimum labeling. Although it is randomized, Reif's algorithm is simpler than those of Shiloach and Vishkin, but R and RA are even simpler and are deterministic.

We know of one algorithm other than that of Hirschberg et al. [82] that does not maintain acyclicity. This is the algorithm of Johnson and Metaxis [89]. Their algorithm runs in $O((\log n)^{3/2})$ steps on an EREW PRAM. It uses a form of shortcutting to eliminate any cycles created by connection steps.

Algorithms that run on a more restricted form of PRAM, or use fewer processors (and thereby do less work) use various kinds of edge alteration, edge addition, and edge deletion, along with techniques to resolve read and write conflicts. Such algorithms are much more complicated than those we have considered. Again we refer the reader to [79, 80] for results and references.

The second era of concurrent connected components algorithms was that of the experimentalists. It began in the 1990's and continues to the present. Experimentation has expanded greatly with the growing importance of huge graphs representing the internet, the world-wide web, friendship connections, and other symmetric relations, as well as the development of cloud computing frameworks. These trends make concurrent algorithms for connected components both practical and useful. The general approach of the experimentalists has been to take one or more existing algorithms, possibly simplify or modify them, implement the resulting suite of algorithms on one or more computing platforms, and report the results of experiments done on some collection of graphs. Examples of such studies include [71, 77, 85, 161, 126, 151, 83, 162].

Some of these papers make claims about the theoretical efficiency of algorithms they propose, but several of these claims are incorrect or unjustified. We give some examples. The first is a paper by Greiner [77] in which he claims an $O(\log^2 n)$ step bound for his "hybrid" algorithm.

Greiner's description of this algorithm is incomplete. The algorithm is a modification of the algorithm of Hirschberg et al. [82]. Each round does a form of direct connect followed by repeated shortcuts followed by an alteration. Since repeated shortcuts guarantee that all trees are flat at the beginning of each round, this is equivalent to using a version of *parent-connect* and not doing alteration. The main novelty in his algorithm is that alternate rounds use maximization instead of minimization in the connect step. He does not specify exactly how the connect step works. There are at least two possibilities. One is to use *direct-connect*, but in alternate rounds replace min by max. The resulting algorithm is a min-max version of algorithm S. The second is to use the following strong version of *direct-connect*, but in alternate rounds replace min by max and ∞ by $-\infty$:

```
strong-direct-connect:
    for each vertex v do
        v.n = ∞
    for each edge {v, w} do
        v.n = min{v.n, w}
        w.n = min{w.n, v}
    for each vertex v do
        if v.n ≠ ∞ then
            v.p = v
```

The resulting algorithm is a min-max version of the Hirschberg et al. algorithm. Greiner claims an $O(\log n)$ bound on the number of rounds and an $O(\log^2 n)$ bound on the number of steps. But these bounds do not hold for the algorithm that uses the min-max version of *direct-connect*: on the bad example of Shiloach and Vishkin consisting of an unrooted tree with vertex n adjacent to vertices 1 through $n-1$, the algorithm takes $\Omega(n)$ steps. This example has a high-degree vertex, but there is a simple example whose vertices are of degree at most three, consisting of a path of odd vertices $1, 3, 5, \ldots, n$ with each even vertex i adjacent to $i+1$.

On the other hand, the algorithm that uses the min-max version of *strong-direct-connect* can create parent cycles of length two, which must be eliminated by a cleanup as in the Hirschberg et al. algorithm. Greiner says nothing about eliminating cycles. We conclude that either his step bound is incorrect or his algorithm is incorrect.

At least one other work reproduces Greiner's error: Soman et al. [145, 146] propose a modification of Greiner's algorithm intended for implementation on a GPU model. Their algorithm is the inefficient version of Greiner's algorithm, modified to use *parent-connect* instead of *direct-connect* and without alteration.

Their specific implementation of the connect step is as follows:

```
alternate-connect:
    for each edge {v, w} do
        if v.p ≠ w.p then
            x = min{v.p, w.p}
            y = max{v.p, w.p}
            if round is even then
                y.p = x
            else x.p = y
```

Soman et al. say nothing about how to resolve concurrent writes. If this resolution is arbitrary, or by minimum in the even rounds and by maximum in the odd rounds, then the algorithm takes $\Omega(n)$ steps on the examples mentioned above.

Algorithm S, and the equivalent algorithm that uses *direct-connect* and alteration, are simpler than the algorithms of Greiner and Soman et al. and have guaranteed $O(\log^2 n)$ step bounds. We conclude that alternating minimization and maximization adds complication without improving efficiency, at least in theory.

Another paper that has an invalid efficiency bound as a result of not handling concurrent writes carefully is that of Yan et al. [161]. They consider algorithms in the PREGEL framework [123], which is a graph-processing platform designed on top of the MPC model. All the algorithms they consider can be expressed in our framework. They give an algorithm obtained from algorithm SV by deleting the first shortcut and replacing the second connect step by the first connect step of Awerbuch and Shiloach's algorithm. In fact, the second connect step does nothing, since any parent update it would do has already been done by the first connect step. That is, this algorithm is equivalent to algorithm SV with the first shortcut and the second connect step deleted. Their termination condition, that all trees are flat, is incorrect, since there could be two or more flat trees in the same component. They claim an $O(\log n)$ bound on steps, but since they assume arbitrary resolution of write conflicts, the actual step bound is $\Theta(n)$ by the example of Shiloach and Vishkin.

A third paper with an analysis gap is that of Stergio, Rughwani, and Tsioutsiouliklis [151]. They present an algorithm that we call SRT, whose main loop expressed in our framework is the following:

```
Algorithm SRT:
    repeat
        for each vertex v do
            v.o = v.p
            v.n = v.p
        for each edge {v, w} do
            if v.o > w.o then
                v.n = min{v.n, w.o}
            else w.n = min{w.n, v.o}
        for each vertex v do
            v.o.p = min{v.o.p, v.n}
        for each vertex v do
            v.p = min{v.p, v.n.o}
    until no v.p changes
```

This algorithm does an extended form of connection combined with a variant of shortcutting that combines old and new parents. It is not monotone. Stergio et al. implemented this algorithm on the Hronos computing platform and successfully solved problems with trillions of edges. They claimed an $O(\log n)$ step bound for the algorithm, but we are unable to make sense of their analysis. Their paper motivated our work.

A recently proposed algorithm similar to SRT is FastSV, proposed by Zhang, Azad, and Hu [162]. This algorithm is not monotone. It combines connecting and shortcutting. In each round it computes the grandparent $v.g = v.p.p$ of each vertex v and uses $v.g$ as a candidate for $v.p$, $w.p.p$ and $w.p$, for each edge $\{v, w\}$. They did not analyze FastSV but instead compared it experimentally to a simplified version of Awerbuch and Shiloach's algorithm called LACC and to other variants of SV. In their experiments SV was fastest.

We have been unable to prove a worst-case polylogarithmic step bound for SRT, nor for FastSV, nor indeed for any non-monotone algorithm that does not use edge alteration. But we do not have bad examples for these algorithms either.

A final paper with an interesting algorithm but incorrect analysis is that of Burkhardt [38]. The main novelty in Burkhardt's algorithm is to replace each edge $\{v, w\}$ by a pair of oppositely directed arcs (v, w) and (w, v) and to use *asymmetric* alteration: he replaces (v, w) by $(w, v.p)$

instead of $(v.p, w.p)$ (unless $w = v.p$). This idea allows him to combine connecting and shortcutting in a natural way. (Burkhardt claims that his algorithm does not do shortcutting, but it does, implicitly.) Burkhardt does not give an explicit stopping rule, saying only, "This is repeated until all labels converge." An iteration can alter arcs without changing any parents, so one must specify the stopping rule carefully. The following is a version of the main loop of Burkhardt's algorithm with the parent updates and the arc alterations disentangled, and which stops when there is one root and all other vertices are leaves:

```
Algorithm B:
    repeat
        for each arc (v, w) do
            if v > w then
                v.p = min{v.p, w}
        for each arc (v, w) do
            if v.p ≠ w then
                replace (v, w) by (w, v.p)
            else delete (v, w)
        for each vertex v do
            if v.p ≠ v then
                add arc (v.p, v)
    until every arc (v, w) has v.p = w.p and v.p ∈ {v, w}
```

Burkhardt claims that his algorithm takes $O(\log d)$ steps, which would be remarkable if true. Unfortunately, a long skinny grid is a counterexample, as shown in [10]. Burkhardt also claimed that the number of arcs existing at any given time is at most $2m + n$. The version above has a $2m$ upper bound on the number of arcs. Two small changes in the algorithm reduce the upper bound on the number of arcs to m and make the shortcutting more efficient: replace each original edge $\{v, w\}$ by *one* arc $(\max\{v, w\}, \min\{v, w\})$, and in the loop over the vertices replace "add arc $(v.p, v)$" by "add arc $(v, v.p.p)$." We call the resulting algorithm AA, for *asymmetric alteration*.

The following pseudocode implements this algorithm:

```
Algorithm AA:
    for each vertex v do v.p = v
    for each edge {v, w} do
        replace {v, w} by arc (max{v, w}, min{v, w})
    repeat
        for each arc (v, w) do
            v.p = min{v.p, w}
        for each arc (v, w) do
            delete (v, w)
            if w ≠ v.p then
                add arc (w, v.p)
        for each vertex v do
            if v ≠ v.p then
                add arc (v, w.p)
    until no arc (v, w) has w ≠ v.p
```

A version of Algorithm AA was proposed to us by Yu-Pei Duo [private communication, 2018]. The techniques of §2.3.2 extend to give an $O(\log^2 n)$ step bound for AA, B, and Burkhardt's original algorithm. We omit the details.

Very recently, theoreticians have become interested in concurrent algorithms for connected components again, with the aim of obtaining a step bound logarithmic in d rather than n, for a suitably powerful model of computation. The first, breakthrough result in this direction was that of Andoni et al. [10]. They gave a randomized algorithm that takes $O(\log d \log\log_{m/n} n)$ steps in the MPC model. Their algorithm uses graph densification based on the distance-doubling technique of [137], controlled to keep the number of edges linearly bounded. Behnezhad et al. [28] improved the result of Andoni et al. by reducing the number of steps to $O(\log d + \log\log_{m/n} n)$. Their algorithm can be implemented in the MPC model or on a very powerful version of the CRCW PRAM that supports a "multiprefix" operation. In next chapter we show that this algorithm and that of Andoni et al. can be simplified and implemented on an ARBITRARY CRCW PRAM.

Chapter 3

Connected Components in Log Diameter Time

In the last chapter, we have introduced many PRAM and MPC algorithms for connected components with a running time of $O(\log n)$, as well as a conditional lower bound of $\Omega(\log d)$ in Theorem 2.1.1. In practice, many graphs in applications have components of small diameter, perhaps poly-logarithmic in n. This observations lead to the question of whether one can find connected components faster on graphs of small diameter, perhaps by exploiting the power of the MPC model. Andoni et al. [10] answered this question "yes" by giving an MPC algorithm that finds connected components in $O(\log d \log\log_{m/n} n)$ time, where d is the largest diameter of a component. Very recently, this time bound was improved to $O(\log d + \log\log_{m/n} n)$ by Behnezhad et al. [28]. Both of these algorithms are complicated and use the extra power of the MPC model, in particular, the ability to sort and compute prefix sums in $O(1)$ communication rounds. These operations require $\Omega(\log n/\log\log n)$ time on a CRCW PRAM with poly(n) processors [25].[1] These results left open the following fundamental problem in theory:[2]

> *Is it possible to break the* $\log n$ *time barrier for connected components of small-diameter graphs on a PRAM (without the additional power of an MPC)?*

In this chapter we give a positive answer by presenting an ARBITRARY CRCW PRAM algorithm that runs in $O(\log d + \log\log_{m/n} n)$ time, matching the round complexity of the MPC algorithm by

[1] Behnezhad et al. also consider the *multiprefix* CRCW PRAM, in which prefix sum (and other primitives) can be computed in $O(1)$ time and $O(m)$ work. A direct simulation of this model on a PRIORITY CRCW PRAM or weaker model would suffer an $\Omega(\log n/\log\log n)$ factor in both time and work, compared to our result.

[2] A repeated matrix squaring of the adjacency matrix computes the connected components in $O(\log d)$ time on a CRCW PRAM, but this is far from work-efficient – the currently best work is $O(n^{2.373})$ [65].

Behnezhad et al. [28]. Our algorithm uses $O(m)$ processors and space, and thus is space-optimal and nearly work-efficient. In contrast to the MPC algorithm, which uses several powerful primitives, we use only hashing and other simple data structures. Our hashing-based approach also applies to the work of Andoni et al. [10], giving much simpler algorithms for connected components and spanning forest, which should be preferable in practice. While the MPC model ignores the total work, our result on the more fine-grained PRAM model captures the inherent complexities of the problems.

In the rest of this chapter, to state bounds simpler, we assume the number of edges m is at least two times n, the number of vertices in the graph. This is because one can add two self-loops on each vertex in the input graph. After this, one can still assume that each vertex or edge has a corresponding processor because one processor can simulate two processors in $O(1)$ time.

3.1 Preliminaries

3.1.1 Computation Models and Main Results

Our main model of computation is the ARBITRARY CRCW PRAM [160]. It consists of a set of processors , each of which has a constant number of cells (words) as the private memory, and a large common memory. The processors run synchronously. In one step, a processor can read a cell in common memory, write to a cell in common memory, or do a constant amount of local computation. Any number of processors can read from or write to the same common memory cell concurrently. If more than one processor writes to the same memory cell at the same time, an arbitrary one succeeds. Our main results are stated below:

Theorem 3.1.1 (Connected Components)**.** *There is an ARBITRARY CRCW PRAM algorithm using $O(m)$ processors that computes the connected components of any given graph. With probability $1 - 1/poly((m \log n)/n)$, it runs in $O(\log d \log \log_{m/n} n)$ time.*

The algorithm of Theorem 3.1.1 can be extended to computing a spanning forest (a set of spanning trees of the components) with the same asymptotic efficiency:

Theorem 3.1.2 (Spanning Forest)**.** *There is an ARBITRARY CRCW PRAM algorithm using $O(m)$ processors that computes the spanning forest of any given graph. With probability $1-1/poly((m \log n)/n)$, it runs in $O(\log d \log \log_{m/n} n)$ time.*

Using the above algorithms as bases, we provide a faster connected components algorithm that is nearly optimal (up to an additive factor of at most $O(\log \log n)$) due to a conditional lower bound of $\Omega(\log d)$ [142, 28].

Theorem 3.1.3 (Faster Connected Components)**.** *There is an ARBITRARY CRCW PRAM algorithm using $O(m)$ processors that computes the connected components of any given graph. With probability $1 - 1/poly((m \log n)/n)$, it runs in $O(\log d + \log\log_{m/n} n)$ time.*

For a dense graph with $m = n^{1+\Omega(1)}$, the algorithms in all three theorems run in $O(\log d)$ time with probability $1 - 1/\text{poly}(n)$; if $d = \log^{\Omega(1)}_{m/n} n$, the algorithm in Theorem 3.1.3 runs in $O(\log d)$ time.[3]

Note that the time bound in Theorem 3.1.1 is dominated by the one in Theorem 3.1.3. We include Theorem 3.1.1 here in pursuit of simpler proofs of Theorem 3.1.2 and Theorem 3.1.3.

3.1.2 Related Work and Technical Overview

In this section, we give an overview of the techniques in our algorithms, highlighting the challenges in the PRAM model and the main novelty in our work compared to that of Andoni et al. [10] and Behnezhad et al. [28].

Andoni et al. [10] observed that if every vertex in the graph has a degree of at least $b = m/n$, then one can choose each vertex as a *leader* with probability $\Theta(\log(n)/b)$ to make sure that with high probability, every non-leader vertex has at least 1 leader neighbor. As a result, the number of vertices in the contracted graph is the number of leaders, which is $\widetilde{O}(n/b)$ in expectation, leading to double-exponential progress, since we have enough space to make each remaining vertex have degree $\widetilde{\Omega}(m/(n/b)) = \widetilde{\Omega}(b^2)$ in the next round and $\widetilde{\Omega}(b^{2^i})$ after i rounds.[4] The process of adding edges is called *expansion*, as it expands the neighbor sets. It can be implemented to run in $O(\log d)$ time. This gives an $O(\log d \log\log_{m/n} n)$ running time.

Behnezhad et al. improved the multiplicative $\log\log_{m/n} n$ factor to an additive factor by streamlining the expansion procedure and double-exponential progress when increasing the space per vertex [28]. Instead of increasing the degree of each vertex *uniformly* to at least b in each round, they allow vertices to have different space budgets, so that the degree lower bound varies on the vertices. They define the *level* $\ell(v)$ of a vertex v to control the *budget* of v (space owned by v): initially each vertex is at level 1 with budget m/n. Levels increase over rounds. Each v is assigned a budget of $b(v) = (m/n)^{c^{\ell(v)}}$ for some fixed constant $c > 1$. The maximal level $L \coloneqq \log_c \log_{m/n} n$ is such that a vertex at level L must have enough space to find all vertices in its component. Based on this idea, they design an MPC algorithm that maintains the following invariant:

[3] Without the assumption that $m \geq 2n$ for simplification, one can replace the m with $2(m+n)$ in all our statements by creating self-loops in the graph.

[4] We use $\widetilde{O}$ to hide polylog(n) factors.

Observation 3.1.4 ([28]). *With high probability, for any two vertices u and v at distance* 2*, after* 4 *rounds, if $\ell(u)$ does not increase then their distance decreases to* 1*; moreover, the* skipped *vertex w originally between u and v satisfies $\ell(w) \leq \ell(u)$.*[5]

Behnezhad et al. proved an $O(\log d + \log\log_{m/n} n)$ time bound by a potential-based argument on an arbitrary fixed shortest path P_1 in the original graph: in round 1 put 1 coin on each vertex of P_1 (thus at most $d+1$ coins in total); for the purpose of analysis only, when inductively constructing P_{i+1} in round i, every skipped vertex on P_i is removed and passes its coins evenly to its successor and predecessor (if exist) on P_i. They claimed that any vertex v still on P_i in round i has at least $1.1^{i-\ell(v)}$ coins based on the following:

Observation 3.1.5 (Claim 3.12 in [28]). *For any path P_i in round i corresponding to an original shortest path, its first and last vertices are on P_j in round j for any $j > i$; moreover, if a vertex on P_i does not increase level in* 4 *rounds, then either its predecessor or successor on P_i is skipped.*

The first statement is to maintain the connectivity for every pair of vertices in the original graph, and the second statement is to guarantee that each vertex obtains enough coins in the next round. (Observation 3.1.5 is seemingly obvious from Observation 3.1.4, however there is a subtle issue overlooked in [28] that invalidates the statement. Their bound is still valid without changing the algorithm by another potential-based argument, which shall be detailed later in this section.)

Since the maximal level is L, by the above claim, a vertex on such a path with length more than 1 in round $R \coloneqq 8\log d + L$ would have at least $1.1^{8\log d} > d+1$ coins, a contradiction, giving the desired time bound.

Let us first show a counter-example for Observation 3.1.5 (not for their time bound). Let the original path P_1 be $(v_1, v_2, \ldots, v_s)$. We want to show that the distance between v_1 and v_s is at most 1 after R rounds, so neither v_1 nor v_s can be skipped during the path constructions over rounds. Suppose v_1 does not increase level in 4 rounds, then for v_1 to obtain enough coins to satisfy the claim, v_2 must be skipped. We call an ordered pair (v_i, v_{i+1}) of consecutive vertices on the path *frozen* if v_i is kept and v_{i+1} is skipped in 4 rounds. So (v_1, v_2) is frozen. Let v_3 (any vertex after v_2 suffices) be the *only* vertex on P_1 directly connecting to v_1 after skipping v_2 in 4 rounds. (Observation 3.1.4 guarantees that a vertex directly connecting to v_1 must exist but cannot guarantee that there is more than 1 such vertex on P_1.) Note that v_3 *cannot* be skipped, otherwise v_1 is isolated from P_1 and thus cannot connect to v_s. Now consider two cases. If v_4 is skipped, then pair (v_3, v_4) is frozen. If v_4 is not skipped, then assume v_4 does not increase level in 4 rounds. For v_4 to obtain enough

[5] For simplicity, we ignore the issue of changing the graph and corresponding vertices for now.

coins, v_5 must be skipped because v_3 cannot be skipped to pass coins to v_4, so the pair (v_4, v_5) is frozen. Observe that from frozen pair (v_1, v_2), in either case we get another frozen pair, which propagates inductively to the end of P_1. If we happen to have a frozen pair (v_{s-1}, v_s), then v_s must be skipped and isolated from v_1, a contradiction.

Here is a fix to the above issue (formally stated in Lemma 3.4.15). Note that only the last vertex v on the path can violate the claim that there are at least $1.1^{i-\ell(v)}$ coins on v in round i. Assuming the claim holds for all vertices on the current path, one can always *drop* (to distinguish from skip) v_s and pass its coins to v_{s-1} (which must be kept) if they are a frozen pair. So v_{s-1}, the new last vertex after 4 rounds, obtains enough coins by the induction hypothesis and the second part of Observation 3.1.4. By the same argument, the resulting path after R rounds has length at most 1. Observe that we dropped $O(R)$ vertices consecutively located at the end of the path, so the concatenated path connecting v_1 and the original v_s has length $O(R)$. Now applying Observation 3.1.4 to v_1, we get that in 4 rounds, either its level increases by 1 or its successor is skipped. Therefore, in $O(R + L)$ rounds, there is no successor of v_1 to be skipped and the graph has diameter at most 1 by a union bound over all the (original) shortest paths.

Now we introduce the new algorithmic ideas in our PRAM algorithm with a matching time bound.

The first challenge comes from processor allocation. To allocate different-sized blocks of processors to vertices in each round, there is actually an existing tool called *approximate compaction*, which maps the k distinguished elements in a length-n array one-to-one to an array of length $O(k)$ with high probability [8]. (The vertices to be assigned blocks are *distinguished* and their names are indices in the old array; after indexing them in the new array, one can assign them predetermined blocks.) However, the current fastest (and work-optimal) approximate compaction algorithm takes $O(\log^* n)$ time, introducing a multiplicative factor [70]. To avoid this, our algorithm first reduces the number of vertices to $n/\text{polylog}(n)$ in $O(\log \log_{m/n} n)$ time, then uses approximate compaction to rename the remaining vertices by an integer in $[n/\text{polylog}(n)]$ in $O(\log^* n)$ time. After this, each cell of the array to be compacted owns $\text{polylog}(n)$ processors, and each subsequent compaction (thus processor allocation) can be done in $O(1)$ time [74].

The second challenge is much more serious: it is required by the union bound that any vertex u must connect to *all* vertices within distance 2 with high probability if u does not increase in level. Behnezhad et al. achieve this goal by an algorithm based on constant-time sorting and prefix sum, which require $\Omega(\log n / \log \log n)$ time on an ARBITRARY CRCW PRAM with $\text{poly}(n)$ processors [25]. Our solution is based on hashing: to expand a vertex u, hash all vertices within distance 2 from

u to a hash table owned by u; if there is a collision, increase the level of u.[6] As a result, we are able to prove the following result, which is stronger than Observation 3.1.4 as it holds deterministically and uses only 1 round:

Observation 3.1.6 (formally stated in Lemma 3.4.24)**.** *For any two vertices u and v at distance 2, if $\ell(u)$ does not increase then their distance decreases to 1 in the next round; moreover, any vertex originally between u and v has level at most $\ell(u)$.*

The idea of increasing the level immediately after seeing a collision gives a much cleaner proof of Observation 3.1.6, but might be problematic in bounding the total space/number of processors: a vertex with many vertices within distance 2 can incur a collision and level increase very often. We circumvent this issue by increasing the level of each budget-b vertex with probability $\widetilde{\Theta}(b^{-\delta})$ before hashing. Then a vertex v with at least b^{δ} vertices within distance 2 would see a level increase with high probability; if the level does not increase, there should be at most b^{δ} vertices within distance 2, thus there is a collision when expanding v with probability $1/\text{poly}(b)$. As a result, the probability of level increase is $\widetilde{\Theta}(b^{-\delta}) + 1/\text{poly}(b) \leq b^{-c}$ for some constant $c > 0$, and we can assign a budget of $b^{1+\Omega(c)}$ to a vertex with increased level, leading to double-exponential progress. As a result, the total space is $O(m)$ with good probability since the union bound is over all $\text{polylog}(n)$ different levels and rounds, instead of $O(n^2)$ shortest paths.

Suitable combination of the ideas above yields a PRAM algorithm that reduces the diameter of the graph to at most 1 in $O(R) = O(\log d + \log\log_{m/n} n)$ time, with one *flexibility*: the relationship between the level $\ell(v)$ and budget $b(v)$ of vertex v in our algorithm is *not* strictly $b(v) = (m/n)^{c^{\ell(v)}}$ for some fixed constant $c > 1$ as in Behnezhad et al. [28]; instead, we allow vertices with the same level to have *two* different budgets. We show that such flexibility still maintains the key invariant of our algorithm (without influencing the asymptotic space bound): if a vertex is not a root in a tree in the labeled digraph, then its level must be strictly lower than the level of its parent (formally stated in Lemma 3.4.4).[7] Using hashing and a proper parent-update method, our algorithm does not need to compute the number of neighbors with a certain level for *each* vertex, which is required in [10, 28] and solved by constant-time sorting and prefix sum on an MPC. If this were done by a direct application of (constant-time) *approximate counting* (cf. [7]) on each vertex, then each round would take $\Omega(k)$ time where k is the maximal degree of any vertex, so our new ideas are essential to obtain the desired time bound.

[6] Hashing also naturally removes the duplicate neighbors to get the desired space bound – a goal achieved by sorting in [10, 28].

[7] Our algorithm adopts the framework of *labeled digraph* (or *parent graph*) for computing and representing components, which is standard in PRAM literatures, see §2.1.

Finally, we note that while it is straightforward to halt when the graph has diameter at most 1 in the MPC algorithm, it is not correct to halt (nor easy to determine) in this case due to the different nature of our PRAM algorithm. After the diameter reaches $O(1)$, to correctly compute components and halt the algorithm, we borrow an idea from [117] to flatten all trees in the labeled digraph in $O(R)$ time, then apply our *slower* connected components algorithm (cf. Theorem 3.1.1) to output the correct components in $O(\log \log_{m/n} n)$ time, which is $O(R)$ total running time.

3.1.3 Framework

We formulate the problem of computing connected components concurrently as follows: label each vertex v with a unique vertex $v.p$ in its component. Such a labeling gives a constant-time test for whether two vertices v and w are in the same component: they are if and only if $v.p = w.p$. We begin with every vertex self-labeled ($v.p = v$) and successively update labels until there is exactly one label per component.

The labels define a directed graph (*labeled digraph*) with arcs $(v, v.p)$, where $v.p$ is the *parent* of v. We maintain the invariant that the only cycles in the labeled digraph are self-loops (arcs of the form (v, v)). Then this digraph consists of a set of rooted trees, with v a root if and only if $v = v.p$. Some authors call the root of a tree the *leader* of all its vertices. We know of only one algorithm in the literature that creates non-trivial cycles, that of Johnson and Metaxis [89]. Acyclicity implies that when the parent of a root v changes, the new parent of v is not in the tree rooted at v (for any order of the concurrent parent changes). We call a tree *flat* if the root is the parent of every vertex in the tree. Some authors call flat trees *stars*.

In our connected components and spanning forest algorithms (cf. §3.2 and §3.3 in the appendix), we maintain the additional invariant that if the parent of a non-root v changes, its new parent is in the same tree as v (for any order of the parent changes). This invariant implies that the partition of vertices among trees changes only by set union; that is, no parent change moves a proper subtree to another tree. We call this property *monotonicity*. Most of the algorithms in the literature that have a correct efficiency analysis are monotone. Liu and Tarjan [117] analyze some non-monotone algorithms. In our faster connected components algorithm (cf. §3.4), only the preprocessing and postprocessing stages are monotone, which means the execution between these two stages can move subtrees between different trees in the labeled digraph.

3.1.4 Building Blocks

Our algorithms use three standard and one not-so-standard building blocks, which link (sub)trees, flatten trees, alter edges, and add edges, respectively. (Classic PRAM algorithms develop many techniques to make the graph sparser, e.g., in [67, 79, 80], not denser by adding edges.)

We treat each edge $\{v, w\}$ as a pair of oppositely directed arcs (v, w) and (w, v). A *direct link* applies to a graph arc (v, w) such that v is a root and w is not in the tree rooted at v; it makes w the parent of v. A *parent link* applies to a graph arc (v, w) and makes $w.p$ the parent of v; note that v and $w.p$ are not necessarily roots. Concurrent direct links maintain monotonicity while concurrent parent links do not. We add additional constraints to prevent the creation of a cycle in both cases. Specifically, in the case of parent links, if a vertex is not a root in a tree in the labeled digraph, then its level must be strictly lower than the level of its parent (formally stated in Lemma 3.4.4).

Concurrent links can produce trees of arbitrary heights. To reduce the tree heights, we use the *shortcut* operation: for each v do $v.p \coloneqq v.p.p$. One shortcut roughly halves the heights of all trees; $O(\log n)$ shortcuts make all trees flat. Hirschberg et al. [82] introduced shortcutting in their connected components algorithm; it is closely related to the *compress* step in tree contraction [127] and to *path splitting* in disjoint-set union [154].

Our third operation changes graph edges. To *alter* $\{v, w\}$, we replace it by $\{v.p, w.p\}$. Links, shortcuts, and edge alterations suffice to efficiently compute components. Liu and Tarjan [117] analyze simple algorithms that use combinations of our first three building blocks.

To obtain a good bound for small-diameter graphs, we need a fourth operation that adds edges. We *expand* a vertex u by adding an edge $\{u, w\}$ for a neighbor v of u and a neighbor w of v. The key idea for implementing expansion is hashing, which is presented below.

Suppose each vertex owns a block of K^2 processors. For each processor in a block, we index it by a pair $(p, q) \in [K] \times [K]$. For each vertex u, we maintain a size-K table $H(u)$. We choose a random hash function $h : [n] \to [K]$. At the beginning of an expansion, for each graph arc (u, v), we write vertex v into the $h(v)$-th cell of $H(u)$. Then we can expand u as follows: each processor (p, q) reads vertex v from the p-th cell of $H(u)$, reads vertex w from the q-th cell of $H(v)$, and writes vertex w into the $h(w)$-th cell of $H(u)$. For each $w \in H(u)$ after the expansion, $\{u, w\}$ is considered an *added* edge in the graph and is treated the same as any other edge.

The key difference between our hashing-based expansion and that in the MPC algorithms is that a vertex w within distance 2 from u might not be in $H(u)$ after the expansion due to a collision, so crucial to our analysis is the way to handle collisions. All hash functions in this chapter are pairwise

independent, so each processor doing hashing in each round only needs to read two words, which uses $O(1)$ private memory and time.

3.2 An $O(\log d \log\log_{m/n} n)$-time Connected Components Algorithm

In this section we present our connected components algorithm. For simplicity in presentation, we also consider the COMBINING CRCW PRAM [8], whose computational power is between the ARBITRARY CRCW PRAM and MPC. This model is the same as the ARBITRARY CRCW PRAM, except that if several processors write to the same memory cell at the same time, the resulting value is a specified symmetric function (e.g., the sum or min) of the individually written values.

We begin with a simple randomized algorithm proposed by Reif [138] but adapted in our framework, which is called Vanilla algorithm. Our main algorithm will use the same framework with a simple preprocessing method to make the graph denser and an elaborated expansion method to add edges before the direct links. We implement the algorithm on a COMBINING CRCW PRAM and then generalize it to run on an ARBITRARY CRCW PRAM in §3.2.5.

3.2.1 Vanilla Algorithm

In each iteration of Vanilla algorithm (see below), some roots of trees are selected to be leaders, which becomes the parents of non-leaders after the LINK. A vertex u is called a *leader* if $u.l = 1$.

> Vanilla algorithm: repeat {RANDOM-VOTE; LINK; SHORTCUT; ALTER} until no edge exists other than loops.
>
> RANDOM-VOTE: for each vertex u: set $u.l := 1$ with probability $1/2$, and 0 otherwise.
>
> LINK: for each graph arc (v, w): if $v.l = 0$ and $w.l = 1$ then update $v.p$ to w.
>
> SHORTCUT: for each vertex u: update $u.p$ to $u.p.p$.
>
> ALTER: for each edge $e = \{v, w\}$: replace it by $\{v.p, w.p\}$.

It is easy to see that Vanilla algorithm uses $O(m)$ processors and can run on an ARBITRARY CRCW PRAM. We call an iteration of the repeat loop in the algorithm a *phase*. Clearly each phase takes $O(1)$ time. We obtain the following results:

Definition 3.2.1. *A vertex is* ongoing *if it is a root but not the only root in its component, otherwise it is* finished.

Lemma 3.2.2. *At the beginning of each phase, each tree is flat and a vertex is ongoing if and only if it is incident with a non-loop edge.*

Proof. The proof is by induction on phases. At the beginning, each vertex is in a single-vertex tree and the edges between trees are in the original graph, so the lemma holds. Suppose it holds for phase $k-1$. After the LINK in phase k, each tree has height at most two since only non-leader root can update its parent to a leader root. The following SHORTCUT makes the tree flat, then the ALTER moves all the edges to the roots. If a root is not the only root in its component, there must be an edge between it and another vertex not in its tree. □

Lemma 3.2.3. *Given a vertex u, after k phases of Vanilla algorithm, u is ongoing with probability at most $(3/4)^k$.*

Proof. We prove the lemma by an induction on k. The lemma is true for $k = 0$. Suppose it is true for $k-1$. Observe that a non-root can never again be a root. For vertex u to be ongoing after k phases, it must be ongoing after $k-1$ phases. By the induction hypothesis this is true with probability at most $(3/4)^{k-1}$. Furthermore, by Lemma 3.2.2, if this is true, there must be an edge $\{u, v\}$ such that v is ongoing. With probability $1/4$, $u.l = 0$ and $v.l = 1$, then u is finished after phase k. It follows that the probability that, after phase k, u is still ongoing is at most $(3/4)^k$. □

By Lemma 3.2.3, the following corollary is immediate by linearity of expectation and Markov's inequality:

Corollary 3.2.4. *After k phases of Vanilla algorithm, the number of ongoing vertices is at most $(7/8)^k n$ with probability at least $1 - (6/7)^k$.*

By Lemma 3.2.2, Corollary 3.2.4, and monotonicity, we have that Vanilla algorithm outputs the connected components in $O(\log n)$ time with high probability.

3.2.2 Algorithmic Framework

In this section we present the algorithmic framework for our connected components algorithm.

For any vertex v in the current graph, a vertex within distance 1 from v in the current graph (which contains the (altered) original edges and the added edges) is called a *neighbor* of v. The

set of neighbors of every vertex is maintained during the algorithm (see the implementation of the EXPAND).

Connected Components algorithm: PREPARE; repeat {EXPAND; VOTE; LINK; SHORTCUT; ALTER} until no edge exists other than loops.

PREPARE: if $m/n \leq \log^c n$ for given constant c then run $c \log_{8/7} \log n$ phases of Vanilla algorithm.

EXPAND: for each ongoing u: expand the neighbor set of u according to some rule.

VOTE: for each ongoing u: set $u.l$ according to some rule in $O(1)$ time.

LINK: for each ongoing v: for each w in the neighbor set of v: if $v.l = 0$ and $w.l = 1$ then update $v.p$ to w.

The SHORTCUT and ALTER are the same as those in Vanilla algorithm. The LINK is also the same in the sense that in our algorithm the graph arc (v, w) is added during the EXPAND in the form of adding w to the neighbor set of v. Therefore, Lemma 3.2.2 also holds for this algorithm.

The details of the EXPAND and VOTE will be presented in §3.2.3 and §3.2.4, respectively. We call an iteration of the repeat loop after the PREPARE a *phase*. By Lemma 3.2.2, we can determine whether a vertex is ongoing by checking the existence of non-loop edges incident on it, therefore in each phase, the VOTE, LINK, SHORTCUT, and ALTER take $O(1)$ time.

Let $\delta = m/n'$, where n' is the number of ongoing vertices at the beginning of a phase. Our goal in one phase is to reduce n' by a factor of at least a positive constant power of δ with high probability with respect to δ, so we do a PREPARE before the main loop to obtain a large enough δ with good probability:

Lemma 3.2.5. *After the* PREPARE*, if* $m/n > \log^c n$*, then* $m/n' \geq \log^c n$*; otherwise* $m/n' \geq \log^c n$ *with probability at least* $1 - 1/\log^c n$.

Proof. The first part is trivial since the PREPARE does nothing. By Corollary 3.2.4, after $c \log_{8/7} \log n$ phases, there are at most $n/\log^c n$ ongoing vertices with probability at least $1 - (6/7)^{c \log_{8/7} \log n} \geq 1 - 1/\log^c n$, and the lemma follows immediately from $m \geq n$. □

We will be focusing on the EXPAND, VOTE, and LINK, so in each phase it suffices to only consider the induced graph on ongoing vertices with current edges. If no ambiguity, we call this induced graph just the graph, call the current edge just the edge, and call an ongoing vertex just a vertex.

In the following algorithms and analyses, we will use the following assumption for simplicity in the analyses.

Assumption 3.2.6. *The number of ongoing vertices n' is known at the beginning of each phase.*

This holds if running on a COMBINING CRCW PRAM with sum as the combining function to compute n' in $O(1)$ time. Later in §3.2.5 we will show how to remove Assumption 3.2.6 to implement our algorithms on an ARBITRARY CRCW PRAM.

3.2.3 The Expansion

In this section, we present the method EXPAND and show that almost all vertices have a large enough neighbor set after the EXPAND with good probability.

Setup

Blocks. We shall use a pool of m processors to do the EXPAND. We divide these into $m/\delta^{2/3}$ indexed *blocks*, where each block contains $\delta^{2/3}$ indexed processors. Since n' and δ are known at the beginning of each phase (cf. Assumption 3.2.6), if a vertex is assigned to a block, then it is associated with $\delta^{2/3}$ (indexed) processors. We map the n' vertices to the blocks by a random hash function h_B. Each vertex has a probability of being the only vertex mapped to a block, and if this happens then we say this vertex *owns* a block.

Hashing. We use a hash table to implement the neighbor set of each vertex and set the size of the hash table as $\delta^{1/3}$, because we need $\delta^{1/3}$ processors for each cell in the table to do an expansion step (see Step (5a) in the EXPAND). We use a random hash function h_V to hash vertices into the hash tables. Let $H(u)$ be the hash table of vertex u. If no ambiguity, we also use $H(u)$ to denote the set of vertices stored in $H(v)$. If u does not own a block, we think that $H(u) = \emptyset$.

We present the method EXPAND as follows:

EXPAND:

1. Each vertex is either *live* or *dormant* in a step. Mark every vertex as *live* at the beginning.
2. Map the vertices to blocks using h_B. Mark the vertices that do not own a block as *dormant*.
3. For each graph arc (v, w): if v is live before Step (3) then use h_V to hash v into $H(v)$ and w into $H(v)$, else mark w as *dormant*.
4. For each hashing done in Step (3): if it causes a *collision* (a cell is written by different values) in $H(u)$ then mark u as *dormant*.
5. Repeat the following until there is neither live vertex nor hash table getting a new entry:
   (a) For each vertex u: for each v in $H(u)$: if v is dormant before Step (5a) in this iteration then mark u as *dormant*, for each w in $H(v)$: use h_V to hash w into $H(u)$.
   (b) For each hashing done in Step (5a): if it causes a collision in $H(u)$ then mark u as *dormant*.

The first four steps and each iteration of Step (5) in the EXPAND take $O(1)$ time. We call an iteration of the repeat loop in Step (5) a *round*. We say a statement holds before round 0 if it is true before Step (3), it holds in round 0 if it is true after Step (4) and before Step (5), and it holds in round i ($i > 0$) if it is true just after i iterations of the repeat loop in Step (5).

Additional notations. We use $\text{dist}(u, v)$ to denote the *distance* between u and v, which is the length of the shortest path from u to v. We use $B(u, \alpha) = \{v \in V \mid \text{dist}(u, v) \leq \alpha\}$ to represent the set of vertices with distance at most α from u. For any $j \geq 0$ and any vertex u, let $H_j(u)$ be the hash table of u in round j.

Consider a vertex u that is dormant after the EXPAND. We call u *fully dormant* if u is dormant before round 0, i.e., u does not own a block. Otherwise, we call u *half dormant*. For a half dormant u, let $i \geq 0$ be the first round u becomes dormant. For u that is live after the EXPAND, let $i \geq 0$ be the first round that its hash table is the same as the table just before round i. The following lemma shows that in this case, the table of u in round $j < i$ contains exactly the vertices within distance 2^j:

Lemma 3.2.7. *For any vertex u that is not fully dormant, let i be defined above, then it must be that $H_i(u) \subseteq B(u, 2^i)$. Furthermore, for any $j \in [0, i-1]$, $H_j(u) = B(u, 2^j)$.*

Proof. According to the update rule of $H(u)$, it is easy to show that for any integer $j \geq 0$, $H_j(u) \subseteq B(u, 2^j)$ holds by induction. Now we prove that for any $j \in [0, i-1]$, $H_j(u) = B(u, 2^j)$. We claim that for any vertex v, if v is not dormant in round j, then $H_j(v) = B(v, 2^j)$. The base case is when $j = 0$. In this case, $H_0(v)$ has no collision, so the claim holds. Suppose the claim holds for $j-1$, i.e., for any vertex v' which is not dormant in round $j-1$, it has $H_{j-1}(v') = B(v', 2^{j-1})$. Let v be any vertex which is not dormant in round j. Then since there is no collision, $H_j(v)$ should be $\bigcup_{v' \in H_{j-1}(v)} H_{j-1}(v') = \bigcup_{v' \in B(v, 2^{j-1})} B(v', 2^{j-1}) = B(v, 2^j)$. Thus the claim is true, and it implies that for any $j \in [0, i-1]$, $H_j(u) = B(u, 2^j)$. □

Lemma 3.2.8. *The* EXPAND *takes $O(\log d)$ time.*

Proof. By an induction on phases, any path in the previous phase is replaced by a new path with each vertex on the old path replaced by its parent, so the diameter never increases. Since u either is fully dormant or stops its expansion in round i, the lemma follows immediately from Lemma 3.2.7. □

Neighbor Set Size Lower Bound

We want to show that the table of u in round i contains enough neighbors, but u becomes dormant in round i possibly dues to propagations from another vertex in the table of u that is dormant in round $i-1$, which does not guarantee the existence of collisions in the table of u (which implies large size of the table with good probability). We overcome this issue by identifying the maximal-radius ball around u with no collision nor fully dormant vertex, whose size serves as a size lower bound of the table in round i.

Definition 3.2.9. *For any vertex u that is dormant after the* EXPAND*, let r be the minimal integer such that there is no collision nor fully dormant vertex in $B(u, r-1)$.*

Lemma 3.2.10. *If u is fully dormant then $r = 0$. If u is half dormant then $2^{i-1} < r \leq 2^i$.*

Proof. If u is fully dormant, then $r = 0$ since $B(u, 0) = \{u\}$. Suppose u is half dormant. We prove the lemma by induction on i. The lemma holds for $i = 0$ because $r > 0$ and if $r \geq 2$ then u cannot be dormant in Step (3) nor Step (4). The lemma also holds for $i = 1$ because if $r = 1$ then u becomes dormant in Step (3) or Step (4) and if $r \geq 3$ then u cannot be dormant in round 1.

Suppose $i \geq 2$. Assume $r \leq 2^{i-1}$ and let $v \in B(u, r)$ be a fully dormant vertex or a vertex that causes a collision in $B(u, r)$. Assume u becomes dormant after round $i-1$. By Lemma 3.2.7, we

know that $H_{i-1}(u) = B(u, 2^{i-1})$. Since $r \leq 2^{i-1}$, there is no collision in $B(u, r)$ using h_V. Thus, there is a fully dormant vertex v in $B(u, r) \subseteq H_{i-1}(u)$. Consider the first round $j \leq i-1$ that v is added into $H(u)$. If $j = 0$, then u is marked as dormant in round 0 by Step (3). If $j > 0$, then in round j, there is a vertex v' in $H_{j-1}(u)$ such that $v \in H_{j-1}(v')$, and v is added into $H_j(u)$ by Step (5a). In this case, u is marked as dormant in round j by Step (5a). In both cases $j = 0$ and $j > 0$, u is marked as dormant in round $j \leq i-1$ which contradicts with the definition of i. So the only way for u to become dormant in round i is for a vertex v to exist in $H_{i-1}(u)$ which is dormant in round $i-1$. Assume for contradiction that $r > 2^i$, then by Definition 3.2.9 there is no collision nor fully dormant vertex in $B(u, 2^i)$. By the induction hypothesis, there exists either a collision or a fully dormant vertex in $B(v, 2^{i-1})$. By Lemma 3.2.7, we know that $H_{i-1}(u) = B(u, 2^{i-1})$. It means that $B(v, 2^{i-1}) \subseteq B(u, 2^i)$ contains a collision or a fully dormant vertex, contradiction. □

To state bounds simply, let $b \coloneqq \delta^{1/18}$, then hash functions h_B and h_V are from $[n]$ to $[m/b^{12}]$ and from $[n]$ to $[b^6]$, respectively. Note that h_B and h_V need to be independent with each other, but each being pairwise independent suffices, so each processor doing hashing only reads two words.

Lemma 3.2.11. *For any vertex u that is dormant after the* EXPAND, $|B(u, r)| \leq b^2$ *with probability at most b^{-2}.*

Proof. Let j be the maximal integer such that $j \leq d$ and $|B(u, j)| \leq b^2$. We shall calculate the probability of $r \leq j$, which is equivalent to the event $|B(u, r)| \leq b^2$.

The expectation of the number of collisions in $B(u, j)$ using h_V is at most $\binom{b^2}{2}/b^6 \leq b^{-2}/2$, then by Markov's inequality, the probability of at least one collision existing is at most $b^{-2}/2$.

For any u to be fully dormant, at least one of the $n' - 1$ vertices other than u must have hash value $h_B(u)$. By union bound, the probability of u being fully dormant is at most

$$\frac{n'-1}{m/b^{12}} \leq \frac{n'}{n'b^6} = \frac{1}{b^6}, \tag{3.1}$$

where the first inequality follows from $m/n' = b^{18}$. Taking union bound over all vertices in $B(u, j)$ we obtain the probability as at most $b^2 \cdot b^{-6} = b^{-4}$.

Therefore, for any dormant vertex u, $\Pr[|B(u, r)| \leq b^2] = \Pr[r \leq j] \leq b^{-2}/2 + b^{-4} \leq b^{-2}$ by a union bound. □

Based on Lemma 3.2.11, we shall show that any dormant vertex has large enough neighbor set after the EXPAND with good probability. To prove this, we need the following lemma:

Lemma 3.2.12. $|B(u, r-1)| \leq |H_i(u)|$.

Proof. Let w be a vertex in $B(u, r-1)$. If $\mathrm{dist}(w, u) \leq 2^{i-1}$ then $w \in H_{i-1}(u)$ by Lemma 3.2.7 and Definition 3.2.9, and position $h_V(w)$ in $H_i(u)$ remains occupied in round i. Suppose $\mathrm{dist}(w, u) > 2^{i-1}$. Let x be a vertex on the shortest path from u to w and x satisfies $\mathrm{dist}(x, w) = 2^{i-1}$. We obtain $B(x, 2^{i-1}) \subseteq B(u, r-1)$ since any $y \in B(x, 2^{i-1})$ has $\mathrm{dist}(y, u) \leq \mathrm{dist}(y, x) + \mathrm{dist}(x, u) \leq 2^{i-1} + r - 1 - 2^{i-1}$, and thus $w \in H_{i-1}(x)$ and $x \in H_{i-1}(u)$. So position $h_V(w)$ in $H_i(u)$ remains occupied in round i as a consequence of Step (5a). Therefore Lemma 3.2.12 holds. □

Lemma 3.2.13. *After the* EXPAND*, for any dormant vertex u, $|H(u)| < b$ with probability at most b^{-1}.*

Proof. By Lemma 3.2.11, $|B(u, r)| > b^2$ with probability at least $1 - b^{-2}$. If this event happens, u must be half dormant. In the following we shall prove that $|H(u)| \geq b$ with probability at least $1 - b^{-4}$ conditioned on this, then a union bound gives the lemma.

If $i = 0$ then $r = 1$ by Lemma 3.2.10, which gives $|B(u, 1)| > b^2 \geq b$. Consider hashing arbitrary b vertices of $B(u, 1)$. The expectation of the number of collisions among them is at most $b^2/b^6 = b^{-4}$, then by Markov's inequality the probability of at least one collision existing is at most b^{-4}. Thus with probability at least $1 - b^{-4}$ these b vertices all get distinct hash values, which implies $|H(u)| \geq b$. So the lemma holds.

Suppose $i \geq 1$. If $|B(u, r-1)| \geq b$ then the lemma holds by Lemma 3.2.12. Otherwise, since $|B(u, r)| > b^2$, by Pigeonhole principle there must exist a vertex w at distance $r - 1$ from u such that $|B(w, 1)| > b$. Then by an argument similar to that in the second paragraph of this proof we have that $H_0(w) \geq b$ with probability at least $1 - b^{-4}$. If $i = 1$ then $r = 2$. Since $w \in H_0(u)$ dues to Definition 3.2.9, at least b positions will be occupied in $H_1(u)$, and the lemma holds.

Suppose $i \geq 2$. Let w_0 be a vertex on the shortest path from u to w such that $\mathrm{dist}(w_0, w) = 1$. Recursively, for $j \in [1, i-2]$, let w_j be a vertex on this path such that $\mathrm{dist}(w_{j-1}, w_j) = 2^j$. We have that

$$\mathrm{dist}(w, w_{i-2}) = \mathrm{dist}(w, w_0) + \sum_{j \in [i-2]} \mathrm{dist}(w_{j-1}, w_j) = 2^{i-1} - 1.$$

Thus

$$\mathrm{dist}(w_{i-2}, u) = r - 1 - \mathrm{dist}(w, w_{i-2}) = r - 2^{i-1} \leq 2^{i-1},$$

where the last inequality follows from Lemma 3.2.10. It follows that $w_{i-2} \in H_{i-1}(u)$ by Lemma 3.2.7 and Definition 3.2.9. We also need the following claim, which follows immediately from $\mathrm{dist}(w_j, u) \leq r - 2^j$ and Definition 3.2.9:

Claim 3.2.14. *$H_j(w_j) = B(w_j, 2^j)$ for all $j \in [0, i-2]$.*

Now we claim that $|H_{i-1}(w_{i-2})| \geq b$ and prove it by induction, then $|H_i(u)| \geq b$ holds since $w_{i-2} \in H_{i-1}(u)$. $|H_1(w_0)| \geq b$ since $w \in H_0(w_0)$ and $|H_0(w)| \geq b$. Assume $|H_j(w_{j-1})| \geq b$. Since $w_{j-1} \in H_j(w_j)$ by $\text{dist}(w_{j-1}, w_j) = 2^j$ and Claim 3.2.14, we obtain $|H_{j+1}(w_j)| \geq b$. So the induction holds and $|H_{i-1}(w_{i-2})| \geq b$.

By the above paragraph and the first paragraph of this proof we proved Lemma 3.2.13. □

3.2.4 The Voting

In this section, we present the method VOTE and show that the number of ongoing vertices decreases by a factor of a positive constant power of b with good probability.

VOTE: for each vertex u: initialize $u.l \coloneqq 1$,

1. If u is live after the EXPAND then for each vertex v in $H(u)$: if $v < u$ then set $u.l \coloneqq 0$.
2. Else set $u.l \coloneqq 0$ with probability $1 - b^{-2/3}$.

There are two cases depending on whether u is live. In Case (1), by Lemma 3.2.7, $H(u)$ must contain all the vertices in the component of u, and so does any vertex in $H(u)$, because otherwise u is dormant. We need to choose the same parent for all the vertices in this component, which is the minimal one in this component as described: a vertex u that is not minimal in its component would have $u.l = 0$ by some vertex v in $H(u)$ smaller than u. Thus all live vertices become finished in the next phase.

In Case (2), u is dormant. Then by Lemma 3.2.13, $|H(u)| \geq b$ with probability at least $1 - b^{-1}$. If this event happens, the probability of no leader in $H(u)$ is at most $(1 - b^{-2/3})^b \leq \exp(-b^{-1/3}) \leq b^{-1}$.

The number of vertices in the next phase is the sum of: (i) the number of dormant leaders, (ii) the number of non-leaders u with $|H(u)| < b$ and no leader in $H(u)$, and (iii) the number of non-leaders u with $|H(u)| \geq b$ and no leader in $H(u)$. We have that the expected number of vertices in the next phase is at most

$$n' \cdot (b^{-2/3} + b^{-1} + (1 - b^{-1}) \cdot b^{-1}) \leq n' \cdot b^{-1/2}.$$

By Markov's inequality, the probability of having more than $n' \cdot b^{-1/4}$ vertices in the next phase is

at most

$$\frac{n' \cdot b^{-1/2}}{n' \cdot b^{-1/4}} \leq b^{-1/4}. \tag{3.2}$$

3.2.5 Removing the Assumption

In this section, we remove Assumption 3.2.6 which holds on a COMBINING CRCW PRAM, thus generalize the algorithm to run on an ARBITRARY CRCW PRAM.

Recall that we set $b = \delta^{1/18}$ where $\delta = m/n'$ is known. The key observation is that in each phase, all results still hold when we use any $\widetilde{n}$ to replace n' as long as $\widetilde{n} \geq n'$ and b is large enough. This effectively means that we use $b = (m/\widetilde{n})^{1/18}$ for the hash functions h_B and h_V. This is because the only places that use n' as the number of vertices in a phase are:

1. The probability of a dormant vertex u having $B(u, r) \leq b^2$ (Lemma 3.2.11). Using $\widetilde{n}$ to rewrite Inequality (3.1), we have:

   $$\frac{n' - 1}{m/b^{12}} = \frac{n' - 1}{\widetilde{n}b^6} \leq \frac{n'}{n'b^6} = \frac{1}{b^6},$$

   followed from $m/\widetilde{n} = b^{18}$ and $\widetilde{n} \geq n'$. Therefore Lemma 3.2.11 still holds.

2. The probability of having more than $n' \cdot b^{-1/4}$ vertices in the next phase. Now we measure the progress by the decreasing in $\widetilde{n}$. The expected number of vertices in the next phase is still at most $n' \cdot b^{-1/4}$, where n' is the exact number of vertices. Therefore by Markov's inequality we can rewrite Inequality (3.2) as:

   $$\frac{n' \cdot b^{-1/2}}{\widetilde{n} \cdot b^{-1/4}} \leq b^{-1/4},$$

   which is the probability of having more than $\widetilde{n} \cdot b^{-1/4}$ vertices in the next phase.

As a conclusion, if $\widetilde{n} \geq n'$ and b is large enough in each phase, all analyses still apply. Let c be the value defined in PREPARE. We give the *update rule* of $\widetilde{n}$:

> Update rule of $\widetilde{n}$:
>
> If $m/n \leq \log^c n$ then set $\widetilde{n} := n/\log^c n$ for the first phase (after the PREPARE), else set $\widetilde{n} := n$.
>
> At the beginning of each phase, update $\widetilde{n} := \widetilde{n}/b^{1/4}$ then update $b := (m/\widetilde{n})^{1/18}$.

So $b \geq \log^{c/18} n$ is large enough. By the above argument, we immediately have the following:

Lemma 3.2.15. *Let $\widetilde{n}$ and n' be as defined above in each phase. If $\widetilde{n} \geq n'$ in a phase, then with probability at least $1 - b^{-1/4}$, $\widetilde{n} \geq n'$ in the next phase.*

By an induction on phases, Lemma 3.2.15, and a union bound, the following lemma is immediate:

Lemma 3.2.16. *Given any integer $t \geq 2$, if $\widetilde{n} \geq n'$ in the first phase, then $\widetilde{n} \geq n'$ in all phases before phase t with probability at least $1 - \sum_{i \in [t-2]} {b_i}^{-1/4}$, where b_i is the parameter b in phase $i \geq 1$*

.

3.2.6 Running Time

In this section, we compute the running time of our algorithm and the probability of achieving it.

Lemma 3.2.17. *After the* PREPARE*, if $\widetilde{n} \geq n'$ in each phase, then the algorithm outputs the connected components in $O(\log \log_{m/n_1} n)$ phases, where n_1 is the $\widetilde{n}$ in the first phase.*

Proof. Let n_i be the $\widetilde{n}$ in phase i. By the update rule of $\widetilde{n}$, we have $n_{i+1} \leq n_i/(m/n_i)^{1/72}$, which gives $m/n_{t+1} \geq (m/n_1)^{(73/72)^t}$. If $t = \lceil \log_{73/72} \log_{m/n_1} m \rceil + 1$ then $n_{t+1} < 1$, which leads to $n' = 0$ at the beginning of phase $t+1$. By Lemma 3.2.2 and monotonicity, the algorithm terminates and outputs the correct connected components in this phase since no parent changes. □

Proof of Theorem 3.1.1. We set $c = 100$ in the PREPARE. If $m/n > \log^c n$, then $\widetilde{n} = n$ in the first phase by Lemma 3.2.5 and the update rule. Since $\delta = m/n_1 \geq \log^c n$, we have that $b \geq \delta^{1/18} \geq \log^5 n$ in all phases. By Lemma 3.2.16, $\widetilde{n} \geq n'$ in all phases before phase $\log n$ with probability at least $1 - \log n \cdot b^{-1/4} = 1 - 1/\mathrm{poly}(m \log n/n)$. If this event happens, by Lemma 3.2.17, the number of phases is $O(\log \log_{m/n} n)$. By Lemma 3.2.8, the total running time is $O(\log d \log \log_{m/n} n)$ with good probability.

If $m/n \leq \log^c n$, then by Lemma 3.2.5 and the update rule of $\widetilde{n}$, after the PREPARE which takes time $O(\log \log n)$, with probability at least $1 - 1/\log^c n$ we have $m/n_1 \geq \log^c n$. If this happens, by the argument in the previous paragraph, with probability at least $1 - 1/\mathrm{poly}(m/n_1 \cdot \log n)$, Connected Components algorithm takes time $O(\log d \log \log_{m/n} n)$. Taking a union bound, we obtain that with probability at least $1 - 1/\mathrm{poly}(m/n_1 \cdot \log n) - 1/\log^c n \geq 1 - 1/\mathrm{polylog}(n) = 1 - 1/\mathrm{poly}(m \log n/n)$, the total running time is $O(\log \log n) + O(\log d \cdot \log \log_{m/n_1} n) = O(\log d \log \log_{m/n} n)$. □

3.3 Spanning Forest Algorithm

Many existing connected components algorithm can be directly transformed into a spanning forest algorithm. For example in Reif's algorithm [138], one can output the edges corresponding to leader contraction in each step to the spanning forest. However this is not the case here as we also add

edges to the graph. The solution in [10] uses several complicated ideas including merging local shortest path trees, which heavily relies on computing the minimum function in constant time – a goal easily achieved by sorting on an MPC but requiring $\Omega(\log \log n)$ time on a PRAM [64].

We show how to modify our connected components algorithm with an extended expansion procedure to output a spanning forest. Observe that in our previous expansion procedure, if a vertex u does not stop expansion in step i, then the space corresponding to u contains all the vertices within distance 2^i from u. Based on this, we are able to maintain the distance from u to the closest leader in $O(\log d)$ time by the distance doubling argument used before. After determining the distance of each vertex to its closest leader, for each edge (u, v), if u has distance x to the closest leader, and v has distance $x - 1$ to the closest leader, then we can set v as the parent of u and add edge (u, v) to the spanning forest. (If there are multiple choices of v, we can choose arbitrary one.) Since this parents assignment does not induce any cycle, we can find a subforest of the graph. If we contract all vertices in each tree of such subforest to the unique leader in that tree (which also takes $O(\log d)$ time by shortcutting as any shortest path tree has height at most d), the problem reduces to finding a spanning forest of the contracted graph. Similar to the analysis of our connected components algorithm, we need $O(\log d)$ time to find a subforest in the contracted graph and the number of contraction rounds is $O(\log \log_{m/n} n)$. Thus, the total running time is asymptotically the same as our connected components algorithm.

Since the EXPAND method will add new edges which are not in the input graph, our connected components algorithm cannot give a spanning forest algorithm directly. To output a spanning forest, we only allow direct links on graph arcs of the input graph. However, we want to link many graph arcs concurrently to make a sufficient progress. We extend the EXPAND method to a new subroutine which can link many graph arcs concurrently. Furthermore, after applying the subroutine, there is no cycle induced by link operations, and the tree height is bounded by the diameter of the input graph.

For each arc e in the current graph, we use $\widehat{e}$ to denote the original arc in the input graph that is altered to e during the execution. Each edge processor is identified by an original arc $\widehat{e}$ and stores the corresponding e in its private memory during the execution. To output the spanning forest, for a original graph arc $\widehat{e} = (v, w)$, if at the end of the algorithm, $\widehat{e}.f = 1$, then it denotes that the graph edge $\{v, w\}$ is in the spanning forest. Otherwise if both $\widehat{e}.f = 0$ and $\widehat{e}'.f = 0$ where $\widehat{e}'$ denotes a graph arc (w, v), then the edge $\{v, w\}$ is not in the spanning forest.

3.3.1 Vanilla Algorithm for Spanning Forest

Firstly, let us see how to extend Vanilla algorithm to output a spanning forest. The extended algorithm is called Vanilla-SF algorithm (see below).

The RANDOM-VOTE and SHORTCUT are the same as those in Vanilla algorithm. In the ALTER we only alter the current edge as in Vanilla algorithm but keep the original edge untouched. We add a method MARK-EDGE and an attribute e for each vertex v to store the current arc that causes the link on v, then $v.\widehat{e}$ is the original arc in the input graph corresponding to $v.e$. The LINK is the same except that we additionally mark the original arc in the forest using attribute f if the corresponding current arc causes the link.

Vanilla-SF algorithm: repeat {RANDOM-VOTE; MARK-EDGE; LINK; SHORTCUT; ALTER} until no edge exists other than loops.

MARK-EDGE: for each current graph arc $e = (v, w)$: if $v.l = 0$ and $w.l = 1$ then update $v.e$ to e and update $v.\widehat{e}$ to $\widehat{e}$.

LINK: for each ongoing u: if $u.e = (u, w)$ exists then update $u.p$ to w and update $u.\widehat{e}.f \coloneqq 1$.

The digraph defined by the labels is exactly the same as in Vanilla algorithm, therefore Lemma 3.2.2 holds for Vanilla-SF algorithm. It is easy to see that Vanilla-SF algorithm uses $O(m)$ processors and can run on an ARBITRARY CRCW PRAM. Each phase takes $O(1)$ time.

Definition 3.3.1. *For any positive integer j, at the beginning of phase j, let F_j be the graph induced by all the edges corresponding to all the arcs $\widehat{e}$ with $\widehat{e}.f = 1$.*

By the execution of the algorithm, for any positive integers $i \leq j$, the set of the edges in F_i is a subset of the set of the edges in F_j.

Lemma 3.3.2. *For any positive integer j, at the beginning of phase j, each vertex u is in the component of $u.p$ in F_j.*

Proof. The proof is by induction. In the first phase, the lemma is vacuously true since $u.p = u$ for all vertices u. Now, suppose the lemma is true at the beginning of phase i. In phase i, if $u.p$ does not change, the claim is obviously true. Otherwise, there are two cases: (i) $u.p$ is changed in the LINK, or (ii) $u.p$ is changed in the SHORTCUT. If $u.p$ is changed in the SHORTCUT, then by Lemma 3.2.2, the original $u.p.p$ is changed in the LINK, and since u and the original $u.p$ is in the same component of F_i, we only need to show that the LINK does not break the invariant. In the LINK, if $u.p$ is updated

to w, there is a current graph arc $u.e = (u, w)$, and thus there exists a graph arc $u.\widehat{e} = (x, y)$ in the input graph such that $x.p = u$ and $y.p = w$ at the beginning of phase i. By the induction hypothesis, x and u are in the same component of F_i and thus of F_{i+1}. Similar argument holds for y and w. Since $u.\widehat{e}.f$ is set to 1 in the LINK, x and y are in the same component in F_{i+1}. Thus, u and w are in the same component in F_{i+1}. □

Lemma 3.3.3. *For any positive integer j, F_j is a forest. And in each tree of F_j, there is exactly one root.*[8]

Proof. By the LINK, every time the size of $\{u \mid u.p = u\}$ decreases by 1, the size of $\{\widehat{e} \mid \widehat{e}.f = 1\}$ increases by 1. Thus the size of $\{\widehat{e} \mid \widehat{e}.f = 1\}$ is exactly $n - |\{u \mid u.p = u\}|$, which induce at least $|\{u \mid u.p = u\}|$ components in F_j. By Lemma 3.3.2, there are at most $|\{u \mid u.p = u\}|$ components in F_j. So there are exactly $|\{u \mid u.p = u\}|$ components in F_j and each such component must be a tree, and each component contains exactly one vertex u with $u.p = u$. □

3.3.2 Algorithmic Framework

In this section, we show how to extend our Connected Components algorithm to a spanning forest algorithm.

> Spanning Forest algorithm: FOREST-PREPARE; repeat {EXPAND; VOTE; TREE-LINK; TREE-SHORTCUT; ALTER} until no edge exists other than loops.
>
> FOREST-PREPARE: if $m/n \leq \log^c n$ for given constant c then run Vanilla-SF algorithm for $c \log_{8/7} \log n$ phases.
>
> TREE-LINK: for each ongoing u: update $u.p$, $u.e$, $u.\widehat{e}$, and $u.\widehat{e}.f$ according to some rule.
>
> TREE-SHORTCUT: repeat {SHORTCUT} until no parent changes.

The EXPAND, VOTE, SHORTCUT, and ALTER are the same as in our connected component algorithm.

Similarly to that in Connected Components algorithm, the FOREST-PREPARE makes the number of ongoing vertices small enough with good probability. As analyzed in §3.2.3 and §3.2.4, the EXPAND takes $O(\log d)$ time, and VOTE takes $O(1)$ time. TREE-SHORTCUT takes $O(\log d')$ time where d' is the height of the highest tree after the TREE-LINK. Later, we will show that the height d' is $O(d)$, giving $O(\log d)$ running time for each phase.

[8]The *tree* we used here in the forest of the graph should not be confused with the tree in the digraph defined by labels.

3.3.3 The Tree Linking

In this section, we present the method TREE-LINK. The purpose of the TREE-LINK is two-fold: firstly, we want to add some edges to expand the current forest (a subgraph of the input graph); secondly, we need the information of these added edges to do link operation of ongoing vertices to reduce the total number of ongoing vertices. For simplicity, all vertices discussed in this section are ongoing vertices in the current phase.

Similar to the EXPAND, we shall use a pool of m processors to do the TREE-LINK. Let n' be the number of vertices. We set all the parameters as the same as in the EXPAND: $\delta = m/n'$, the processors are divided into $m/\delta^{2/3}$ indexed blocks where each block contains $\delta^{2/3}$ indexed processors, and both h_B, h_V are the same hash functions used in the EXPAND. Comparing to Connected Components algorithm, we store not only the final hash table $H(u)$ of u, but also the hash table $H_j(u)$ of u in each round $j \geq 0$. (In Connected Components algorithm, H_j is an analysis tool only.) Let T denote the total number of rounds in Step (5) of the EXPAND.

We present the method TREE-LINK as follows, which maintains: (i) the largest integer $u.\alpha$ for each vertex u such that there is neither collisions, leaders, nor fully dormant vertices in $B(u, u.\alpha)$; (ii) $u.\beta$ as the distance to the nearest leader v (if exists in $B(u, u.\alpha + 1)$) from u; (iii) a hash table $Q(u)$ to store all vertices in $B(u, u.\alpha)$, which is done by reducing the radius by a factor of two in each iteration and attempting to expand the current $Q(u)$ to a temporary hash table $Q'(u)$.

For simplicity, if there is no ambiguity, we also use $Q(u)$ to denote the set of vertices which are stored in the table $Q(u)$. We call each iteration j from T down to 0 in Step (2) a *round*.

TREE-LINK:

1. For each vertex u:
   (a) If $u.l = 1$, set $u.\alpha \coloneqq -1$ and set hash table $Q(u) \coloneqq \emptyset$.
   (b) If $u.l = 0$ and u does not own a block, set $u.\alpha \coloneqq -1$ and set $Q(u) \coloneqq \emptyset$.
   (c) If $u.l = 0$ and u owns a block, set $u.\alpha \coloneqq 0$ and use h_V to hash u into $Q(u)$.
2. For $j = T \to 0$: for each vertex u with $u.\alpha \geq 0$: if every v in table $Q(u)$ is live in round j of Step (5) of the EXPAND:
   (a) Initialize $Q'(u) \coloneqq \emptyset$.
   (b) For each v in $Q(u)$: for each w in $H_j(v)$: use h_V to hash w into $Q'(u)$.
   (c) If there is neither collisions nor leaders in $Q'(u)$ then set $Q(u)$ to be $Q'(u)$ and increase $u.\alpha$ by 2^j.
3. For each current graph arc (v, w): if $v.l = 1$ then mark w as a *leader-neighbor*.
4. For each vertex u:
   (a) If $u.l = 1$ then set $u.\beta \coloneqq 0$.
   (b) If $u.l = 0$ and $Q(u)$ contains a vertex w marked as a leader-neighbor then set $u.\beta \coloneqq u.\alpha + 1$.
5. For each current graph arc $e = (v, w)$: if $v.\beta = w.\beta + 1$ then write e into $v.e$ and write $\widehat{e}$ to $v.\widehat{e}$.
6. For each vertex u: if $u.e = (u, w)$ exists then update $u.p$ to w and update $u.\widehat{e}.f \coloneqq 1$.

Lemma 3.3.4. *In any round in the* TREE-LINK*, for any vertex u, $Q(u) = B(u, u.\alpha)$. Furthermore, at the end of the* TREE-LINK*, $u.\alpha$ is the largest integer such that there is neither collisions nor dormant vertices in $B(u, u.\alpha)$.*

Proof. Firstly, we show that $Q(u) = B(u, u.\alpha)$. Our proof is by an induction on j of Step (2). The base case holds since before Step (2), the initialization of $Q(u)$ and $u.\alpha$ satisfy the claim. Now suppose $Q(u) = B(u, u.\alpha)$ at the beginning of round j of Step (2). There are two cases. The first case is that $Q(u)$ does not change in this round. In this case the claim is true by the induction hypothesis.

The second case is that $Q(u)$ will be set to $Q'(u)$ in Step (2c). Since there is no collision in $Q'(u)$, we have $Q'(u) = \bigcup_{v \in Q(u)} H_j(v)$, where $Q(u)$ is not set in Step (2c) yet. Since every v in $Q(u)$ is live in round j of Step (5) of the EXPAND, we have $H_j(v) = B(v, 2^j)$ according to Lemma 3.2.7. Together with the induction hypothesis, $Q'(u) = \bigcup_{v \in B(u,u.\alpha)} B(v, 2^j) = B(u, u.\alpha + 2^j)$. Thus by Step (2c), after updating $Q(u)$ and $u.\alpha$, the first part of the lemma holds. Now we prove the second part of the lemma. For convenience in the notation, we say that $B(u, \alpha)$ satisfies property $\mathcal{P}$ if and only if there is neither collisions, leaders, nor fully dormant vertices in $B(u, \alpha)$. We shall show that at the end of the TREE-LINK, $u.\alpha$ is the largest integer such that $B(u, u.\alpha)$ satisfies $\mathcal{P}$. For any vertex u, if $u.l = 0$ and u is fully dormant, or $u.l = 1$, then the claim holds due to Step (1). Consider a vertex u with $u.l = 0$ and u owning a block. Our proof is by induction on j of Step (2). We claim that at the end of round j of Step (2), the following two invariants hold: (i) $B(u, u.\alpha + 2^j)$ does not satisfy $\mathcal{P}$; (ii) $B(u, u.\alpha)$ satisfies $\mathcal{P}$. The base case is before Step (2). There are two cases: if u is live after round T of Step (5) of the EXPAND, then there is a leader in $B(u, 2^{T+1})$ by the VOTE; otherwise, there is either a fully dormant vertex or a collision in $B(u, 2^{T+1})$. Thus, the invariants hold for the base case.

Now suppose the invariants hold after round j of Step (2). In round $j - 1$, there are two cases. In the first case, $Q(u)$ is set to $Q'(u)$. This means that before Step (2c), $\forall v \in Q(u) = B(u, u.\alpha)$, $B(v, 2^{j-1})$ satisfies $\mathcal{P}$. Thus, $B(u, u.\alpha + 2^{j-1})$ satisfies $\mathcal{P}$. Notice that by the induction, $B(u, u.\alpha + 2^j)$ does not satisfy $\mathcal{P}$. Therefore, the invariants hold after updating $Q(u)$ and $u.\alpha$ in Step (2c). In the second case, $Q(u)$ remains unchanged. There exists $v \in Q(u) = B(u, u.\alpha)$ such that $B(v, 2^{j-1})$ does not satisfy $\mathcal{P}$ which means that $B(u, u.\alpha + 2^{j-1})$ does not satisfy $\mathcal{P}$. Since $Q(u)$ and $u.\alpha$ can only change together, $B(u, u.\alpha)$ satisfies $\mathcal{P}$. The invariants also hold. Therefore, after round $j = 0$, $B(u, u.\alpha)$ satisfies $\mathcal{P}$ and $B(u, u.\alpha + 1)$ does not satisfy $\mathcal{P}$, giving the second part of the lemma. □

Lemma 3.3.5. *For any vertex u, if $u.\beta$ is updated, then $u.\beta = \min_{v:v.l=1} dist(u, v)$.*

Proof. For any vertex u with $u.l = 1$, $u.\beta$ is set to 0. Now, consider a vertex u with $u.l = 0$. By Lemma 3.3.4, $Q(u) = B(u, u.\alpha)$ and there is no leader in $B(u, u.\alpha)$. If $w \in B(u, u.\alpha)$ is marked as a leader-neighbor, there is a vertex v with $v.l = 1$ such that $v \in B(u, u.\alpha + 1)$. Thus, in this case $u.\beta = u.\alpha + 1 = \min_{v:v.l=1} \text{dist}(u, v)$. □

The following lemma shows that the construction of the tree in Steps (5,6) using the β values is valid.

Lemma 3.3.6. *For any vertex u, if $u.\beta > 0$, then there exists an edge $\{u, w\}$ such that $w.\beta = u.\beta - 1$.*

Proof. Consider a vertex u with $u.\beta = 1$. Then $u.\alpha = 0$, and by Lemma 3.3.4, $Q(u) = B(u, 0) = \{u\}$. Thus, u is marked as a leader-neighbor which means that there is a graph edge $\{u, v\}$ where v is a leader. Notice that $v.\beta = 0$. So the lemma holds for u with $u.\beta = 1$.

Consider a vertex u with $u.\beta > 1$. By Lemma 3.3.5, there is a leader v such that $\mathrm{dist}(u, v) = u.\beta$. Let $\{u, w\}$ be a graph edge with $\mathrm{dist}(w, v) = u.\beta - 1$, which must exist by Step (5). It suffices to show that $w.\beta = u.\beta - 1$. Since $B(w, u.\beta - 1)$ contains a leader v, and $B(w, u.\beta - 2) \subseteq B(u, u.\beta - 1) = B(u, u.\alpha)$ which does not contain a leader by Lemma 3.3.4, we have that $w.\alpha = u.\beta - 2$. Let $\{x, v\}$ be a graph edge such that $\mathrm{dist}(w, x) = u.\beta - 2$, which must exist by Step (5). Then, x is marked as a leader-neighbor and x is in $B(w, w.\alpha)$. Hence $w.\beta = w.\alpha + 1 = u.\beta - 2 + 1 = u.\beta - 1$. □

The above lemma implies that if $u.\beta > 0$ then u is a non-root in the next phase due to Steps (5,6):

Corollary 3.3.7. *For any vertex u, if $u.\beta > 0$, then u is finished in the next phase.*

Lemma 3.3.8. *The height of any tree is $O(d)$ after the* TREE-LINK.

Proof. If $u.p$ is updated to w, then $u.\beta = w.\beta + 1$ by Step (6). Thus, the height of a tree is at most $\max_u u.\beta + 1$. By Lemma 3.3.5 and the fact that an ALTER and adding edges never increase the diameter, the height of any tree is at most d. □

As mentioned at the end of §3.3.2, by the above lemma and Lemma 3.2.8, the following is immediate:

Corollary 3.3.9. *Each phase of the algorithm takes $O(\log d)$ time.*

Similar to Definition 3.3.1 and Lemma 3.3.3, we can show the following, which guarantees the correctness of Spanning Forest algorithm:

Lemma 3.3.10. *At the end of the* TREE-LINK*, all the edges $\widehat{e}$ with $\widehat{e}.f = 1$ constitute a forest. And in each tree of the forest, there is exactly one root.*

3.3.4 Running Time

Now let us analyze the number of vertices in the next phase. If a vertex u is live after the EXPAND, then it is finished in the next phase. Thus all the vertices in the next phase are dormant after the EXPAND in this phase. Consider a dormant vertex u, and let r be that defined in Definition 3.2.9 with respect to u.

Lemma 3.3.11. *For a dormant non-leader u, if there is a leader in $B(u, r)$, then u is finished in the next phase.*

Proof. Let v be the leader closest to u. Since v is in $B(u,r)$, by the definition of r and Lemma 3.3.4, we have that $u.\alpha = \text{dist}(u,v) - 1$, and $Q(u) = B(u, \text{dist}(u,v) - 1)$ which contains a leader-neighbor. By Step (4b) in the TREE-LINK, $u.\beta = \text{dist}(u,v) > 0$. By Corollary 3.3.7, u is finished in the next phase. □

Using the above lemma and Corollary 3.3.9, the remaining analysis is almost identical to that in Connected Components algorithm.

Proof of Theorem 3.1.2. By Lemma 3.2.11, we have that $|B(u,r)| \leq b^2$ with probability at most b^{-2}. Conditioned on $|B(u,r)| > b^2$, the probability that $B(u,r)$ contains no leader is at most b^{-1}. The number of vertices in the next phase is at most the sum of (i) the number of dormant leaders, (ii) the number of vertices u with $|B(u,r)| \leq b^2$, and (iii) the number of vertices u with $|B(u,r)| > b^2$ and no leader in $B(u,r)$. Thus, the probability that a dormant vertex u is in the next phase is at most $b^{-2/3} + b^{-2} + (1 - b^{-2}) \cdot b^{-1} \leq b^{-1/2}$. By Markov's inequality, the probability of having more than $n' \cdot b^{-1/4}$ vertices in the next phase is at most $b^{-1/4}$.

Finally, using exactly the same analyses in §3.2.5 and §3.2.6, we obtain that the algorithm runs on an ARBITRARY CRCW PRAM and outputs the spanning forest. Moreover, with probability $1 - 1/\text{poly}(m \log n/n)$, the number of phases is $O(\log \log_{m/n} n)$ thus the total running time is $O(\log d \log \log_{m/n} n)$ (cf. Corollary 3.3.9). □

3.4 Faster Connected Components Algorithm

In this section we give the full proof of Theorem 3.1.3 by presenting a faster algorithm (see below) for connected components with a detailed analysis.

> Faster Connected Components algorithm: COMPACT; repeat {EXPAND-MAXLINK} until the graph has diameter $O(1)$ and all trees are flat; run Connected Components algorithm from §3.2.

Each iteration of the repeat loop is called a *round*. The *break condition* that the graph has diameter $O(1)$ and all trees are flat shall be tested at the end of each round. The main part of this section is devoted to the repeating EXPAND-MAXLINK. Before that, we explain the method COMPACT.

COMPACT: PREPARE; rename each ongoing vertex to a distinct id in $[2m/\log^c n]$ and assign it a block of size $\max\{m/n, \log^c n\}/\log^2 n$.

Recall that in the PREPARE in §3.2.2, it is guaranteed that the number of ongoing vertices is at most $m/\log^c n$ with good probability, then we use hashing to assign them blocks. The problem of this method is that in this section, we need, with high probability, all roots to own blocks as we need to union bound (as the events are dependent) all roots on all shortest paths (see §3.1.2), but the hashing method only gives that each root owns a block with probability $1 - 1/\text{polylog}(n)$. To solve this problem, we need the following tool when implementing COMPACT:

Definition 3.4.1. *Given a length-n array A which contains k* distinguished *elements,* approximate compaction *is to map all the distinguished elements in A one-to-one to an array of length $2k$.*

Lemma 3.4.2 ([74])**.** *There is an ARBITRARY CRCW PRAM algorithm for approximate compaction that runs in $O(\log^* n)$ time and uses $O(n)$ processors with high probability. Moreover, if using $n \log n$ processors, the algorithm runs in $O(1)$ time.*[9]

Lemma 3.4.3. *With good probability,* COMPACT *renames each ongoing vertex a distinct id in $[2m/\log^c n]$, assigns each of them a block of size $\max\{m/n, \log^c n\}/\log^2 n$, runs in $O(\log\log_{m/n} n)$ time, and uses $O(m)$ processors in total.*

Proof. If $m/n > \log^c n$, then the COMPACT does not run Vanilla algorithm and each (ongoing) vertex already has a distinct id in $[n] \subseteq [2m/\log^c n]$. Since there are $\Theta(m)$ processors in total, it is easy to assign each vertex a block of size $m/n/\log^2 n$, which is $\max\{m/n, \log^c n\}/\log^2 n$.

Else if $m/n \leq \log^c n$, the COMPACT runs Vanilla algorithm for $c \log_{8/7} \log n = O(\log\log_{m/n} n)$ phases such that the number of ongoing vertices is at most $m/\log^c n$ with good probability (cf. Corollary 3.2.4). Conditioned on this happening, the COMPACT uses approximate compaction to compact all the at most $m/\log^c n$ ongoing vertices (distinguished elements) to an array of length at most $2m/\log^c n$, whose index in the array serves as a distinct id. By Lemma 3.4.2, this runs in $O(\log^* n)$ time with probability $1 - 1/\text{poly}(n)$. Once the ongoing vertices are indexed, it is easy to assign each of them a block of size $m/(m/\log^c n)/\log^2 n$, which is $\max\{m/n, \log^c n\}/\log^2 n$. By a union bound, the COMPACT takes $O(\log\log_{m/n} n)$ time with good probability. □

[9] [74] actually compacts the k distinguished elements of A to an array of length $(1+\epsilon)k$ using $O(n/\log^* n)$ processors with probability $1 - 1/c^{n^{1/25}}$ for any constants $\epsilon > 0$ and $c > 1$, but Lemma 3.4.2 suffices for our use. The second part of the lemma is a straightforward corollary (cf. §2.4.1 in [74]).

The renamed vertex id is used only for approximate compaction; for all other cases, we still use the original vertex id. Renaming the vertex id to range $[2m/\log^c n]$ not only facilitates the processor allocation at the beginning, but more importantly, guarantees that each subsequent processor allocation takes $O(1)$ time, because the array A to be compacted has length $|A| \leq 2m/\log^c n$ while the number of processors is at least $|A| \log |A|$ when $c \geq 10$ (cf. Lemma 3.4.2, see §3.1.2 and details in §3.4.1).

In the remainder of this section, we present the detailed method EXPAND-MAXLINK in §3.4.1, then we prove that Faster Connected Components algorithm correctly computes the connected components of the input graph (cf. §3.4.2), each round (EXPAND-MAXLINK) can be implemented on an ARBITRARY CRCW PRAM in constant time (cf. §3.4.3), uses $O(m)$ processors over all rounds (cf. §3.4.4), reduces the diameter to $O(1)$ and flatten all trees in $O(\log d + \log\log_{m/n} n)$ rounds (cf. §3.4.5), leading to the proof of Theorem 3.1.3 in §3.4.6.

3.4.1 Algorithmic Framework

In this section, we present the key concepts and algorithmic framework of the EXPAND-MAXLINK.

Level and budget. The *level* $\ell(v)$ of a vertex v is a non-negative integer that can either remain the same or increase by one during a round. By Lemma 3.4.3, at the beginning of round 1, each ongoing vertex v owns a block of size $b_1 := \max\{m/n, \log^c n\}/\log^2 n$, whose level is defined as 1; the level of a non-root vertex whose parent is ongoing is defined as 0 and $b_0 := 0$ for soundness; all other vertices are ignored as their components have been computed. During a round, some roots become non-roots by updating their parents; for those vertex remains to be a root, its level might be increased by one; a root with level ℓ is assigned to a block of size $b_\ell := b_1^{1.01^{\ell-1}}$ at the end of the round. Given $b \geq 0$, a vertex v has *budget* $b(v) := b$ if the maximal-size block owned by v has size b. Each block of size b is partitioned into $\sqrt{b}$ (indexed) tables, each with size $\sqrt{b}$.

Neighbor set. Recall from §3.2.2 that the edges that define the current graph include: (i) the (altered) original edges corresponding to edge processors, and (ii) the (altered) added edges in the tables over all rounds of all vertices; any vertex within distance 1 from v (including v) in the current graph is called a neighbor of v. For any vertex v, let $N(v)$ be the set of its neighbors. In Step (3) we use the old $N(v)$ when initializing the loop that enumerates $N(v)$. For any vertex set S, define $N(S) := \bigcup_{w \in S} N(w)$, and define $S.p := \{w.p \mid w \in S\}$.

Hashing. At the beginning of a round, one random hash function h is chosen, then all neighbor roots of all roots use h to do individual hashing in Step (3). As same as in §3.2, a pairwise independent h suffices, thus each processor only reads two words. The hashing in Step (5) follows the same manner using the same h. For each vertex v, let $H(v)$ be the first table in its block, which will store the added edges incident on v. Step (5) is implemented by storing the old tables for all vertices while hashing new items (copied from the old $H(v)$ and old $H(w)$ in the block of w) into the new table.

EXPAND-MAXLINK:

1. MAXLINK; ALTER.
2. For each root v: increase $\ell(v)$ with probability $b(v)^{-0.06}$.
3. For each root v: for each root $w \in N(v)$: if $b(w) = b(v)$ then hash w into $H(v)$.
4. For each root v: if there is a collision in $H(v)$ then mark v as dormant. For each vertex v: if there is a dormant vertex in $H(v)$ then mark v as dormant.
5. For each root v: for each $w \in H(v)$: for each $u \in H(w)$: hash u into $H(v)$. For each root v: if there is a collision in $H(v)$ then mark v as dormant.
6. MAXLINK; SHORTCUT; ALTER.
7. For each root v: if v is dormant and did not increase level in Step (2) then increase $\ell(v)$.
8. For each root v: assign a block of size $b_{\ell(v)}$ to v.

MAXLINK: repeat {for each vertex v: let $u := \arg\max_{w \in N(v).p} \ell(w)$, if $\ell(u) > \ell(v)$ then update $v.p$ to u} for 2 iterations.

The ALTER and SHORTCUT (cf. Steps (1,6)) are the same as in §3.2. The MAXLINK uses parent links, instead of direct links in §3.2 and §3.3.

3.4.2 Correctness

In this section, we prove that Faster Connected Components algorithm correctly computes the connected components of the input graph when it ends.

First of all, we prove a useful property on levels and roots.

Lemma 3.4.4. *If a vertex v is a non-root at any step, then during the execution after that step, v is a non-root, $\ell(v)$ cannot change, and $0 \leq \ell(v) < \ell(v.p)$.*

Proof. The proof is by an induction on rounds. At the beginning of the first round, by the definition of levels (cf. §3.4.1), each ongoing vertex v has level 1 and $v = v.p$, and each non-root with an ongoing parent has level 0 (vertices in trees rooted at finished vertices are ignored), so the lemma holds.

In Step (1), each iteration of a MAXLINK can only update the parent to a vertex with higher level, which cannot be itself. In Step (2), level increase only applies to roots. The invariant holds after the MAXLINK in Step (6) for the same reasons as above. In SHORTCUT (cf. Step (6)), each vertex v updates its parent to $v.p.p$, which, by the induction hypothesis, must be a vertex with level higher than v if v is a non-root, thus cannot be v. In Step (7), level increase only applies to roots. All other steps and ALTERs do not change the labeled digraph nor levels, giving the lemma. □

Based on this, we can prove the following key result, which implies the correctness of our algorithm.

Lemma 3.4.5. *The following conditions hold at the end of each round:*

1. *For any component in the input graph, its vertices are partitioned into trees in the labeled digraph such that each tree belongs to exactly one component.*
2. *A tree does not contain all the vertices in its component if and only if there is an edge between a vertex in this tree and a vertex in another tree.*

The proof of Lemma 3.4.5 relies on the following invariant:

Lemma 3.4.6. *For any vertices v and w, if w is a neighbor of v in the current graph or $w = v.p$, then v and w are in the same component. If v is incident with an edge, then there exists a path in the current graph between v and $v.p$.*

Proof. By inspecting Vanilla algorithm (called within COMPACT) and monotonicity, Lemma 3.4.6 holds at the beginning of the first round. By an induction on rounds, it suffices to prove the following: (i) the ALTER (cf. Steps (1,6)) and Step (5) preserve the invariant (v and w are in the same component), as only these two change the neighbor sets; (ii) the MAXLINK and SHORTCUT (cf. Steps (1,6)) preserve the invariant, as only these two change the parents.

Each iteration of the MAXLINK preserves the invariant since both v and its neighbor w, and $w, w.p$ are in the same component by the induction hypothesis. So MAXLINK preserves the invariant. To

prove that an ALTER preserves the invariant, let (v', w') be an edge before ALTER such that $v'.p = v$ and $w'.p = w$ (to make w a neighbor of v after an ALTER). By the induction hypothesis, w', w, v', v, and v', w' are in the same component, thus so do v, w. A SHORTCUT preserves the invariant since $v, v.p$ and $v.p, v.p.p$ are both in the same component. Step (5) preserves the invariant, because $H(v)$, $H(w)$ are subsets of the neighbor sets of v and w respectively before Step (5) (cf. Step (3)), thus v, w and w, u are both in the same component by the induction hypothesis, and so do v and its new neighbor u. This finishes the induction and proves the first part of the lemma.

Now we prove the second part of the lemma. By an induction, we assume the invariant holds before some step and prove that each step of EXPAND-MAXLINK preserves the invariant. Before a SHORTCUT, there are paths between $v, v.p$ and $v.p, v.p.p$, thus there is a path between v and $v.p$ after updating $v.p$ to the old $v.p.p$. Before an ALTER, there must exist a child v' of v such that v' is incident with an edge, otherwise v has no incident edge after the ALTER. By the induction hypothesis, there is a path between v' and v, which is altered to be a new path between v and $v.p$ after the ALTER. In each iteration of a MAXLINK, if $v.p$ does not change then the invariant trivially holds; otherwise, let (v, w) be the edge that updates $v.p$ to $w.p$. By the induction hypothesis, there is a path between w and $w.p$, so there is a path between v and $v.p = w.p$ after the iteration. Other steps in the EXPAND-MAXLINK can only add edges, giving the lemma. □

Proof of Lemma 3.4.5. Conditions (1) and (2) hold at the beginning of the first round by Lemma 3.2.2 and monotonicity of Vanilla algorithm. By Lemma 3.4.4, the levels along any tree path from leaf to root are monotonically increasing, so there cannot be a cycle and the labeled digraph must be a collection of trees. By Lemma 3.4.6 and an induction on tree, all vertices of a tree belong to the same component. This proves Condition (1).

For any tree T, if there is an edge between a vertex v in T and another vertex w in another tree T', then by Lemma 3.4.6, v, w belong to the same component and so do vertices in T and T', which means T does not contain all the vertices of this component. On the other hand, assume T does not contain all the vertices in its component, then there must be another tree T' corresponding to the same component. By an induction, we assume the invariant that there is an edge between a vertex v in T and a vertex w in another tree holds, then proves that an ALTER and (each iteration of) a MAXLINK preserve the invariant. An ALTER moves the edge between v and w to their parents in their trees, whose two endpoints must belong to different trees as w not in T, so the invariant holds. Assume a parent update in an iteration of MAXLINK changes the parent of u to a vertex $u.p$ in T'. Let u' be the old parent of u in T. By Lemma 3.4.6, there are paths between u, u' and $u, u.p$,

so there must be a path between u' and $u.p$. Since u' and $u.p$ belong to different trees: the new T and T' respectively, there must be en edge from a vertex in T to another tree, and an edge from a vertex in T' to another tree, giving the lemma. □

Lemma 3.4.7. *If Faster Connected Components algorithm ends, then it correctly computes the connected components of the input graph.*

Proof. When the repeat loop ends, all trees are flat, which means in the last round, all trees are flat before the ALTER in Step (6) since an ALTER does not change the labeled digraph. After the ALTER, all edges are only incident on roots. By Lemma 3.4.5, these trees partition the connected components; moreover, if a root is not the only root in its component then there must be an edge between it and another root in its component, i.e., the preconditions of Lemma 3.2.2 is satisfied. This implies that the following Connected Components algorithm is applicable and must correctly compute the connected components of the input graph by §3.2. □

3.4.3 Implementation

In this section, we show how to implement EXPAND-MAXLINK such that any of the first $O(\log n)$ rounds runs in constant time with good probability.

Lemma 3.4.8. *With good probability, any of the first $O(\log n)$ rounds can be implemented to run in $O(1)$ time.*

Proof. The ALTER (cf. Steps (1,6)) applies to all edges in the current graph. Since each edge corresponds to a distinct processor, Step (3) and the ALTER take $O(1)$ time.

Steps (2,4,7) and SHORTCUT take $O(1)$ time as each vertex has a corresponding processor and a collision can be detected using the same hash function to write to the same location again.

In each of the 2 iterations in MAXLINK, each vertex v updates its parent to a neighbor parent with the highest level if this level is higher than $\ell(v)$. Since a vertex can increase its level by at most 1 in any round (cf. Steps (2,7)), there are $O(\log n)$ different levels. Let each neighbor of v write its parent with level ℓ to the ℓ-th cell of an array of length $O(\log n)$ in the block of v (arbitrary win). Furthermore, by the definitions of level and budget, the block of any vertex with positive level in any round has size at least $b_1 = \Omega(\log^3 n)$ when $c \geq 10$. (Vertex with level 0 does not have neighbors.) Therefore, we can assign a processor to each pair of the cells in this array such that each non-empty cell can determine whether there is a non-empty cell with larger index then finds the non-empty cell

with largest index in $O(1)$ time, which contains a vertex with the maximum level. As a result, Steps (1,6) take $O(1)$ time.

By Step (3), any $w \in H(v)$ has $b(w) = b(v)$, so each $u \in H(w)$ such that $w \in H(v)$ owns a processor in the block of v since $\sqrt{b(v)} \cdot \sqrt{b(w)} = b(v)$ and any vertex in a table is indexed (by its hash value). Therefore, together with collision detection, Step (5) takes $O(1)$ time.

In Step (8), each vertex is assigned to a block. The pool of $\Theta(m)$ processors is partitioned into $\Theta(\log^2 n)$ zones such that the processor allocation in round r for vertices with level ℓ uses the zone indexed by (r, ℓ), where $r, \ell \in O(\log n)$ in the first $O(\log n)$ rounds. Since there are $\Theta(m)$ processors in total and all the vertex ids are in $[2m/\log^c n]$ with good probability (cf. Lemma 3.4.3), we can use $\Theta(m/\log n)$ processors for each different level and apply Lemma 3.4.2 to index each root in $O(1)$ time with probability $1 - 1/\text{poly}(n)$ such that the indices of vertices with the same level are distinct, then assign each of them a distinct block in the corresponding zone. Therefore, Step (8) takes $O(1)$ time with good probability by a union bound over all $O(\log n)$ levels and rounds.

Finally, we need to implement the break condition in $O(1)$ time, i.e., to determine whether the graph has diameter $O(1)$ and all trees are flat at the end of each round. The algorithm checks the following two conditions in each round: (i) all vertices do not change their parents nor labels in this round, and (ii) for any vertices v, w, u such that $w \in H(v)$, $u \in H(w)$ before Step (5), the $h(u)$-th cell in $H(v)$ already contains u. Conditions (i) and (ii) can be checked in $O(1)$ time by writing a flag to vertex processor v if they do not hold for v, then let each vertex with a flag write the flag to a fixed processor. If there is no such flag then both Conditions (i) and (ii) hold and the loop breaks. If there is a non-flat tree, some parent must change in the SHORTCUT in Step (6). If all trees are flat, they must be flat before the ALTER in Step (6), then an ALTER moves all edges to the roots. Therefore, if Condition (i) holds, all trees are flat and edges are only incident on roots. Moreover, no level changing means no vertex increase its level in Step (2) and there is no dormant vertex in Step (7). So for each root v, $N(v) = H(v)$ after Step (3) and $N(N(v)) = H(v)$ after Step (5) as there is no collision. By Condition (ii), the table $H(v)$ does not change during Step (5), so $N(v) = N(N(v))$. If there exists root v such that there is another root with distance at least 2 from v, then there must exist a vertex $w \neq v$ at distance exactly 2 from v, so $w \notin N(v)$ and $w \in N(N(v))$, contradicting with $N(v) = N(N(v))$. Therefore, any root is within distance at most 1 from all other roots in its component and the graph has diameter $O(1)$.

Since each of them runs in $O(1)$ time with good probability, the lemma follows. □

3.4.4 Number of Processors

In this section, we show that with good probability, the first $O(\log n)$ rounds use $O(m)$ processors in total.

First of all, observe that in the case that a root v with level ℓ increases its level in Step (2) but becomes a non-root at the end of the round, v is not assigned a block of size $b_{\ell(v)}$ in Step (8). Instead, v owns a block of size $b_\ell = b_{\ell(v)-1}$ from the previous round. Since in later rounds a non-root never participates in obtaining more neighbors by maintaining its table in Steps (3-5) (which is the only place that requires a larger block), such *flexibility* in the relationship between level and budget is acceptable.

By the fact that any root v at the end of any round owns a block of size $b_{\ell(v)} = b_1^{1.01^{\ell(v)-1}}$, a non-root can no longer change its level nor budget (cf. Lemma 3.4.4), and the discussion above, we obtain:

Corollary 3.4.9. *Any vertex v owns a block of size b at the end of any round where $b_{\ell(v)-1} = b_1^{1.01^{\ell(v)-2}} \le b \le b_1^{1.01^{\ell(v)-1}} = b_{\ell(v)}$; if v is a root, then the upper bound on b is tight.*

Secondly, we prove two simple facts related to MAXLINK.

Lemma 3.4.10. *For any vertex v with parent v' and any $w \in N(v)$ before an iteration of* MAXLINK*, it must be $\ell(w.p) \ge \ell(v')$ after the iteration; furthermore, if $\ell(w.p) > \ell(v)$ before an iteration, then v must be a non-root after the iteration.*

Proof. For any $w \in N(v)$, its parent has level $\max_{u \in N(w)} \ell(u.p) \ge \ell(v')$ after an iteration of MAXLINK. This implies that if $\ell(w.p) > \ell(v)$ before an iteration, then after that $\ell(v.p)$ is at least the level of the old parent of w which is strictly higher than $\ell(v)$, so v must be a non-root. □

Lemma 3.4.11. *For any root v with budget b at the beginning of any round, if there is a root $w \in N(v)$ with at least $b^{0.1}$ neighbor roots with budget b after Step (1), then v either increases level in Step (2) or is a non-root at the end of the round with probability $1 - b^{-0.06}$.*

Proof. Assume v does not increase level in Step (2). Let w be any root in $N(v)$ after Step (1). Since each root $u \in N(w)$ with budget b (thus level at least $\ell(v)$ by Corollary 3.4.9) increases its level with probability $b^{-0.06}$ independently, with probability at least

$$1 - (1 - b^{-0.06})^{b^{0.1}} \ge 1 - \exp(-b^{0.04}) \ge 1 - n^{-b^{0.03}} \ge 1 - b^{-0.06},$$

at least one u increases level to at least $\ell(v)+1$ in Step (2), where in the second inequality we have used the fact that $b^{0.01} \geq b_1^{0.01} \geq \log n$ because $b_1 = \max\{m/n, \log^c n\}/\log^2 n$ and $c \geq 200$. Since $u \in N(N(v))$ after Step (1), by Lemma 3.4.10, there is a $w' \in N(v)$ such that $\ell(w'.p) \geq \ell(v)+1$ after the first iteration of MAXLINK in Step (6). Again by Lemma 3.4.10, this implies that v cannot be a root after the second iteration and the following SHORTCUT. □

Using the above result, we can prove the following key lemma, leading to the total number of processors.

Lemma 3.4.12. *For any root v with budget b at the beginning of any round, $b(v)$ is increased to $b^{1.01}$ in this round with probability at most $b^{-0.05}$.*

Proof. In Step (2), $\ell(v)$ increases with probability $b^{-0.06}$. If $\ell(v)$ does increase here then it cannot increase again in Step (7), so we assume this is not the case (and apply a union bound at the end).

If there is a root $w \in N(v)$ with at least $b^{0.1}$ neighbor roots with budget b after Step (1), then v is a root at the end of the round with probability at most $b^{-0.06}$ by the assumption and Lemma 3.4.11. So we assume this is not the case.

By the previous assumption we know that at most $b^{0.1}$ vertices are hashed into $H(v)$ in Step (3). By pairwise independency, with probability at most $(b^{0.1})^2/\sqrt{b} = b^{-0.3}$ there is a collision as the table has size $\sqrt{b}$, which will increase $\ell(v)$ (cf. Steps (4,7)).

Now we assume that there is no collision in $H(v)$ in Step (3), which means $H(v)$ contains all the at most $b^{0.1}$ neighbor roots with budget b. By the same assumption, each such neighbor root w has at most $b^{0.1}$ neighbor roots with budget b, so there is a collision in $H(w)$ in Step (3) with probability at most $(b^{0.1})^2/\sqrt{b} = b^{-0.3}$. By a union bound over all the $|H(v)| \leq b^{0.1}$ such vertices, v is marked as dormant in the second statement of Step (4) (and will increase level in Step (7)) with probability $b^{-0.2}$.

It remains to assume that there is no collision in $H(v)$ nor in any $H(w)$ such that $w \in H(v)$ after Step (4). As each such table contains at most $b^{0.1}$ vertices, in Step (5) there are at most $b^{0.2}$ vertices to be hashed, resulting in a collision in $H(v)$ with probability at most $(b^{0.2})^2/\sqrt{b} = b^{-0.1}$, which increases $\ell(v)$ in Step (7).

Observe that only a root v at the end of the round can increase its budget, and the increased budget must be $b^{1.01}$ since the level can increase by at most 1 during the round and $b = b_{\ell(v)}$ at the beginning of the round by Corollary 3.4.9. By a union bound over the events in each paragraph, $b(v)$ is increased to $b^{1.01}$ with probability at most $b^{-0.06} + b^{-0.06} + b^{-0.3} + b^{-0.2} + b^{-0.1} \leq b^{-0.05}$. □

Finally, we are ready to prove an upper bound on the number of processors.

Lemma 3.4.13. *With good probability, the first $O(\log n)$ rounds use $O(m)$ processors in total.*

Proof. The number of processors for (altered) original edges and vertices are clearly $O(m)$ over all rounds (where each vertex processor needs $O(1)$ private memory to store the corresponding parent, (renamed) vertex id, the two words for pairwise independent hash function, level, and budget). By the proof of Lemma 3.4.8, with good probability the processor allocation in Step (8) always succeeds and has a multiplicative factor of 2 in the number of processors allocated before mapping the indexed vertices to blocks (cf. Definition 3.4.1). Therefore, we only need to bound the number of processors in blocks that are assigned to a vertex in Step (8) in all $O(\log n)$ rounds.

For any positive integer ℓ, let n_ℓ be the number of vertices that ever reach budget b_ℓ during the first $O(\log n)$ rounds. For any vertex v that ever reaches budget b_ℓ, it has exactly one chance to reach budget $b_{\ell+1}$ in a round if v is a root in that round, which happens with probability at most ${b_\ell}^{-0.05}$ by Lemma 3.4.12. By a union bound over all $O(\log n)$ rounds, v reaches budget $b_{\ell+1}$ with probability at most $O(\log n) \cdot {b_\ell}^{-0.05} \leq {b_\ell}^{-0.04}$ when $c \geq 200$, since $b_\ell \geq b_1 \geq \log^{c-2} n$. We obtain $\mathbb{E}[n_{\ell+1} \mid n_\ell] \leq n_\ell \cdot {b_\ell}^{-0.04}$, thus by $b_{\ell+1} = {b_\ell}^{1.01}$, it must be:

$$\mathbb{E}[n_{\ell+1} b_{\ell+1} \mid n_\ell] \leq n_\ell \cdot {b_\ell}^{-0.04} \cdot {b_\ell}^{1.01} = n_\ell b_\ell \cdot {b_\ell}^{-0.03}.$$

By Markov's inequality, $n_{\ell+1} b_{\ell+1} \leq n_\ell b_\ell$ with probability at least $1 - {b_\ell}^{-0.03} \geq 1 - {b_1}^{-0.03}$. By a union bound on all $\ell \in O(\log n)$, $n_\ell b_\ell \leq n_1 b_1$ for all $\ell \in O(\log n)$ with probability at least $1 - O(\log n) \cdot {b_1}^{-0.03} \geq 1 - {b_1}^{-0.01}$, which is $1 - 1/\text{poly}((m \log n)/n)$ by $b_1 = \max\{m/n, \log^c n\}/\log^2 n$ and $c \geq 100$. So the number of new allocated processors for vertices with any budget in any of the first $O(\log n)$ rounds is at most $n_1 b_1$ with good probability.

Recall from Lemma 3.4.3 that with good probability,

$$\begin{aligned} n_1 \cdot b_1 &= \min\{n, 2m/\log^c n\} \cdot \max\{m/n, \log^c n\}/\log^2 n \\ &= O(m/\log^2 n). \end{aligned}$$

Finally, by a union bound over all the $O(\log n)$ levels and $O(\log n)$ rounds, with good probability the total number of processors is $O(m)$. □

3.4.5 Diameter Reduction

Let $R \coloneqq O(\log d + \log\log_{m/n} n)$ where the constant hidden in $O(\cdot)$ will be determined later in this section. The goal is to prove that $O(R)$ rounds of EXPAND-MAXLINK suffice to reduce the diameter of the graph to $O(1)$ and flatten all trees with good probability.

In a high level, our algorithm/proof is divided into the following 3 stages/lemmas:

Lemma 3.4.14. *With good probability, after round R, the diameter of the graph is $O(R)$.*

Lemma 3.4.15. *With good probability, after round $O(R)$, the diameter of the graph is at most 1.*

Lemma 3.4.16. *With good probability, after round $O(R)$, the diameter of the graph is $O(1)$ and all trees are flat.*

Lemma 3.4.16 implies that the repeat loop must end in $O(R)$ rounds with good probability by the break condition.

Path Construction

To formalize and quantify the effect of reducing the diameter, consider any shortest path P in the original input graph G. When (possibly) running Vanilla algorithm on G in COMPACT, an ALTER replaces each vertex on P by its parent, resulting in a path P' of the same length as P. (Note that P' might not be a shortest path in the current graph and can contain loops.) Each subsequent ALTER in COMPACT continues to replace the vertices on the old path by their parents to get a new path. Define P_1 to be the path obtained from P by the above process at the beginning of round 1, by $|P| \leq d$ we immediately get:

Corollary 3.4.17. *For any P_1 corresponding to a shortest path in G, $|P_1| \leq d$.*

In the following, we shall fix a shortest path P in the original graph and consider its corresponding paths over rounds. Each ALTER (cf. Steps (1,6)) in each round does such replacements to the old paths, but we also add edges to the graph for reducing the diameter of the current graph: for any vertices v and w on path P', if the current graph contains edge (v, w), then all vertices exclusively between v and w can be removed from P', which still results in a valid path in the current graph from the first to the last vertex of P', reducing the length. If all such paths reduce their lengths to at most d', the diameter of the current graph is at most d'. Formally, we have the following inductive construction of paths for diameter reduction:

Definition 3.4.18 (path construction)**.** *Let all vertices on P_1 be* active*. For any positive integer r, given path P_r with at least 4 active vertices at the beginning of round r,* EXPAND-MAXLINK *constructs P_{r+1} by the following 7 phases:*

1. *The* ALTER *in Step (1) replaces each vertex v on P_r by $v' := v.p$ to get path $P_{r,1}$. For any v' on $P_{r,1}$, let $\underline{v'}$ be on P_r such that $\underline{v'}.p = v'$.*[10]
2. *Let the subpath containing all active vertices on $P_{r,1}$ be $P_{r,2}$.*
3. *After Step (5), set i as 1, and repeat the following until $i \geq |P_{r,2}| - 1$: let $v' := P_{r,2}(i)$, if $\underline{v'}$ is a root and does not increase level during round r then: if the current graph contains edge $(v', P_{r,2}(i+2))$ then mark $P_{r,2}(i+1)$ as* skipped *and set i as $i+2$; else set i as $i+1$.*
4. *For each $j \in [i+1, |P_{r,2}|+1]$, mark $P_{r,2}(j)$ as* passive*.*
5. *Remove all skipped and passive vertices from $P_{r,2}$ to get path $P_{r,5}$.*
6. *Concatenate $P_{r,5}$ with all passive vertices on $P_{r,1}$ and $P_{r,2}$ to get path $P_{r,6}$.*
7. *The* ALTER *in Step (6) replaces each vertex v on $P_{r,6}$ by $v.p$ to get path P_{r+1}.*

For any vertex v on P_r that is replaced by v' in Phase (1), if v' is not skipped in Phase (3), then let $\overline{v}$ be the vertex replacing v' in Phase (7), and call $\overline{v}$ the corresponding vertex *of v in round $r+1$.*

Lemma 3.4.19. *For any non-negative integer r, the P_{r+1} constructed in Definition 3.4.18 is a valid path in the graph and all passive vertices are consecutive from the successor of the last active vertex to the end of P_{r+1}.*

Proof. The proof is by an induction on r. Initially, P_1 is a valid path by our discussion on ALTER at the beginning of this section: it only replaces edges by new edges in the altered graph; moreover, the second part of the lemma is trivially true as all vertices are active. Assuming P_r is a valid path and all passive vertices are consecutive from the successor of the last active vertex to the end of the path. We show the inductive step by proving the invariant after each of the 7 phases in Definition 3.4.18. Phase (1) maintains the invariant. In Phase (2), $P_{r,2}$ is a valid path as all active vertices are consecutive at the beginning of $P_{r,1}$ (induction hypothesis). In Phase (3), if a vertex v is skipped, then there is an edge between its predecessor and successor on the path; otherwise there is an edge between v and its successor by the induction hypothesis; all passive vertices are consecutive

[10] Semantically, there might be more than one $\underline{v'}$ with the same parent v', but our reference always comes with an index on the replaced path, whose corresponding position in the old path is unique. All subsequent names follow this manner.

from the successor of the last non-skipped vertex to the end of $P_{r,2}$ (cf. Phase (4)), so the invariant holds. In Phase (6), since the first passive vertex on $P_{r,2}$ is a successor of the last vertex on $P_{r,5}$ and the last passive vertex on $P_{r,2}$ is a predecessor of the first passive vertex on $P_{r,1}$ (induction hypothesis), the invariant holds. Phase (7) maintains the invariant. Therefore, P_{r+1} is a valid path and all passive vertices are consecutive from the successor of the last active vertex to the end of P_{r+1}. □

Now we relate the path construction to the diameter of the graph:

Lemma 3.4.20. *For any positive integer r, the diameter of the graph at the end of round r is $O(\max_{P_r} |P_{r,2}| + r)$.*

Proof. Any shortest path P in the original input graph from s to t is transformed to the corresponding P_1 at the beginning of round 1, which maintains connectivity between the corresponding vertices of s and t on P_1 respectively. By an induction on the number of ALTERs and Lemma 3.4.19, the corresponding vertices of s and t are still connected by path P_{r+1} at the end of round r. Note that by Lemma 3.4.19, P_{r+1} can be partitioned into two parts after Phase (2): subpath $P_{r,2}$ and the subpath containing only passive vertices. Since in each round we mark at most 2 new passive vertices (cf. Phases (3,4)), we get $|P_{r+1}| \leq |P_{r,5}| + 2r \leq |P_{r,2}| + 2r$. If any path P_{r+1} that corresponds to a shortest path in the original graph have length at most d', the graph at the end of round r must have diameter at most d', so the lemma follows. □

It remains to bound the length of any $P_{r,2}$ in any round r, which relies on the following potential function:

Definition 3.4.21. *For any vertex v on P_1, define its* potential $\phi_1(v) \coloneqq 1$. *For any positive integer r, given path P_r with at least 4 active vertices at the beginning of round r and the potentials of vertices on P_r, define the* potential *of each vertex on P_{r+1} based on Definition 3.4.18 as follows:*[11]

- *For each v replaced by $v.p$ in Phase (1), $\phi_{r,1}(v.p) \coloneqq \phi_r(v)$.*
- *After Phase (4), for each active vertex v on $P_{r,2}$, if the successor w of v is skipped or passive, then $\phi_{r,4}(v) \coloneqq \phi_{r,1}(v) + \phi_{r,1}(w)$.*
- *After Phase (6), for each vertex v on $P_{r,6}$, if v is active on $P_{r,2}$, then $\phi_{r,6}(v) \coloneqq \phi_{r,4}(v)$, otherwise $\phi_{r,6}(v) \coloneqq \phi_{r,1}(v)$.*

[11] The second subscript of a potential indicates the phase to obtain that potential; the subscript for paths in Definition 3.4.18 follows the same manner.

- *For each v replaced by $v.p$ in Phase (7), $\phi_{r+1}(v.p) \coloneqq \phi_{r,6}(v)$.*

We conclude this section by some useful properties of potentials.

Lemma 3.4.22. *For any path P_r at the beginning of round $r \geq 1$, the following holds: (i) $\sum_{v \in P_r} \phi_r(v) \leq d + 1$; (ii) for any v on P_r, $\phi_r(v) \geq 1$; (iii) for any non-skipped v on $P_{r,2}$ and its corresponding vertex $\overline{v}$ on P_{r+1}, $\phi_{r+1}(\overline{v}) \geq \phi_r(v)$.*

Proof. The proof is by an induction on r. The base case follows from $\phi(v) = 1$ for each v on P_1 (cf. Definition 3.4.21) and Corollary 3.4.17. For the inductive step, note that by Definition 3.4.21, the potential of a corresponding vertex is at least the potential of the corresponding vertex in the previous round (and can be larger in the case that its successor is skipped or passive). This gives (ii) and (iii) of the lemma. For any vertex u on $P_{r,2}$, if both u and its successor are active, then $\phi_r(u)$ is presented for exactly 1 time in $\sum_{v \in P_r} \phi_r(v)$ and $\sum_{v \in P_{r+1}} \phi_{r+1}(v)$ respectively; if u is active but its successor w is skipped or passive, then $\phi_r(u) + \phi_r(w)$ is presented for exactly 1 time in each summations as well; if u and its predecessor are both passive, then $\phi_r(u)$ is presented only in $\sum_{v \in P_r} \phi_r(v)$; the potential of the last vertex on $P_{r,2}$ might not be presented in $\sum_{v \in P_{r+1}} \phi_{r+1}(v)$ depending on i after Phase (3). Therefore, $\sum_{v \in P_{r+1}} \phi_{r+1}(v) \leq \sum_{v \in P_r} \phi_r(v)$ and the lemma holds. □

Proof of Lemma 3.4.14

Now we prove Lemma 3.4.14, which relies on several results. First of all, we need an upper bound on the maximal possible level:

Lemma 3.4.23. *With good probability, the level of any vertex in any of the first $O(\log n)$ rounds is at most $L \coloneqq 1000 \max\{2, \log\log_{m/n} n\}$.*

Proof. By Lemma 3.4.13, with good probability the total number of processors used in the first $O(\log n)$ rounds is $O(m)$. We shall condition on this happening then assume for contradiction that there is a vertex v with level at least L in some round.

If $\log\log_{m/n} n \leq 2$, then $m/n \geq n^{1/4}$. By Corollary 3.4.9, a block owned by v has size at least

$$b_1{}^{1.01^{2000-2}} \geq b_1{}^{20} \geq (m/n/\log^2 n)^{20} \geq (n^{1/5})^{20} = n^4 \geq m^2,$$

which is a contradiction as the size of this block owned by v exceeds the total number of processors $O(m)$.

Else if $\log \log_{m/n} n > 2$, then by Corollary 3.4.9, a block owned by v has size at least

$$b_1^{1.01^{L-2}} \geq b_1^{1.01^{999 \log \log_{m/n} n}} \geq b_1^{(\log_{m/n} n)^{10}} \geq b_1^{8 \log_{m/n} n}. \tag{3.3}$$

Whether $m/n \leq \log^c n$ or not, if $c \geq 10$, it must be $b_1 = \max\{m/n, \log^c n\}/\log^2 n \geq \sqrt{m/n}$. So the value of (3.3) is at least $n^4 \geq m^2$, contradiction. Therefore, the level of any vertex is at most L. □

We also require the following key lemma:

Lemma 3.4.24. *For any root v and any $u \in N(N(v))$ at the beginning of any round, let u' be the parent of u after Step (1). If v does not increase level and is a root during this round, then $u' \in H(v)$ after Step (5).*

To prove Lemma 3.4.24, we use another crucial property of the algorithm, which is exactly the reason behind the design of MAXLINK.

Lemma 3.4.25. *For any root v and any $u \in N(N(v))$ at the beginning of any round, if v does not increase level in Step (2) and is a root at the end of the round, then $u.p$ is a root with budget $b(v)$ after Step (1).*

Proof. By Lemma 3.4.4, v is a root during this round. For any $w \in N(v)$ and any $u \in N(w)$, applying Lemma 3.4.10 for 2 times, we get that $\ell(v) \leq \ell(w.p)$ and $\ell(v) \leq \ell(u.p)$ after the MAXLINK in Step (1). If there is a $u \in N(N(v))$ such that $u.p$ is a non-root or $\ell(u.p) > \ell(v)$ before the ALTER in Step (1), it must be $\ell(u.p.p) > \ell(v)$ by Lemma 3.4.4. Note that $u.p$ is in $N(N(v))$ after the ALTER, which still holds before Step (6) as we only add edges. By Lemma 3.4.10, there is a $w' \in N(v)$ such that $\ell(w'.p) > \ell(v)$ after the first iteration of MAXLINK in Step (6). Again by Lemma 3.4.10, this implies that v cannot be a root after the second iteration, a contradiction. Therefore, for any $u \in N(N(v))$, $u.p$ is a root with level $\ell(v)$ (thus budget $b(v)$) after Step (1). □

With the help of Lemma 3.4.25 we can prove Lemma 3.4.24:

Proof of Lemma 3.4.24. For any vertex u, let $N'(u)$ be the set of neighbors after Step (1). First of all, we show that after Step (5), $H(v)$ contains all roots in $N'(N'(v))$ with budget b, where b is the budget of v at the beginning of the round. For any root $w \in N'(v)$, in Step (3), all roots with budget $b(w)$ in $N'(w)$ are hashed into $H(w)$. If there is a collision in any $H(w)$, then v must be dormant (cf. Step (4)) thus increases level in either Step (2) or (7), contradiction. So there is no collision in

$H(w)$ for any $w \in N'(v)$, which means $H(w) \supseteq N'(w)$. Recall that $v \in N'(v)$ and we get that all roots with budget $b(w) = b$ from $N'(N'(v))$ are hashed into $H(v)$ in Step (5). Again, if there is a collision, then v must be dormant and increase level in this round. Therefore, $N'(N'(v)) \subseteq H(v)$ at the end of Step (5).

By Lemma 3.4.25, for any $u \in N(N(v))$ at the beginning of any round, $u' = u.p$ is a root with budget b in $N'(N'(v))$ after Step (1). Therefore, $u' \in H(v)$ at the end of Step (5), giving Lemma 3.4.24. □

The proof of Lemma 3.4.14 relies on the following lemma based on potentials:

Lemma 3.4.26. *At the beginning of any round $r \geq 1$, for any active vertex v on any path P_r, $\phi_r(v) \geq 2^{r-\ell(v)}$.*

Proof. The proof is by an induction on r. The base case holds because for any (active) vertex v on P_1, $\phi_1(r) = 1$ and $r = \ell(v) = 1$ (other vertices do not have incident edges). Now we prove the inductive step from r to $r+1$ given that the corresponding vertex $\overline{v}$ of $v \in P_r$ is on P_{r+1} and active.

Suppose v is a non-root at the end of round r. If v is a non-root at the end of Step (1), then $\ell(v.p) > \ell(v)$ after Step (1) by Lemma 3.4.4, and $\ell(\overline{v}) \geq \ell(v.p) > \ell(v)$; else if v first becomes a non-root in Step (6), then $\overline{v} = v.p$ and $\ell(v.p) > \ell(v)$ after Step (6) by Lemma 3.4.4. So by the induction hypothesis, $\phi_{r+1}(\overline{v}) \geq \phi_r(v) \geq 2^{r-\ell(v)} \geq 2^{r+1-\ell(\overline{v})}$.

Suppose v increases its level in round r. Let ℓ be the level of v at the beginning of round r. If the increase happens in Step (2), then v is a root after Step (1). Whether v changes its parent in Step (6) or not, the level of $\overline{v} = v.p$ is at least $\ell + 1$. Else if the increase happens in Step (7), then v is a root after Step (6). So $\overline{v} = v$ and its level is at least $\ell + 1$ at the end of the round. By the induction hypothesis, $\phi_{r+1}(\overline{v}) \geq \phi_r(v) \geq 2^{r-\ell} \geq 2^{r+1-\ell(\overline{v})}$.

It remains to assume that v is a root and does not increase level during round r. By Lemma 3.4.24, for any $u \in N(N(v))$ at the beginning of round r, the parent u' of u after Step (1) is in $H(v)$ after Step (5). Since v is a root during the round, it remains on $P_{r,2}$ after Phase (2). We discuss two cases depending on whether v is at position before $|P_{r,2}| - 1$ or not.

In Phase (3), note that if $v = P_{r,2}(i)$ where $i < |P_{r,2}| - 1$, then $P_{r,2}(i+2)$ is the parent of a vertex in $N(N(v))$ after Step (1), which must be in $H(v)$ after Step (5). Therefore, the graph contains edge $(v, P_{r,2}(i+2))$ and $v' \coloneqq P_{r,2}(i+1)$ is skipped, thus $\phi_{r,4}(v) = \phi_{r,1}(v) + \phi_{r,1}(v')$ by Definition 3.4.21. Since $i + 1 \leq |P_{r,2}| + 1$, v' is an active vertex on $P_{r,1}$. By the induction hypothesis, $\phi_{r,1}(v') = \phi_r(\underline{v'}) \geq 2^{r-\ell(\underline{v'})}$ (recall that $\underline{v'}$ is replaced by its parent v' in Phase (1)/Step (1)). If $\ell(\underline{v'}) > \ell(v)$,

then applying Lemma 3.4.10 for two times we get that v is a non-root after Step (1), a contraction. Therefore, $\phi_{r,1}(v') \geq 2^{r-\ell(\underline{v'})} \geq 2^{r-\ell(v)}$ and $\phi_{r,4}(v) \geq \phi_{r,1}(v)+\phi_{r,1}(v') \geq \phi_r(v)+2^{r-\ell(v)} \geq 2^{r+1-\ell(v)}$.

On the other hand, if $i \geq |P_{r,2}| - 1$ is reached after Phase (3), it must be $i < |P_{r,2}| + 1$ by the break condition of the loop in Phase (3). Note that $v' \coloneqq P_{r,2}(i+1)$ is marked as passive in Phase (4), and by Definition 3.4.21, $\phi_{r,4} = \phi_{r,1}(v) + \phi_{r,1}(v')$. Moreover, since $i + 1 \leq |P_{r,2}| + 1$, v' is an active vertex on $P_{r,1}$. Using the same argument in the previous paragraph, we obtain $\phi_{r,4}(v) \geq 2^{r+1-\ell(v)}$.

By Definition 3.4.21, after Phase (7), $\phi_{r+1}(\overline{v}) = \phi_{r,6}(v) = \phi_{r,4}(v) \geq 2^{r+1-\ell(v)} = 2^{r+1-\ell(\overline{v})}$. As a result, the lemma holds for any active vertex $\overline{v}$ on P_{r+1}, finishing the induction and giving the lemma. □

Proof of Lemma 3.4.14. Let $R \coloneqq \log d + L$, where L is defined in Lemma 3.4.23. By Lemma 3.4.23, with good probability, $\ell(v) \leq L$ for any vertex v in any of the first $O(\log n)$ rounds, and we shall condition on this happening. By Lemma 3.4.26, at the beginning of round R, if there is a path P_R of at least 4 active vertices, then for any of these vertices v, it must be $\phi_R(v) \geq 2^{R-\ell(v)} \geq 2^{R-L} \geq d$. So $\sum_{v \in P_R} \phi_r(v) \geq 4d > d + 1$, contradicting with Lemma 3.4.22. Thus, any path P_R has at most 3 active vertices, which means $|P_{R,2}| \leq 3$ by Definition 3.4.18. Therefore, by Lemma 3.4.20, the diameter of the graph at the end of round R is $O(R)$ with good probability. □

Proof of Lemma 3.4.16

Now we prove Lemma 3.4.16 as outlined at the beginning of §3.4.5. Based on the graph and any P_R at the beginning of round $R + 1$, we need a (much simpler) path construction:

Definition 3.4.27. *For any integer $r > R$, given path P_r with $|P_r| \geq 3$ at the beginning of round r,* EXPAND-MAXLINK *constructs P_{r+1} by the following:*

1. *The* ALTER *in Step (1) replaces each vertex v on P_r by $v' \coloneqq v.p$ to get path $P_{r,1}$. For any v' on $P_{r,1}$, let $\underline{v'}$ be on P_r such that $\underline{v'}.p = v'$.*

2. *After Step (5), let $v' \coloneqq P_{r,1}(1)$, if $\underline{v'}$ is a root at the end of round r and does not increase level during round r then: if the current graph contains edge $(v', P_{r,1}(3))$ then remove $P_{r,1}(2)$ to get path $P_{r,2}$.*

3. *The* ALTER *in Step (6) replaces each vertex v on $P_{r,2}$ by $v.p$ to get path P_{r+1}.*

For any vertex v on P_r that is replaced by v' in the first step, if v' is not removed in the second step, then let $\overline{v}$ be the vertex replacing v' in the third step, and call $\overline{v}$ the corresponding vertex *of v in round $r + 1$.*

An analog of Lemma 3.4.19 immediately shows that P_r is a valid path for any $r \geq R+1$. The proof of Lemma 3.4.15 is simple enough without potential:

Proof of Lemma 3.4.15. By Lemma 3.4.14, at the beginning of round $R+1$, with good probability, any P_{R+1} has length $O(R)$. We shall condition on this happening and apply a union bound at the end of the proof. In any round $r > R$, for any path P_r with $|P_r| \geq 3$, consider the first vertex v' on $P_{r,1}$ (cf. Definition 3.4.27). If $\underline{v'}$ is a non-root or increases its level during round r, then by the first 3 paragraphs in the proof of Lemma 3.4.26, it must be $\ell(\overline{v'}) \geq \ell(\underline{v'}) + 1$. Otherwise, by Lemma 3.4.24, there is an edge between v' and the successor of its successor in the graph after Step (5), which means the successor of v' on $P_{r,1}$ is removed in the second step of Definition 3.4.27. Therefore, the number of vertices on P_{r+1} is one less than P_r if $\ell(\overline{v'}) = \ell(\underline{v'})$ as the level of a corresponding vertex cannot be lower. By Lemma 3.4.23, with good probability, the level of any vertex in any of the $O(\log n)$ rounds cannot be higher than L. As P_{R+1} has $O(R)$ vertices, in round $r = O(R) + L + R = O(R) \leq O(\log n)$, the number of vertices on any P_r is at most 2. Therefore, the diameter of the graph after $O(R)$ rounds is at most 1 with good probability. □

Proof of Lemma 3.4.16. Now we show that after the diameter reaches 1, if the loop has not ended, then the loop must break in $2L + \log_{5/4} L$ rounds with good probability, i.e., the graph has diameter $O(1)$ and all trees are flat.

For any component, let u be a vertex in it with the maximal level and consider any (labeled) tree of this component. For any vertex v in this tree that is incident with an edge, since the diameter is at most 1, v must have an edge with u, which must be a root. So v updates its parent to a root with the maximal level after a MAXLINK, then any root must have the maximal level in its component since a root with a non-maximal level before the MAXLINK must have an edge to another tree (cf. Lemma 3.4.5). Moreover, if v is a root, this can increase the maximal height among all trees in its component by 1.

Consider the tree with maximal height ξ in the labeled digraph after Step (1). By Lemmas 3.4.4 and 3.4.23, with good probability $\xi \leq L$. The maximal level can increase by at most 1 in this round. If it is increased in Step (7), the maximal height is at most $\lceil \xi/2 \rceil + 1$ after the MAXLINK in the next round; otherwise, the maximal height is at most $\lceil (\xi+1)/2 \rceil \leq \lceil \xi/2 \rceil + 1$. If $\xi \geq 4$, then the maximal height of any tree after Step (1) in the next round is at most $(4/5)\xi$ (the worst case is that a tree with height 5 gets shortcutted to height 3 in Step (6) and increases its height by 1 in the MAXLINK of Step (1) in the next round). Therefore, after $\log_{5/4} L$ rounds, the maximal height of any tree is at most 3.

Beyond this point, if any tree has height 1 after Step (1), then it must have height 1 at the end of the previous round since there is no incident edge on leaves after the ALTER in the previous round, thus the loop must have been ended by the break condition. Therefore, the maximal-height tree (with height 3 or 2) cannot increase its height beyond this point. Suppose there is a tree with height 3, then if the maximal level of vertices in this component does not change during the round, this tree cannot increase its height in the MAXLINK of Step (6) nor that of Step (1) in the next round, which means it has height at most 2 as we do a SHORTCUT in Step (6). So after L rounds, all trees have heights at most 2 after Step (1). After that, similarly, if the maximal level does not increase, all trees must be flat. Therefore, after additional L rounds, all trees are flat after Step (1). By the same argument, the loop must have ended in the previous round. The lemmas follows immediately from $L = O(R)$ and Lemma 3.4.15. □

3.4.6 Proof of Theorem 3.1.3

With all pieces from §3.4.2-§3.4.5 and Theorem 3.1.1, we are ready to prove Theorem 3.1.3.

Proof of Theorem 3.1.3. By Lemma 3.4.16 and the break condition, the repeat loop in Faster Connected Components algorithm runs for $O(\log d + \log\log_{m/n} n) \leq O(\log n)$ rounds. So by Lemma 3.4.8 and a union bound, this loop runs in $O(\log d + \log\log_{m/n} n)$ time with good probability. Moreover, since the diameter is $O(1)$ after the loop (cf. Lemma 3.4.16), the following Connected Component algorithm runs in $O(\log\log_{m/n} n)$ time with good probability (cf. Theorem 3.1.1). By Lemma 3.4.3, the method COMPACT also runs in $O(\log\log_{m/n} n)$ time with good probability. Therefore, by a union bound, the total running time is $O(\log d + \log\log_{m/n} n)$ with good probability. When it ends, by Lemma 3.4.7, the algorithm correctly computes the connected components of the input graph. Since there are at most $O(\log n)$ rounds in the loop, by Lemma 3.4.3, Lemma 3.4.13, and Theorem 3.1.1, the total number of processors is $O(m)$ with good probability by a union bound. Theorem 3.1.3 follows from a union bound at last. □

Part II

Streaming Algorithms for Graph Matching

Chapter 4

Simple Algorithm for Approximate Bipartite Matching

4.1 Background

An interesting class of algorithms for bipartite matching are *auction algorithms* that interpret the problem as finding a welfare-maximizing allocation of items to bidders in an auction. These algorithms have their root in the work of [51] and have been extensively studied from both economic and algorithmic perspectives (see, e.g., [141, 32, 33]).

The standard auction algorithm treats the vertices on the two sides of the bipartition as *bidders* and *items*, respectively. The "auction" then starts with every bidder as *unallocated* and every item with *price* 0. In each iteration of the auction, an arbitrary unallocated bidder is selected to report an item with minimum price among the items he desires (i.e., are in the neighborhood of the corresponding vertex in the graph). The price of this item is then increased by ϵ and it will be reallocated to this bidder. This process is continued while keeping the price of items capped at 1 (i.e., the items with price 1 are no longer reallocated). It is well known that this auction terminates in $O(n/\epsilon)$ iterations and the resulting allocation of items to bidders gives a $(1-\epsilon)$-approximate matching of the original graph. It is also easy to construct examples in which $\Omega(n/\epsilon)$ iterations are needed for this auction to converge to its final allocation.

Our Results. We present a simple variant of the auction algorithm with much faster convergence guarantees by replacing the reallocation rule in each iteration.

Suppose in every iteration of the auction algorithm, we pick a maximal matching in the subgraph consisting of the unallocated bidders and all their minimum-price items; then, the auction terminates in a $(1-\epsilon)$-approximate matching of the input graph in only $O(1/\epsilon^2)$ iterations.

While multiple variants of the auction algorithm have been considered previously (and many of them are listed in [32, 33, 34]), we are not aware of any prior auction algorithm with similar guarantees as our main result. The closest result to ours is that of [55] which proves that if in every iteration, every unallocated bidder reports the index of a *uniformly at random* minimum-price item (as opposed to an arbitrary one), and the items are reallocated arbitrarily among the bidders that reported them, then the entire auction converges in $O(\log n/\epsilon^2)$ iterations. Interestingly, the result of [55] can be seen as *almost* a special case of our auction algorithm: one way of computing a maximal matching in each iteration of our algorithm is to sample one minimum-price item per each unallocated bidder and append the sampled edges greedily to a maintained matching; repeating this process for $O(\log n)$ steps then will gives us a maximal matching[1] (see, e.g. [121, 111]). As such, each of the $O(1/\epsilon^2)$ iterations of our auction algorithm can be implemented in $O(\log n)$ rounds of sampling one item per unallocated bidder, leading to a $(1-\epsilon)$-approximation in $O(\log n/\epsilon^2)$ rounds of sampling (the only difference between this and the algorithm of [55] is that [55] intertwines the sampling part with increasing prices, while ours does not).

Our auction algorithm gives a simple and "light-weight" way of boosting the $(1/2)$-approximation of maximal matchings to a $(1-\epsilon)$-approximation. "All" one needs to implement this approach is to be able to maintain the prices of items, and run *any* maximal matching algorithm on the subgraph of the input between unallocated bidders and the minimum-price items in their neighborhoods. In the following, we give two important applications of this approach; we believe this technique can be useful in many other settings as well.

Semi-streaming algorithms. In the graph streaming model, the edges of the input n-vertex graph $G = (V, E)$ are presented in a stream (in an arbitrary/adversarial order). A *semi-streaming* algorithm is allowed to make one or a few passes over the stream, use a limited memory of $\widetilde{O}(n)$ (measured in machine words of $\Theta(\log n)$ bits), and at the end output the answer to the problem at hand, say, find an approximate maximum matching of G.

Maximum matching is arguably the most studied problem in the graph streaming model; see, e.g. [62, 92, 73, 107, 90, 106, 14, 66, 60, 31, 125, 2, 58, 156, 4, 6, 94, 96, 18, 19] and references therein. The simple greedy algorithm that maintains a maximal matching of the input graph achieves a 1/2-

[1] This is basically Luby's celebrated distributed/parallel maximal independent set algorithm [121] applied to the line graph to obtain a maximal matching instead.

approximation in $O(n)$ space. Beating this factor is a longstanding open problem in the area but it is known that any approximation better than $(1-1/e)$ is not possible via a semi-streaming algorithm [92] (see also [73]). The state-of-the-art multi-pass algorithms can achieve a $(1-\epsilon)$-approximation in $O(\frac{1}{\epsilon^2} \cdot \log\log(1/\epsilon))$ passes for bipartite graphs [4] and in $(1/\epsilon)^{O(1/\epsilon)}$ passes for general graphs [125] (there is also an $O(\log n/\epsilon)$ pass algorithm [6] with better dependence on ϵ but worse on n).

Our new auction algorithm implies the following theorem.

Theorem 4.1.1. *There is a deterministic semi-streaming algorithm for bipartite matching that for any $\epsilon > 0$ (not necessarily a constant) gives a $(1-\epsilon)$-approximation in $O(1/\epsilon^2)$ passes and $O(n)$ space.*

This algorithm is obtained by running our auction algorithm (which requires storing prices in $O(n)$ space) and using the straightforward greedy algorithm for maximal matching (in $O(n)$ space and one pass) in each iteration of the auction. The previous best algorithm for bipartite matching was that of [4] that uses $O(n/\epsilon)$ space and $O(\frac{1}{\epsilon^2} \cdot \log\log(1/\epsilon))$ passes and was based on multiplicative weight update method. Thus, our algorithm considerably simplifies and improves both the pass complexity (slightly) and, more importantly, the space complexity of the algorithm - the latter is now clearly optimal as $\Omega(n)$ space is needed just to maintain the output matching.

Furthermore, the fact that the space of our algorithm has no dependence on ϵ implies that it can be used to recover very accurate approximation of maximum matching still in the semi-streaming space albeit at a cost of a large pass complexity: in particular, this gives an algorithm that can find a matching that is short of being optimal by only $n^{1/2+\delta}$ edges for any constant $\delta > 0$ in $O(n^{1-2\delta})$ passes. We are not aware of any prior streaming algorithm with such a guarantee and hope that this will also help in designing a semi-streaming $n^{1-\Omega(1)}$ pass algorithm for *exact* bipartite matching.

Massively parallel computation (MPC) algorithms. The MPC model was introduced first in [100] as an abstraction of MapReduce-style computation and has since been refined in a series of papers [76, 26, 9]. In this model, there are p machines each with a memory of size s such that $p \cdot s = O(m)$, where m is the number of edges in the input, namely, the size of the input. The computation proceeds in synchronous rounds: in each round, each machine performs some local computation and at the end of the round they exchange messages. All messages sent and received by each machine in each round have to fit into the local memory of the machine, and hence their length is bounded by s in each round. At the end, the machines collectively output the solution.

In this dissertation, we work both in the *linear memory* regime wherein the per-machine memory is $s = O(n)$ and in the *strongly sublinear memory* regime where $s = n^{\alpha}$ for some constant $\alpha \in (0, 1)$.

Maximum matching has received a significant attention in the MPC model as well; see, e.g. [111, 48, 12, 17, 14, 69, 68, 29, 66, 6, 133] and references therein. In particular, a $(1-\epsilon)$-approximation to maximum matching is known in $O(1/\epsilon)$ rounds when memory per machine is super-linear, i.e., $n^{1+\Omega(1)}$ [6] (see also [111]), in $(1/\epsilon)^{O(1/\epsilon)} \cdot \log\log n$ rounds when memory is linear [14, 68] (see also [48, 29]), and in $(1/\epsilon)^{O(1/\epsilon)} \cdot \sqrt{\log n}$ rounds when memory is strongly sublinear [69, 133].

Our auction algorithm gives an exponential improvement on the round complexity of prior algorithms for the case of bipartite graphs in linear and strongly sublinear memory regimes.

Theorem 4.1.2. *There are randomized massively parallel computation algorithms for the bipartite matching problem that for any $\epsilon > 0$ (not necessarily a constant) give a $(1-\epsilon)$-approximation in:*

- *$O(1/\epsilon^2 \cdot \log\log n)$ rounds and $O(n)$ memory per machine; and,*
- *$O(1/\epsilon^2 \cdot \sqrt{\log n})$ rounds and $O(n^{\alpha})$ memory per machine for any constant $\alpha > 0$.*

(The dependence on n in number of rounds for both results can be replaced by the maximum degree).

These algorithms are also simply obtained by using known MPC algorithms for computing maximal matchings in $O(\log\log n)$ rounds and $O(n)$ memory [29] and $O(\sqrt{\log n})$ rounds and $O(n^{\alpha})$ memory [69] in each iteration of the auction algorithm.

In the remainder of the chapter, we first give the details of our auction algorithm and then present a short proof of each of its applications in Theorems 4.1.1 and 4.1.2. Before that, a quick notation:

Notation. For any vertex v in a graph $G = (V, E)$, $N(v)$ denotes the neighbors of v. We use $\mu(G)$ to denote the maximum matching size of G.

4.2 A New Auction Algorithm

We design a new auction algorithm for finding a $(1-\epsilon)$-approximate matching in a bipartite graph $G = (L \sqcup R, E)$. We refer to vertices in L as 'bidders' and to vertices in R as 'items'.

Notation. For any bidder $i \in L$, we define the *valuation* of i for items in R as the function $v_i : R \to \{0, 1\}$ where $v_i(j) = 1$ if $j \in N(i)$ and $v_i(j) = 0$ otherwise. Similarly, we define the *utility* of a bidder i when given item a_i as $u_i := v_i(a_i) - p_{a_i}$, namely, the value of the allocated item a_i for

the bidder minus the price that i has to play for the item; the utility of an unallocated bidder i is $u_i = 0$.

A new auction algorithm for bipartite matching.

(i) For each bidder $i \in L$, let a_i denote the item allocated to i (initially $a_i = \perp$ to mean i is unallocated/unmatched).

(ii) For each item $j \in R$, set the 'price' of j to be $p_j = 0$.

(iii) For $r = 1$ to $R := \lceil \frac{2}{\epsilon^2} \rceil$ iterations:

(a) For each unallocated bidder $i \in L$, define $D_i := \arg\min_{j \in N(i), p_j < 1} p_j$, as the *demand set* of i, namely, the set of items in the neighborhood of i that have minimal price < 1.

(b) Consider the subgraph G_r between bidders and their demand sets. Let M_r be **any maximal matching** of G_r.

(c) For any bidder-item pair $(i, j) \in M_r$, allocate j to i by setting $a_i = j$ and $a_{i'} = \perp$ for the previous owner i' of j. Increase the price of j to $p_j = p_j + \epsilon$.

(iv) Return the matching M as (i, a_i) for all $i \in L$ where $a_i \neq \perp$.

The algorithm clearly runs in $O(1/\epsilon^2)$ iterations and each iteration requires computing an arbitrary maximal matching of a subgraph of G. We now prove the correctness of the algorithm.

Let M^* be a maximum matching of G and $\mathsf{OPT} \subseteq L$ be the set of bidders in L that are matched by M^*. For any $i \in \mathsf{OPT}$, we further use $o_i \in R$ to denote the item that is allocated to i in M^*.

Definition 4.2.1. *We say that a bidder i is* ϵ-happy *if and only if $u_i \geq v_i(j) - p_j - \epsilon$ for all $j \in N(i)$, i.e., changing the allocation of i to any other item does not increase the utility of i by more than ϵ.*

A simple observation is in order.

Observation 4.2.2. *In every iteration of the auction, allocated bidders as well as unallocated bidders with empty demand sets are ϵ-happy.*

Observation 4.2.2 holds as: (i) each allocated bidder picked the minimum price item in its neighborhoods (and increased its price only by ϵ), and the price of items are monotonically increasing;

(*ii*) every item in the neighborhoods of an unallocated bidder with empty demand set has price 1 and thus picking that item cannot increase the utility of the bidder by more than ϵ.

We first show that if a "large" number of bidders in OPT become ϵ-happy at *any* point in the auction, then the final matching M achieves a $(1-\epsilon)$-approximation.

Lemma 4.2.3. *Suppose at the end of some iteration of the auction, $(1-\epsilon)\cdot\mu(G)$ bidders in OPT are ϵ-happy. Then, the final matching M has size at least $(1-2\epsilon)\cdot\mu(G)$.*

Proof. Consider the allocation $a_1,\ldots,a_n$ and prices $p_1,\ldots,p_n$ in such an iteration. Let Happy denote the set of all ϵ-happy players. For any bidder $i\in\mathsf{Happy}\cap\mathsf{OPT}$, by Definition 4.2.1,

$$u_i \geq v_i(o_i) - p_{o_i} - \epsilon,$$

and for any other bidder $u_i \geq 0$ as utility is never negative. Summing over all ϵ-happy bidders,

$$\begin{aligned}\sum_{i\in\mathsf{Happy}} u_i &\geq \sum_{i\in\mathsf{Happy}\cap\mathsf{OPT}} u_i \\ &\geq \sum_{i\in\mathsf{Happy}\cap\mathsf{OPT}} (v_i(o_i) - p_{o_i} - \epsilon) \\ &\geq (1-2\epsilon)\cdot\mu(G) - \sum_{i\in\mathsf{Happy}\cap\mathsf{OPT}} p_{o_i},\end{aligned}$$

where we used the fact that $v_i(o_i)=1$ and $|\mathsf{Happy}\cap\mathsf{OPT}|\geq(1-\epsilon)\cdot\mu(G)$. On the other hand,

$$\begin{aligned}\sum_{i\in\mathsf{Happy}} u_i &= \sum_{i\in\mathsf{Happy}\wedge a_i\neq\perp} (v_i(a_i) - p_{a_i}) \\ &\leq |M| - \sum_{i\in\mathsf{Happy}\wedge a_i\neq\perp} p_{a_i} \\ &= |M| - \sum_{j\in R} p_j,\end{aligned}$$

where: (*i*) the first equality holds because utility of unallocated bidders is zero, (*ii*) the first inequality holds because number of allocated bidders can only increase during the auction (and at the end is equal to the size of the final matching), and (*iii*) the final equality holds as all allocated bidders belong to Happy by Observation 4.2.2 and an item has non-zero price if and only if it is allocated.

By combining the above two inequalities, we have,

$$\begin{aligned}|M| &\geq (1-2\epsilon)\cdot\mu(G) + \left(\sum_{j\in R} p_j - \sum_{i\in \mathsf{Happy}\cap\mathsf{OPT}} p_{o_i}\right)\\ &\geq (1-2\epsilon)\cdot\mu(G),\end{aligned}$$

finalizing the proof. □

We now prove that in one of the iterations, there will be a large number of ϵ-happy bidders in OPT.

Lemma 4.2.4. *There exists an iteration $r \leq R$ wherein at least $(1-\epsilon)\cdot\mu(G)$ bidders in OPT are ϵ-happy.*

Proof. Consider the following two potential functions:

$$\Phi_{bidders} := \sum_{i\in\mathsf{OPT}} \min_{j\in N(i)} p_j, \qquad \Phi_{items} := \sum_{j\in R} p_j;$$

the first one (essentially) bounds the price of the demand set of bidder in OPT and the second one is simply the sum of all prices. Since the prices start at 0 and are capped at 1, we have $0 \leq \Phi_{bidders} \leq \mu(G)$. Moreover, since throughout the auction all items with non-zero prices remain allocated and we cannot have more than $\mu(G)$ allocated items, we also have $0 \leq \Phi_{items} \leq \mu(G)$. Finally, as the prices can only increase in the auction, both functions are monotone.

Consider any iteration of the auction wherein at least $\epsilon\cdot\mu(G)$ bidders in OPT are *not* ϵ-happy. By Observation 4.2.2, these bidders are necessarily unallocated. Since we are picking a maximal matching between unallocated bidders and their demand sets, at the end of this iteration, any of these bidders either become allocated or their demand set *at the current* prices becomes empty. Each of the former bidders contributes a value of ϵ to Φ_{items} (as the price of their allocated items increases by ϵ), while each of the latter bidders contributes ϵ to $\Phi_{bidders}$ (as minimum price of their neighborhoods increases by ϵ). As such, $\Phi_{bidder} + \Phi_{items}$ increases by $\epsilon^2\cdot\mu(G)$ in this iteration. Considering the maximum value of $\Phi_{bidder} + \Phi_{item}$ is at most $2\mu(G)$, we can only have $2/\epsilon^2$ such iterations, concluding the proof. □

Lemmas 4.2.4 and 4.2.3 now imply the following theorem.

Theorem 4.2.5. *The new auction algorithm terminates in a $(1-\epsilon)$-approximate matching of the input graph in $O(1/\epsilon^2)$ iterations no matter how the maximal matchings in each iterations are chosen.*

4.3 Applications

In this section, we mention some immediate applications of our auction algorithm.

Streaming Algorithms

Proof of Theorem 4.1.1. Recall that in each iteration r, we only need to compute an arbitrary maximal matching M_r in the subgraph G_r between bidders and their demand sets. The graph G_r is built (implicitly) in one pass: at the beginning of iteration r, we store whether a bidder i is allocated for all i, as well as the prices of all the items[2]; in one pass we compute $d_i = \min_{j \in N(i)} p_j$ for all i and store them. Finally, we read the stream for one more pass and maintain a current matching M_r starting with $\emptyset$: on arriving an edge (i, j), it is in G_r if and only if i is unallocated and $p_j = d_i < 1$, and add (i, j) into M_r if it does not share endpoint with the current M_r. Since each iteration takes two passes, Theorem 4.1.1 immediately follows from Theorem 4.2.5. □

Massively Parallel Computation (MPC) Algorithms

Proof of Theorem 4.1.2. Our MPC algorithms simply implement the streaming algorithm in the proof of Theorem 4.1.1 and apply the known MPC algorithms for maximal matching: each iteration of the auction algorithm can be implemented in $O(\log \log n)$ rounds and $O(n)$ memory per machine, or in $O(\sqrt{\log n})$ rounds and $O(n^\alpha)$ memory per machine. Using standard MPC techniques (sorting and prefix sum computation), one can compute $d_i = \min_{j \in N(i)} p_j$ for all i in $O(1)$ MPC rounds using $O(n)$ memory or $O(n^\alpha)$ memory per machine for any constant $\alpha > 0$. Since we store whether all bidders are allocated and all prices in the memories of machines, the subgraph G_r is built. Finally, we call the MPC algorithms for computing the maximal matching M_r in G_r in $O(\log \log n)$ rounds and $O(n)$ memory [29], or in $O(\sqrt{\log n})$ rounds and $O(n^\alpha)$ memory [69], completing the proof of Theorem 4.1.2. □

[2]This takes $O(n)$ space because there are $O(1/\epsilon)$ different prices where $\epsilon \geq 1/(n+1)$ - otherwise we could compute an exact matching, so a price can be stored in one word with $O(\log n)$ bits.

Chapter 5

Exact Maximum Weight Bipartite Matching

5.1 Background

As we mentioned in §1.4 in the introduction of this thesis, almost all existing streaming algorithms for maximum weight bipartite matching are only able to find an *approximate* solution. To the best of our knowledge, the only exception is a folklore algorithm that repeatedly finds an augmenting path by a breath-first search, which takes $O(n \log n)$ passes and $\widetilde{O}(n)$ space to find an exact maximum cardinality bipartite matching. Obtaining even an $O(n)$-pass semi-streaming algorithm for this problem seems to be a major challenge, which raises the following open problem:

> *Is there a semi-streaming algorithm for exact maximum weight bipartite matching that uses $o(n)$ passes?*

Another motivation for this problem is the two (folklore) reductions: (1) s-t reachability $\preceq$ maximum bipartite matching, and (2) s-t reachability $\preceq$ single source shortest path, all of which are fundamental problems in algorithmic graph theory. We note that s-t reachability [118] and single source shortest path [63, 104, 41] can be both solved in $n^{1/2+o(1)}$ passes in semi-streaming, whereas for exact maximum cardinality bipartite matching, only an $O(n \log n)$ passes upper bound is known, leaving a huge open gap. To the best of our knowledge, [104, 41] are also the only two semi-streaming algorithms with an $n^{\Omega(1)}$ pass bounds. As such, a further motivation for answering our problem is a better understanding on the power of semi-streaming in the multi-pass regime.

Our main result in this chapter is an affirmative answer to the above problem, obtaining a sublinear-pass bipartite matching algorithm through *space-efficient interior-point method* (IPM). The IPM optimization framework was proposed by Karmarkar [101] as a powerful tool for improving the running time of linear programming solver and, subsequently, for several specific graph-type linear programming. Alas, in contrast to classical IPM solvers, which do not have a space constraint, in the streaming setting the algorithm is not allowed to store all the m constraints of the (n-variable) LP.

Indeed, the most enduring message of our result is showing that *interior point method for maximum bipartite matching can be implemented in the semi-streaming model in a non-trivial space efficient manner, whereas for general matrices, LP solvers are impossible to be implemented in near-linear space.* More precisely, we present a semi-streaming algorithm that takes $\widetilde{O}(\sqrt{m})$ IPM iterations to compute the maximum weight bipartite matching, where each iteration employs $O(\log n)$ passes to implement a streaming version of an SDD solver for computing the Newton step of the path-following IPM. To the best of our knowledge, our algorithm is the first to implement IPM in space-limited models such as the streaming model.

More specifically, to solve the maximum weight bipartite matching problem in $\widetilde{O}(n)$ space, we consider its dual, the *generalized minimum vertex cover* problem in bipartite graph, and solve this problem using a streaming version of IPM. Therefore, this work also yields an $\widetilde{O}(\sqrt{m})$-pass semi-streaming algorithm for the minimum vertex cover problem in bipartite graph. To transform the dual solution to a primal solution in $\widetilde{O}(n)$ space, we use the isolation lemma, whose original construction uses $\widetilde{\Theta}(m)$ random bits [129]. We bypass this issue by implementing an alternative construction proposed by [42] that uses $\widetilde{O}(n)$ random bits in the semi-streaming model.

5.1.1 Our Contribution

We give our main result and two byproducts as follows.

We provide a semi-streaming algorithm for bipartite matching running in $\widetilde{O}(\sqrt{m})$ passes. This solves the long standing open problem of whether maximum matching can be solved in $o(n)$ passes, in all regimes where $m = n^{2-\epsilon}$ for any constant $\epsilon > 0$.

Theorem 5.1.1 (Main theorem, informal version of Theorem 5.11.1)**.** *Given a bipartite graph G with n vertices and m edges, there exists a streaming algorithm that computes an (exact) maximum weight matching of G in $\widetilde{O}(\sqrt{m})$ passes and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$.*

We also show that minimum vertex cover in bipartite graph, which is the dual problem of

maximum matching, can be solved within the same number of passes and space. A *vertex cover* of a graph is a set of vertices that includes at least one endpoint of every edge of the graph, and a *minimum vertex cover* is a vertex cover of minimum size.

Theorem 5.1.2 (Informal version of Theorem 5.10.8)**.** *Given a bipartite graph G with n vertices and m edges, there exists a streaming algorithm that computes a minimum vertex cover of G in $\widetilde{O}(\sqrt{m})$ passes and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$.*[1]

Our main result is obtained by a novel application of IPM together with SDD system solver in the streaming model. To the best of our knowledge, this is the first work that implements those techniques in the streaming model in which only $\widetilde{O}(n)$ space is allowed. We put our result of SDD solver here for possible independent interests.

Theorem 5.1.3 (Informal version of Theorem 5.9.9)**.** *There is a streaming algorithm which takes input an SDD matrix A, a vector $b \in \mathbb{R}^n$, and a parameter $\epsilon \in (0,1)$, if there exists $x^* \in \mathbb{R}^n$ such that $Ax^* = b$, then with probability $1 - 1/\operatorname{poly}(n)$, the algorithm returns an $x \in \mathbb{R}^n$ such that*

$$\|x - x^*\|_A \leq \epsilon \cdot \|x^*\|_A$$

in $O(1 + \log(1/\epsilon)/\log\log n)$ passes and $\widetilde{O}(n)$ space.[2]

We hope the tools developed in this dissertation will find further applications in the study of streaming algorithms and beyond.

5.1.2 Related Work

Streaming algorithms for matching. Maximum matching has been extensively studied in the streaming model for decades, where almost all of them fall into the category of approximation algorithms. For algorithms that only make one pass over the edges stream, researchers make continuous progress on pushing the constant approximation ratio above 1/2, which is under the assumption that the edges are arrived in a uniform random order [93, 13, 59, 30]. The random-order assumption makes the problem easier (at least algorithmically). A more general setting is multi-pass streaming with adversarial edge arriving. Under this setting, the first streaming algorithm that beats the 1/2-approximation of bipartite cardinality matching is [61], giving a $2/3 \cdot (1-\epsilon)$-approximation in

[1] We can actually solve a *generalized* version of the minimum vertex cover problem in bipartite graph: each edge e needs to be *covered* for at least $b_e \in \mathbb{Z}^+$ times, where the case of $b = \mathbf{1}_m$ is the classic minimum vertex cover.

[2] Our algorithm can actually solve any SDD_0 system, which is more general than an SDD system. See a formal definition of SDD_0 matrix in §5.3.

$1/\epsilon \cdot \log(1/\epsilon)$ passes. The first to achieve a $(1-\epsilon)$-approximation is [124], which takes $(1/\epsilon)^{1/\epsilon}$ passes.[3] Since then, there is a long line of research in proving upper bounds and lower bounds on the number of passes to compute a maximum matching in the streaming model [3, 57, 72, 57, 91, 54, 5, 20, 22, 21] (see next subsection for more details). Notably, [3, 5] use linear programming and duality theory (see the next subsection for more details).

However, all the algorithms above can only compute an approximate maximum matching: to compute a matching whose size is at least $(1-\epsilon)$ times the optimal, one needs to spend $\text{poly}(1/\epsilon)$ passes (see [54, 5] and the references therein).

Despite the intimate relationship between the matching problem and the streaming model, no breakthrough is made towards solving exact bipartite matching in the streaming model. We remark that a simple folklore algorithm inspired by the classic algorithm of Hopcroft and Karp [84] can actually find the exact maximum cardinality bipartite matching in $\widetilde{O}(n)$ passes using $\widetilde{O}(n)$ space, whose details can be found in the next subsection.

Streaming spectral sparsifer. Initialized by the study of cut sparsifier in the streaming model [1], a simple one-pass semi-streaming algorithm for computing a spectral sparsifier of any weighted graph is given in [102], which suffices for our use in this chapter. The problem becomes more challenging in a dynamic setting, i.e., both insertion and deletion of edges from the graph are allowed. Using the idea of linear sketching, [95] gives a single-pass semi-streaming algorithm for computing the spectral sparsifier in the dynamic setting. However, their brute-force approach to recover the sparsifier from the sketching uses $\Omega(n^2)$ time. An improved recover time is given in [97] but requires more spaces, e.g., $\epsilon^{-2}n^{1.5}\log^{O(1)} n$. Finally, [98] proposes a single-pass semi-streaming algorithm that uses $\epsilon^{-2}n\log^{O(1)} n$ space and $\epsilon^{-2}n\log^{O(1)} n$ recover time to compute an ϵ-spectral sparsifier which has $O(\epsilon^{-2}n\log n)$ edges. Note that $\Omega(\epsilon^{-2}n\log n)$ space is necessary for this problem [39].

SDD solver and IPM. There is a long line of work focusing on fast SDD solvers [149, 108, 109, 103, 45, 136, 110]. Spielman and Teng give the first nearly-linear time SDD solver, which is simplified with a better running time in later works. The current fastest SDD solver runs in $O(m\log^{1/2} n\,\text{poly}(\log\log n)\log(1/\epsilon))$ time [45]. All of them require $\widetilde{\Theta}(m)$ space.

The interior point method was originally proposed by Karmarkar [101] for solving linear program. Since then, there is a long line of work on speeding up interior point method for solving classical

[3] For the weighted case, there is a $(1/2-\epsilon)$-approximation algorithm that only takes one pass [135].

optimization problems, e.g., linear program [157, 139, 158, 130, 113, 112, 113, 114, 46, 115, 147, 37, 35, 148, 88, 56],[4] and semi-definite program [11, 131, 132, 87, 86].

5.1.3 Previous Techniques

In this section, we summarize the previous techniques for computing the maximum matching in the streaming model.

- In §5.1.3, we introduce some representative approximation algorithms for bipartite matching.
- In §5.1.3, we present a potential method to compute an exact bipartite matching, showcasing the current bottleneck in the field.
- In §5.1.3, we discuss a simple folklore semi-streaming algorithm that uses $O(n \log n)$ passes, which is by far the best.
- In §5.1.3, we introduce some related bipartite matching algorithms which use more space than that permitted in the semi-streaming model.

Approximation Algorithms

Given a parameter $\epsilon \in (0, 1)$, many streaming algorithms are to find a matching of size $(1 - \epsilon)$ times the size of the maximum matching. The space and passes usages of these approximation algorithms are increasing functions of $1/\epsilon$.[5]

A natural idea to find an approximate matching is to iteratively sample a small subset of edges and use these edges to refine the current matching. These algorithms are called *sampling-based* algorithms. In [5], Ahn and Guha show that by adaptively sampling $\widetilde{O}(n)$ edges in each iteration, one can either obtain a certificate that the sampled edges admit a desirable matching, or these edges can be used to refine the solution of a specific LP. The LP is a nonstandard relaxation of the matching problem, and will eventually be used to produce a good approximate matching. The algorithm of Ahn and Guha can compute a $(1 - \epsilon)$-approximate matching for weighted (not necessarily bipartite) graph in $\widetilde{O}(1/\epsilon)$ passes and $\widetilde{O}(n \operatorname{poly}(1/\epsilon))$ space. However, the degree of $\operatorname{poly}(1/\epsilon)$ in the space usage can be very large, making their algorithm inapplicable for small (non-constant) $\epsilon = o(1/\log n)$ in the semi-streaming model.

[4] We remark that the fastest classical LP solvers are due to [113], [88], and [37]. The space requirements for those algorithms are $O(nm)$ [113, 37] and $O((m + n)^2)$ [88].

[5] We will be focusing on approximate algorithms that find a matching that is *close to* (or can potentially be used to find) an exact maximum matching, so all the constant-approximate algorithms are not introduced here. We refer the interested readers to [15] and the references therein.

Finding a $(1-\epsilon)$-approximate maximum matching with no space dependence on ϵ requires different methods. Inspired by the well-studied *water filling* process in online algorithms (see [52] and the references therein), Kapralov proposes an algorithm that generalizes the water filling process to multiple passes [91]. This algorithm works in the vertex arrival semi-streaming model, where a vertex and all of its incident edges arrive in the stream together. The observation is that the water filling from pass $(k-1)$ to pass k follows the same manner as that in the first pass (with a more careful double-counting method), then solving differential equations gives a $(1-1/\sqrt{2\pi k})$-approximate matching in k passes.

Kapralov's algorithm removes the poly$(\log n)$ factor in the number of passes comparing to [3], giving a $(1-\epsilon)$-approximate maximum matching in $O(1/\epsilon^2)$ passes, albeit in a stronger vertex arrival model. Recall that in the previous chapter, we gave a simple semi-streaming algorithm that computes a $(1-\epsilon)$-approximate maximum matching in $O(1/\epsilon^2)$ passes, removing the vertex arrival condition [21]. The algorithm considers the left vertices in a bipartite graph as *bidders* and the right vertices as *items*, then the maximum cardinality bipartite matching problem is to find an auction maximizing the social welfare. The algorithm only needs to maintain a price for each item. In each round, every unmatched bidder chooses the item that maximizes his own revenue. They show that a simple auction scheme converges in $O(1/\epsilon^2)$ rounds (passes). The space usage is $O(n\log(1/\epsilon))$ for storing all the prices, which is $\widetilde{O}(n)$ since $\epsilon \geq 1/(n+1)$ (otherwise we could obtain an exact matching).

From Approximate to Exact Maximum Matching

A potential method to compute an exact maximum matching is to augment an approximate matching by repeatedly finding augmenting paths.[6] The state-of-the-art semi-streaming algorithm based on auctions by [21] takes $O(1/\epsilon^2)$ passes to find a $(1-\epsilon)$-approximate maximum cardinality bipartite matching, where $\epsilon \geq 1/(n+1)$ is a parameter. (For $\epsilon = \Omega(1/\log n)$, there could be algorithms that take fewer passes. Recall that the algorithm of [5] does not work in semi-streaming when ϵ is too small.) So given $o(n)$ passes, no existing algorithm can find a matching of size at least $\text{OPT} - O(\sqrt{n})$, assuming the maximum matching has size $\text{OPT} = \Theta(n)$. Therefore, we are only able to obtain a matching of size $\text{OPT} - \Omega(\sqrt{n})$ from previous approximation algorithms. However, currently there is no semi-streaming algorithm that solves directed graph reachability, a problem that is no harder than finding one augmenting path, in $o(\sqrt{n})$ passes [118]. (The linear-work parallel algorithm of

[6] Given a matching in a graph, an *augmenting path* is a path that starts and ends at an unmatched vertex, and alternately contains edges that are outside and inside the matching.

[118] can be translated into a semi-streaming algorithm that finds an augmenting path in $n^{1/2+o(1)}$ passes.) So augmenting a matching of size $\mathrm{OPT} - \Omega(\sqrt{n})$ to the maximum matching also takes $\Omega(n)$ passes. In conclusion, based on existing algorithms, the method of augmenting an approximate matching does not yield an $o(n)$-pass semi-streaming algorithm. To break the n-pass barrier under this framework, we need either (1) an $o(n)$-pass approximation algorithm that finds a matching of size $\mathrm{OPT} - n^{1/2-\Omega(1)}$, or (2) an $n^{1/2-\Omega(1)}$-pass algorithm that finds at least one augmenting path.

A Folklore Algorithm with $O(n \log n)$ Passes

A simple folklore algorithm inspired by the classic algorithm of Hopcroft and Karp [84] can actually find the exact maximum cardinality bipartite matching in $\widetilde{O}(n)$ passes using $\widetilde{O}(n)$ space. The main idea is the following. Let OPT be the size of the maximum matching in the given n-vertex bipartite graph. If the current matching has size i, then there must exist $(\mathrm{OPT} - i)$ disjoint augmenting paths, so the shortest augmenting path has length at most $n/(\mathrm{OPT} - i)$. Using a breath-first search (simply ignore the edge in the stream that is not incident with the frontier of the breath-first search), one can find this path in $n/(\mathrm{OPT} - i)$ passes and augment the current matching. Therefore, the total number of passes to compute the perfect matching is at most

$$\sum_{i=0}^{\mathrm{OPT}-1} \frac{n}{\mathrm{OPT} - i} = n \cdot \sum_{i=1}^{\mathrm{OPT}} \frac{1}{i} = O(n \log n).$$

This simple algorithm is by far the state-of-the-art: to the best of our knowledge, before this work, no (semi-)streaming algorithm computes an exact matching in $o(n \log n)$ passes.

Other Algorithms with More Space

Since the breakthrough result from [122], new improvements have been made in solving the maximum weight bipartite matching problem, and more generally, the maximum flow problem (see [119] and the references therein). The maximum cardinality bipartite matching algorithm by [119] runs in $m^{4/3+o(1)}$ time and takes $\Omega(m)$ space, which is currently the fastest when the graph is sufficiently sparse. For moderately dense graph, a better algorithm is given in [36] very recently, which runs in $\widetilde{O}(m + n^{1.5})$ time and takes $\Omega(m)$ space. For the maximum weight bipartite matching problem, the $O(n^{\omega})$-time algorithm using matrix inversion where $\omega < 2.373$ (see [128]) requires $\Omega(n^2)$ space, and the $\widetilde{O}(m\sqrt{n})$-time algorithm by [113] requires $\Omega(m)$ space.

We remark that, except IPM, another method to solve the matching problem is to use simplex-type LP algorithms. However, those algorithms are mainly designed for the case where the dimension

n is small, thus they do not produce meaningful results under the graph setting where $n \leq m \leq n^2$. For instance, [40] shows that the non-recursive implementation of Clarkson's algorithm [44] gives a streaming LP solver that uses $O(n)$ passes and $\widetilde{O}(n\sqrt{m})$ spaces, which is $\Omega(m)$ space for graph problems. They also show that the recursive implementation gives a streaming LP solver that uses $n^{O(1/\delta)}$ passes and $(n^2 + m^\delta)\,\mathrm{poly}(1/\delta)$ spaces. Additionally, [16] proposes a streaming algorithm for solving n-dimensional LP that uses $O(nr)$ pass and $O(m^{1/r})\,\mathrm{poly}(n, \log m)$ space, where $r \geq 1$ is a parameter.

5.2 Our Techniques

Our approach is based on a novel combination of several techniques from both continuous optimization and combinatorics, specifically, interior point methods, streaming SDD solvers, LP duality, and the isolation lemma.

The key obstacle to overcome in this dissertation is the space limitation of implementing IPMs in the streaming model.

To better appreciate the implication of the $\widetilde{O}(n)$ space constraint, note that storing a matching already takes $\widetilde{\Theta}(n)$ space, meaning that we have only a polylog space overhead per vertex to store auxiliary information. The conventional way of solving maximum bipartite matching using an IPM solver would get stuck at the very beginning - maintaining the solution of the relaxed linear program, which is a fractional matching, already requires $\Omega(m)$ space for storing all LP constraints, which seems inevitable.

Our main contribution is fixing this problem by showing that a certain IPM ([139]) can actually be made to work in sublinear space (in the number of LP constraints) for *graph* constraint matrices. Our key insight is showing that solving the *dual* form of the above LP, which corresponds to the generalized (fractional) minimum vertex cover problem, is sufficient, and therefore only $\widetilde{O}(n)$ space is needed for maintaining a fractional solution. We use several techniques to establish this argument. The first idea is to use complementary slackness for the dual solution to learn which n edges will be in the final maximum matching and therefore reduce the size of the graph from m to n. However, this is not always the case: For instance, in a bipartite graph that admits a perfect matching, all left vertices form a minimum vertex cover, but the complementary slackness theorem gives no information on which edges are in the perfect matching. To circumvent this, we need to slightly perturb the weight on every edge, so that the minimum vertex cover (which is now unique) indeed provides enough information. We use the isolation lemma [129] to achieve this.

Our second contribution is implementing the isolation lemma in limited space. Perturbing the weight on every edge requires storing $\widetilde{O}(m)$ bits of randomness, since the perturbation should remain identical across two different passes. We bypass this issue by using the generalized isolation lemma proposed by [42], in which only $O(\log(Z))$ bits of randomness is needed, where Z is the number of candidates. In our case, $Z \leq n^n$ is the number of all possible matchings. So $\widetilde{O}(n)$ space usage perfectly fits into the semi-streaming model. We design an oracle that by storing $\widetilde{O}(n)$ random bits, will output the same perturbations for all edges in all passes.

Our third contribution is implementing an SDD (and Laplacian) solver in the semi-streaming model, which, to the best of our knowledge, was not done prior to our work.

The last piece of this work is to select a suitable IPM solver for our framework. Many recently-developed fast IPM solvers do not fit into $\widetilde{O}(n)$ space. For example, [119, 120] cannot handle the dual LP of matching, while the general LP solvers [113, 88, 37] need to maintain *both* primal and dual solutions. The matching solver [36] based on IPM also maintains both primal and dual solutions. Fortunately, we show that Renegar's IPM algorithm [139] can be implemented into a streaming version using all above techniques.

In the following subsections we elaborate on each of the above components:

- In §5.2.1, we provide a high-level picture of how [139]'s interior point method works.
- In §5.2.2, we describe our contribution on our implementations of SDD solver, IPM, and the isolation lemma in the streaming model.
- In §5.2.3, we show a novel application of the isolation lemma to turn dual into primal.

5.2.1 IPM Analysis

Given a matrix $A \in \mathbb{R}^{m \times n}$ and two vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, consider the following linear program

$$\min_{x \in \mathbb{R}^n} c^\top x \quad \text{s.t.} \quad Ax \geq b.$$

We briefly review several basic concepts in Renegar's algorithm [139] (see §5.5 and 5.6). Renegar's IPM solves the above LP by introducing two functions: one is the barrier function $F : \mathbb{R}^n \to \mathbb{R}$ and the other is the (parametrized) perturbed objective function $f_t : \mathbb{R}^n \to \mathbb{R}$. The standard log-barrier function can be written as

$$F(x) := \sum_{i \in [m]} \log(a_i^\top x - b_i),$$

where $a_1^\top, \cdots, a_m^\top$ are rows of matrix A. It is easy to see that the domain of $F(x)$ is exactly the region $\{x \in \mathbb{R}^n : Ax > b\}$. For any parameter $t > 0$, we define the perturbed function $f_t : \mathbb{R}^n \to \mathbb{R}$ as follows

$$f_t(x) := t \cdot c^\top x + F(x).$$

Renegar's method was based on a type of interior point methods known as the *path following method*, which incrementally minimizing $f_t(x)$. Once we get the minimizer x_t^* of $f_t(x)$ for some large t, it is not far from being optimal, e.g., $c^\top x_t^* \leq \mathrm{OPT} + m/t$.

From Renegar's framework, the potential function $\Phi_t(x) := \|\nabla f_t(x)\|_{H(x)}$ is used to measure how close from x to the minimizer x_t^*, where $H(x) := \nabla^2 F(x)$ is the Hessian matrix. $\Phi_t(x)$ is always greater or equal than 0. On the other hand, $\Phi_t(x) = 0$ means $x = x_t^*$. In each iteration, our goal is to increase t to t^{new}, move x to x^{new}, while keeping $\Phi_{t^{\mathrm{new}}}(x^{\mathrm{new}})$ small:

$$\Phi_{t^{\mathrm{new}}}(x^{\mathrm{new}}) = \Phi_t(x) + \underbrace{\Phi_t(x^{\mathrm{new}}) - \Phi_t(x)}_{x \text{ move}} + \underbrace{\Phi_{t^{\mathrm{new}}}(x^{\mathrm{new}}) - \Phi_t(x^{\mathrm{new}})}_{t \text{ move}}.$$

We are able to prove the following (see §5.7):

1. In the x moving part, let x^{new} be applying one newton step from x on function $f_t(x)$. Implementing Newton step requires an SDD solver. After that, we can guarantee the standard quadratic convergence rate of Newton method:

$$\Phi_t(x^{\mathrm{new}}) \leq \Phi_t(x)^2.$$

2. In the t moving part, we multiplicative increase t to t^{new}. We can guarantee that

$$\Phi_{t^{\mathrm{new}}}(x^{\mathrm{new}}) \leq \Phi_t(x^{\mathrm{new}}) + |t^{\mathrm{new}}/t - 1| \cdot \sqrt{m}.$$

Combining the above two parts, we can set $t^{\mathrm{new}} = (1 + O(1/\sqrt{m})) \cdot t$ so that an $\widetilde{O}(\sqrt{m})$-iteration IPM follows.

5.2.2 Our Implementations in the Streaming Model

Implement IPM in the streaming model. All the existing IPMs are working in classical model where there is no space limitation. One of our contribution is to implement IPM in streaming model.

We show that the iterating procedure in IPM can be implemented in the streaming model and each iteration can be done in $\widetilde{O}(1)$ passes (see §5.8). Recall that each iteration is performing a Newton step $(\nabla^2 F(x))^{-1}\nabla f_t(x) \in \mathbb{R}^n$. Note that $\nabla^2 F(x) \in \mathbb{R}^{n\times n}$ is an SDD matrix in our setting, and $\nabla^2 F(x)$, $\nabla f_t(x)$ both rely on the slack variable $s(x) := Ax - b \in \mathbb{R}^m$, but we cannot even store length-m vector b in the space. We address this problem by

- implementing an SDD solver in the streaming model;
- implementing the algorithm in the isolation lemma in the streaming model.

Implement the Laplacian solver in the streaming model. For simplicity, we outline a streaming Laplacian solver here, which gives an SDD solver by standard reductions (see §5.9). Given a graph G, let L_G be the Laplacian matrix of G. Let A be the signed edge-vertex incident matrix, and let $a_1^\top, \cdots, a_m^\top$ be rows of $A \in \mathbb{R}^{m\times n}$. Let $s \in \mathbb{R}^m$ be slack variable.

Note that L_G has $\mathrm{nnz}(L_G) = O(m)$ so it cannot be stored in memory, but it can be written as [7]

$$L_G = \sum_{i\in[m]} \frac{a_i a_i^\top}{s_i^2}$$

Suppose we have an oracle that outputs s_i for any given $i \in [m]$, then one can do simple things like multiplying L_G by a vector v, e.g., $L_G \cdot v$. This can be done since we can read one pass of all edges, and note that for edge $e_i = (u, v)$, we have $a_i = \mathbf{1}_u - \mathbf{1}_v$ so that $\frac{a_i a_i^\top}{s_i^2} \cdot y \in \mathbb{R}^n$ can be computed and stored. In fact, we show that this simple operation suffices: by [95], we can compute a sparsifier H of G. Using L_H, the Laplacian of H, as a preconditioner of L_G, we can do iterative refinement on the solution:

$$x^{(t+1)} := x^{(t)} + L_H^{-1} \cdot (b - L_G \cdot x^{(t)}).$$

After $\widetilde{O}(1)$ iterations, x will converge to the solution $x^* = L_G^{-1} y$ with desired precision.

Implement the isolation lemma in the streaming model. We start with the definition of the isolation lemma (see §5.4 for details).

Definition 5.2.1 (Isolation lemma)**.** *Given a set system $(S, \mathcal{F})$ where $\mathcal{F} \subseteq \{0,1\}^S$. Given weight w_i to each element i in S, the weight of a set F in $\mathcal{F}$ is defined as $\sum_{i\in F} w_i$. The goal of the isolation*

[7] We want to remark that another equivalent definition is $L_G = D_G - A_G$, where $D_G \in \mathbb{R}^{n\times n}$ is a diagonal matrix ($(D_G)_{i,i}$ is the weighted degree of vertex i) and A_G is the adjacency matrix of graph G, where $(A_G)_{i,j} = (A_G)_{j,i}$ is the weight of edge (i, j) in the graph.

lemma is to design a scheme that can assign weight oblivious to $\mathcal{F}$, such that there is a unique set in $\mathcal{F}$ that has the minimum (maximum) weight under this assignment.

The isolation lemma always involves randomness since its output is oblivious to $\mathcal{F}$. The isolation lemma says that if we randomly choose this weight, then with a good probability the uniqueness is ensured. However, this does not apply to the streaming setting since this weight vector is of length m, which cannot be stored in memory. The ideal is to *consistently* output the same weight vector in different passes.

[42] reduces the number of random bits from $|S|$ to $\log(|\mathcal{F}|) \leq |S|$. In the case of maximum weight bipartite matching, we have $|\mathcal{F}| \leq (n+1)^n$ since each of the n vertices can choose none or one of the vertices in the matching. Our streaming implementation (See Algorithm 2) is to store these $\log((n+1)^n) = \widetilde{O}(n)$ random bits, so that we can consistently output the same weight vector in different passes.

5.2.3 Turn Dual to Primal

To see how the isolation lemma helps, let us see a simple example.

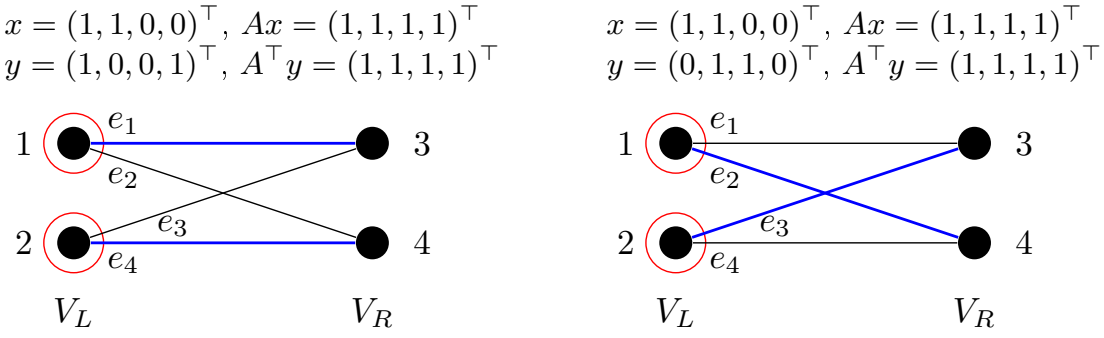

Figure 5.1: The red circle is a minimum vertex cover, which is an optimal dual solution. The blue edge is a maximum matching, which is an optimal primal solution. In both examples, the primal and dual satisfy complementary slackness Eq. (5.1).

Suppose the graph has a (maximum weight) perfect matching. Then the following trivial solution is optimal to the dual LP: choosing all vertices in V_L. Let us show what happens when applying the complementary slackness theorem. The complementary slackness theorem says that when y is a feasible primal solution and x is a feasible dual solution, then y is optimal to the primal and x is optimal to the dual **if and only if**

$$\langle y, Ax - \mathbf{1}_m \rangle = 0 \quad \text{and} \quad \langle x, \mathbf{1}_n - A^\top y \rangle = 0. \tag{5.1}$$

From the above case, we have $Ax - \mathbf{1}_m = 0$, so the first equality $\langle y, Ax - \mathbf{1}_m \rangle = 0$ puts no constraint on y. Therefore, any solution $y \geq \mathbf{0}_m$ to the linear system $a_i^\top y_i = 1,\ \forall i \in V_L$ is an optimal solution, where a_i is the i-th column of A. Note that this linear system has m variables and $|V_L|$ equations, which is still hard to find a solution in $\widetilde{O}(n)$ space.[8]

Roughly speaking, the complementary slackness theorem does not help in this case since choosing all vertices in V_L is always a feasible solution, and is actually an optimal solution when a perfect matching exists.

Now consider perturbing the primal objective function by some vector $b \in \mathbb{R}^m$ such that the optimal solution to the following primal LP is **unique**:

$$\textbf{Primal} \quad \max_{y \in \mathbb{R}^m} b^\top y, \qquad \text{s.t. } A^\top y \leq \mathbf{1}_n \text{ and } y \geq \mathbf{0}_m.$$

Suppose we find the optimal solution x in the dual LP and we want to recover the optimal solution y in the primal LP. Again by plugging in the complementary slackness theorem, we get at most n equations from the second part $\langle x, \mathbf{1}_n - A^\top y \rangle = 0$. Since the optimal y is unique and y has dimension m, the first part $\langle y, Ax - \mathbf{1}_m \rangle$ must contribute at least $m - n$ equations. Note that these equations have the form

$$y_i = 0,\ \forall i \in [m] \text{ s.t. } (Ax)_i - 1 > 0.$$

This means that the corresponding edges are *unnecessary* in order to get one maximum matching. As a result, we can reduce the number of edges from m to $O(n)$, then compute a maximum matching in $\widetilde{O}(n)$ space without reading the stream.

5.2.4 Roadmap

The rest of the chapter is organized as follows. In §5.3, we define the basic notations. In §5.4, we present the generalized isolation lemma in the semi-streaming model, which will be used to recover the maximum matching from a minimum vertex cover.

In §5.5, we give some preliminaries for interior point method. In §5.6, we present our streaming algorithm of interior point method. In §5.7, we present the error analysis of interior point method. In §5.8, we show the pass complexity of our interior point method.

In §5.9, we give an SDD solver in the streaming model, which is a necessary component of our interior point method. In §5.10, we discuss the streaming algorithm for minimum vertex cover. In

[8] In general, the inverse of a sparse matrix can be dense, which means the standard Gaussian elimination method for linear system solving does not apply in the semi-streaming model.

§5.11, we combine all the pieces together to get the final algorithm for maximum weight bipartite matching. In §5.12, we complement §5.9 by providing two reductions from weaker solvers to our final SDD_0 solver.

5.3 Notations

Notations. For a positive integer n, we denote $[n] = \{1, 2, \cdots, n\}$.

We use $\mathbb{E}[]$ for expectation and $\Pr[]$ for probability.

For a positive integer n, we use I_n to denote the identity matrix of size $n \times n$.

For a vector $v \in \mathbb{R}^n$, we use the standard definition of ℓ_p norms: $\forall p \geq 1$, $\|v\|_p = (\sum_{i=1}^n |v_i|^p)^{1/p}$. Specially, $\|v\|_\infty = \max_{i \in [n]} |v_i|$. We use $\|v\|_0$ to denote the number of nonzero entries in vector v. We use $\text{supp}(v)$ to denote the support of vector v.

We use $\mathbf{0}_n$ to denote a length-n vector where every entry is 0. We use $\mathbf{0}_{m \times n}$ to denote a $m \times n$ matrix where each entry is 0. Similarly, we use the notation $\mathbf{1}_n$ and $\mathbf{1}_{m \times n}$.

Matrix operators. For a square matrix A, we use $\text{tr}[A]$ to denote its trace. For a square and full rank matrix A, we use A^{-1} denote the true inverse of A. For a matrix A, we use $A^\dagger$ to denote its pseudo inverse. We say a square matrix A is positive definite, if for all x, $x^\top A x > 0$. We say a square matrix A is positive semi-definite, if for all x, $x^\top A x \geq 0$. We use $\succeq$ and $\preceq$ to denote the p.s.d. ordering. For example, we say $A \succeq B$, if $x^\top A x \geq x^\top B x, \forall x$.

Matrix norms. For a matrix A, we use $\|A\|_1$ to denote its entry-wise ℓ_1 norm, i.e., $\|A\|_1 = \sum_{i,j} |A_{i,j}|$. We use $\|A\|_F$ to denote its Frobenius norm $\|A\|_F = (\sum_{i,j} A_{i,j}^2)^{1/2}$. We use $\|A\|$ to denote its spectral/operator norm.

Matrix approximation. Let $A, B \in \mathbb{R}^{n \times n}$ be positive semi-definite matrix. Let $\epsilon \in (0, 1)$. We say $A \approx_\epsilon B$ if

$$(1 - \epsilon) x^\top A x \leq x^\top B x \leq (1 + \epsilon) x^\top A x, \quad \forall x \in \mathbb{R}^n$$

Note that if we have $A \approx_\epsilon B$, then $(1 - \epsilon)\|x\|_A \leq \|x\|_B \leq (1 + \epsilon)\|x\|_A$ for all $x \in \mathbb{R}^n$.

The SDD matrix-related definitions.

Definition 5.3.1 (Edge-vertex incident matrix)**.** *Let $G = (V_L, V_R, E)$ be a connected undirected*

bipartite graph. The (unsigned) edge-vertex incidence matrix *is denoted as follows*

$$B(e,v) = \begin{cases} 1, & \text{if } v \text{ incident to } e; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 5.3.2 (signed edge-vertex incident matrix)**.** *Let $G = (V_L, V_R, E)$ be a connected directed bipartite graph where all edges orientate from V_L to V_R. The* signed edge-vertex incidence matrix *is denoted as follows*

$$B(e,v) = \begin{cases} +1, & \text{if } v \text{ incident to } e \text{ and } v \in V_L; \\ -1, & \text{if } v \text{ incident to } e \text{ and } v \in V_R; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 5.3.3 (SDDM, SDD matrix)**.** *A square matrix A is* weakly diagonally dominant *if $A_{i,i} \geq \sum_{j \neq i} |A_{i,j}|$ for i, and is* strictly diagonally dominant *if $A_{i,i} > \sum_{j \neq i} |A_{i,j}|$ for i. A matrix A is SDD_0 if it is symmetric and weakly diagonally dominant, and is SDD if it is symmetric and strictly diagonally dominant. A matrix A is $SDDM_0$ if it is SDD and $A_{i,j} \leq 0$ for all $i \neq j$. An $SDDM_0$ matrix is SDDM if it is strictly diagonally dominant.*

Fact 5.3.4. *An $SDDM_0$ matrix must be positive semi-definite. If an $SDDM_0$ matrix has zero row-sums, then it is a Laplacian matrix. If an $SDDM_0$ matrix has at least one positive row-sum, then it is an SDDM matrix and positive definite.*

Bit complexity. Given a linear programming

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \ & c^\top x \\ \text{s.t.} \ & Ax \geq b, \end{aligned} \tag{5.2}$$

where $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$ all having integer coefficient.

The bit complexity L is defined as

$$L := \log(m) + \log(1 + \mathrm{d}_{\max}(A)) + \log(1 + \max\{\|c\|_\infty, \|b\|_\infty\}),$$

where $\mathrm{d}_{\max}(A)$ denotes the largest absolute value of the determinant of a square sub-matrix of A.

It is well known that poly(L)-bit precision is sufficient to implement an IPM (e.g., see [49] and

the references therein). This is because the absolute values of all intermediate arithmetic results are within $[2^{-\operatorname{poly}(L)}, 2^{\operatorname{poly}(L)}]$, and the errors in all the approximations are at least $1/\operatorname{poly}(n)$. Therefore, truncating all the arithmetic results to $\operatorname{poly}(L)$ bits for some sufficiently large polynomial preserves all the error parameters and thus the same analysis holds.

We will need the following tools in our later analysis.

Lemma 5.3.5 ([81])**.** *Let $A \in \mathbb{R}^{m\times n}$ be the unsigned edge-vertex incident matrix of a bipartite graph $G = (V_L, V_R, E)$. Let $S := \{Ax \le b \mid x \ge \mathbf{0}_m\}$, $S' := \{A^\top y \le b' \mid y \ge \mathbf{0}_n\}$, where $b \in \mathbb{Z}^m$, $b' \in \mathbb{Z}^n$. Then both A and $A^\top$ are totally unimodular, i.e., all square submatrices of them have determinants of $0, 1, -1$, and all extreme points of S and S' are integral.*

5.4 Isolation Lemma in the Streaming Model

[42] already shows how to implement isolation lemma using a small amount of randomness, and in particular, in our application, the amount of randomness is $O(n)$ and therefore just fits into our memory. However, since their focus is on the number of randomness, they still use extra space that we cannot tolerate. In this section, we make their algorithm into an oracle. This oracle only stores the random seed and performs exactly the same as the original algorithm, so that we can use it in the streaming model. Formally, our result is stated in Lemma 5.4.1.

Before we prove, we give a simple notation here.

For a vector $w \in \mathbb{Z}^n$ and a set $S \subseteq [n]$, we denote $w_S := \sum_{i\in S} w_i$. We can define $w_S := \sum_{i\in S} w_i$ similarly when $w_i : \mathbb{Z}^t \to \mathbb{Z}$ is a function.

Lemma 5.4.1 (Streaming implementation of isolation lemma)**.** *Let $n, \mathcal{F}, Z, w$ be define as in Lemma 5.4.2. The Algorithm 1 in Lemma 5.4.2 can be implemented into such an oracle $\mathcal{I}$: $\mathcal{I}$ can output w_i given any $i \in [n]$. Furthermore, $\mathcal{I}$ uses $O(\log(Z) + \log(n))$ space.*

Proof. Our $\mathcal{I}$ is the streaming implementation of Algorithm 2. It is easy to see that after running the procedure Initialize, the procedure Query(i) will output w_i given any $i \in [n]$.

This oracle stores $m, Z \in \mathbb{N}$ and $r \in \mathbb{N}^t$ in memory. Note that m, Z can be stored in $O(\log(Z) + \log(n))$ bits, and r can be stored in

$$O(t \cdot \log n) = O(\lceil \log(m)/\log(n) \rceil \cdot \log(n)) = O(\log(Z) + \log(n))$$

bits. And it is easy to see in Query, all computation can be done within $O(\log(Z) + \log(n))$ space. □

5.4.1 Isolation Lemma

We state the generalized isolation lemma from previous work [42].

Lemma 5.4.2 (Generalized isolation lemma [42])**.** *Fix $n \in \mathbb{N}$. Fix an unknown family $\mathcal{F} \subseteq [n]$. Let Z denote a positive integer such that $Z \geq |\mathcal{F}|$, there exists an algorithm (Algorithm 1) that uses $O(\log(Z) + \log(n))$ random bits to output a vector $w \in [0, n^7]^n$, such that with probability at least $1/4$, there is a unique set $S^* \in \mathcal{F}$ that has minimum weight w_{S^*}.*

Proof. The proof is already done in [42]. For the completeness, we rewrite their proof here. By Lemma 5.4.4, with probability at least $1/2$, all sets $S \in \mathcal{F}$ has distinct $w_S^{(2)}$. Under the event of we success, by Lemma 5.4.5, we get that all sets $S \in \mathcal{F}$ has distinct $w_S^{(3)}$. Then by Lemma 5.4.6, we get that with probability at least $1/2$, the output w will give a unique minimum set in $\mathcal{F}$. Since the two events of success are independent, the final success probability is at least $1/4$. □

5.4.2 Algorithms

We present an algorithm that implements the isolation lemma in the streaming model. We give the original implementation appeared in [42] in Algorithm 1. Then in Algorithm 2, we show how to implement the algorithm in the streaming model.

Algorithm 1 Conceptual implementation of the isolation lemma. Algorithm 2 is the implementation of this algorithm in streaming model.

procedure ISOLATION($n, Z \in \mathbb{N}$) ▷ $Z \geq |\mathcal{F}|$, Lemma 5.4.2
  $w_i^{(1)} \leftarrow 2^i$, $\forall i \in [n]$ ▷ $w^{(1)} \in \mathbb{N}^n$
  Choose m uniformly at random from $\{1, 2, \cdots, (2nZ^2)^2\}$. ▷ $m \leq 4n^2Z^4$
  For each i, define $w_i^{(2)} \leftarrow w_i^{(1)} \mod m$. ▷ $w^{(2)} \in [m]^n$
  $t \leftarrow \lceil \log(m)/\log(n) \rceil$
  for $i = 1 \to n$ **do**
    $\overline{b_{i,t-1}, \cdots, b_{i,1}, b_{i,0}} \leftarrow w_i^{(2)}$ ▷ Write $w_i^{(2)}$ in base n. $b_{i,j} \in [n]$ are digits.
    ▷ Note that t is an upper bound on the length
    $w_i^{(3)}(y_0, \cdots, y_{t-1}) \leftarrow \sum_{j=0}^{t-1} b_{i,j} \cdot y_j$ ▷ $w_i^{(3)} : \mathbb{Z}^t \to \mathbb{Z}$ is a linear form.
  end for
  Choose $r_0, \cdots, r_{t-1}$ uniformly and independently at random from $\{1, 2, \cdots, n^5\}$.
  $w_i^{(4)} \leftarrow w_i^{(3)}(r_0, \cdots, r_{t-1})$, $\forall i \in [n]$
  return $w^{(4)} \in \mathbb{N}^n$.
end procedure

Algorithm 2 Data structure: Isolation(). This algorithm is the streaming implementation of Algorithm 1.

data structure ▷ Lemma 5.4.1

members

$n, t \in \mathbb{N}$ ▷ $t = O(\log(Z)/\log(n))$

$m, Z \in \mathbb{N}$ ▷ $m = O(n^2 Z^4)$, $Z \geq |\mathcal{F}|$

$r_0, \cdots, r_{t-1} \in \mathbb{N}$ ▷ $r_i \leq n^5$

end members

procedure Initialize($n, Z \in \mathbb{N}$) ▷ Initialization

Choose $m \in \mathbb{N}$ uniformly at random from $\{1, 2, \cdots, (2nZ^2)^2\}$

$t \leftarrow \lceil \log m / \log n \rceil$ ▷ $t \in \mathbb{N}$

Choose $r_0, \cdots, r_{t-1}$ uniformly and independently at random from $\{1, 2, \cdots, n^5\}$ ▷ $r \in \mathbb{N}^t$

end procedure

procedure Query($i \in [n]$)

$w^{(1)} \leftarrow 2^i$ ▷ $w^{(1)} \in \mathbb{N}$, $w^{(1)} \leq 2^n$

$w^{(2)} \leftarrow w_i^{(1)} \mod m$ ▷ $w^{(2)} \in \mathbb{N}$, $w^{(2)} \leq 2^n$

$\overline{b_{t-1}, \cdots, b_1, b_0} \leftarrow w^{(2)}$ ▷ $b_j \in [n]$, $\forall j \in [t]$

▷ Write $w^{(2)}$ in base n. Note that t is an upper bound on the length

$w^{(4)} \leftarrow \sum_{j=0}^{t-1} b_j \cdot r_j$

return $w^{(4)}$ ▷ $w^{(4)} \leq n^7$

end procedure

end data structure

5.4.3 Proof of Uniqueness: Step 1

The following lemma is from [155].

Lemma 5.4.3 (Step 1, [155]). *Let $L \geq 100$ and let S be any subset of $\{1, \cdots, L^2\}$ such that $|S| \geq \frac{1}{2}L^2$. Then, the least common multiple of the elements in S exceeds 2^L.*

5.4.4 Proof of Uniqueness: Step 2

Lemma 5.4.4 (Step 2). *With probability at least $1/2$, all sets $S \in \mathcal{F}$ have distinct weights $w_S^{(2)}$.*

Proof. We write $\mathcal{F} = \{S_1, \cdots, S_k\}$ where $k \leq Z$. Define

$$I := \prod_{1 \leq i < j \leq k} |w_{S_i}^{(1)} - w_{S_j}^{(1)}|.$$

First note that $I \neq 0$ since every set $S \in \mathcal{F}$ have distinct weights $w_S^{(1)}$ by definition of $w_i^{(1)} = 2^i$ (Line 2, Algorithm 1).

Next, we give an upper bound on I. For each pair of $1 \leq i < j \leq k$, $|w_{S_i}^{(1)} - w_{S_j}^{(1)}| \leq 2^{n+1}$. There are totally Z^2 pairs, so $I < 2^{2nZ^2}$.

Let $L = 2nZ^2$. Let $S = \{1, \cdots, L^2\}$ be all the possible choices of m. We have that at least half choices of m satisfies $I \mod m \neq 0$, since otherwise by applying Lemma 5.4.3, I is at least the least common multiplier of half numbers of S, i.e., $I \geq 2^L$, contradicting with the upper bound on $I < 2^{2nZ^2}$.

Therefore, with probability at least $1/2$ (the randomness is over the choice of m), $I \mod m \neq 0$, which means that all sets $S \in \mathcal{F}$ have distinct weight $w_S^{(2)}$ by our definition of $w_i^{(2)} = w_i^{(1)} \mod m$ (Line 4, Algorithm 1). □

5.4.5 Proof of Uniqueness: Step 3

Lemma 5.4.5 (Step 3). *If $w_S^{(2)}$ are distinct for all $S \in \mathcal{F}$, then the linear form $w_S^{(3)}$ are all distinct for all $S \in \mathcal{F}$.*

Proof. Use proof by contradiction. Suppose there exists two distinct sets $S_1, S_2 \in \mathcal{F}$ that $w_{S_1}^{(3)} = w_{S_2}^{(3)}$. Let $y_i = n^i$, $\forall i \in [t]$. We will have

$$w_{S_1}^{(2)} = w_{S_1}^{(3)}(y_0, \cdots, y_{t-1}) = w_{S_2}^{(3)}(y_0, \cdots, y_{t-1}) = w_{S_2}^{(2)}$$

by definition of $w_i^{(3)}$ (Line 9), which is a contradict to our assumption that all $w_S^{(2)}$ are distinct. □

5.4.6 Proof of Uniqueness: Step 4

Lemma 5.4.6 (Step 4)**.** *Let $\mathcal{C}$ be any collection of distinct linear forms over at most t variables $y = y_0, \cdots, y_{t-1}$ with coefficients in $\{0, 1, \cdots, n-1\}$. Choose a random $r = r_0, \cdots, r_{t-1}$ by assigning each r_i uniformly and independently from $[n^2 \cdot t]$. Then in the assignment $y = r$ there will be a unique linear form with minimum value, with probability at least $1/2$.*

Proof. We call a variable y_i to be *singular* under an assignment $r \in \mathbb{Z}^t$ if there exists two minimum linear forms in $\mathcal{C}$ under assignment r. Then an assignment r gives unique minimum linear form if and only if no variable y_i is singular under this assignment. We will calculate the probability of y_i being singular under random assignment r and then take union bound over every y_i.

For each y_i, fix all $r_0, \cdots, r_{i-1}, r_{i+1}, \cdots, r_{t-1} = a_0, \cdots, a_{i-1}, a_{i+1}, \cdots, a_{t-1}$ other than r_i. Now, every linear form f under this partial assignment can be written as $a_f + b_f \cdot y_i$ with $b_f < n$. We split $\mathcal{C}$ into n classes $\mathcal{C}_0, \cdots, \mathcal{C}_{n-1}$ where $\mathcal{C}_j$ contains all linear forms with $b_f = j$. Let p_j be the minimum a_f among all linear forms in $\mathcal{C}_j$. According to the definition of singular, y_i is singular on assignment r_i if and only if the minimum value in the list

$$\{p_0, p_1 + r_i, p_2 + 2r_i, \cdots, p_{n-1} + (n-1) \cdot r_i\}$$

is not unique, which is upper bounded by the probability that the elements in the list has a collision. Since every pair of elements in the list can have at most one choice of r_i such that they are equal, we have

$$\begin{aligned}
\Pr_{r_i}[y_i \text{ is singular} \mid r_j = a_j,\ \forall j \neq i] &\leq \Pr_{r_i}[\exists l, m \text{ s.t. } p_l + l \cdot r_i = p_m + m \cdot r_i] \\
&\leq \sum_{l \neq m} \Pr_{r_i}[p_l + l \cdot r_i = p_m + m \cdot r_i] \\
&\leq \binom{n}{2} \cdot \frac{1}{n^2 t} \\
&\leq \frac{1}{2t}.
\end{aligned}$$

Finally, by a union bound on the events that every y_i is not singular, the conclusion follows with probability at least $1 - t \cdot \frac{1}{2t} = 1/2$. □

5.5 Preliminary for IPM

Let us consider the linear programming

$$\min_{x \in \mathbb{R}^n} c^\top x, \text{ s.t. } Ax \geq b$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$.

We denote $a_1^\top, \cdots, a_m^\top$ as the row vectors of matrix $A \in \mathbb{R}^{m \times n}$.

5.5.1 Definitions

In this section, we give definitions for barrier function, feasible solution, slack variables, gradient, hessian, perturbed objective function, Netwon step, potential functions.

Statement	Comments
Def. 5.5.1	Barrier function
Def. 5.5.2	Feasible solution
Def. 5.5.3	Slack variables
Def. 5.5.4	Gradient and Hessian with respect to barrier function
Def. 5.5.5	Perturbed objective function
Def. 5.5.6	Newton step
Def. 5.5.7	Φ_t, depends on t
Def. 5.5.8	Ψ, does not depend on t

Table 5.1: IPM related definitions.

Definition 5.5.1 (Barrier function)**.** *Define the* logarithmic barrier function $F(x) : \mathbb{R}^n \to \mathbb{R}$ *as follows:*

$$F(x) := -\sum_{i \in [m]} \ln(a_i^\top x - b_i). \tag{5.3}$$

Definition 5.5.2 (Feasible solution)**.** *For any $x \in \mathbb{R}^n$, we say x is feasible if for all $i \in [m]$, $a_i^\top x > b_i$.*

Definition 5.5.3 (Slack)**.** *For simplicity, we also define the* slack $s_i(x) := a_i^\top x - b_i \in \mathbb{R}$ *for all $x \in \mathbb{R}^n$ and $i \in [m]$.*

Definition 5.5.4 (Gradient and Hessian of barrier function)**.** *We define the gradient $g(x) : \mathbb{R}^n \to \mathbb{R}^n$*

and Hessian $H(x) : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ of the barrier function $F(x)$ as follows:

$$g(x) := \nabla F(x) = -\sum_{i \in [m]} \frac{a_i}{s_i(x)} \in \mathbb{R}^n \tag{5.4}$$

$$H(x) := \nabla^2 F(x) = \sum_{i \in [m]} \frac{a_i a_i^\top}{s_i(x)^2} \in \mathbb{R}^{n \times n}. \tag{5.5}$$

Definition 5.5.5 (Perturbed objective function)**.** *Define the* perturbed objective function $f_t : \mathbb{R}^n \to \mathbb{R}$*, where $t > 0$ is a parameter:*

$$f_t(x) := t \cdot c^\top x + F(x). \tag{5.6}$$

Definition 5.5.6 (Newton step)**.** *Given a scalar $t > 0$ and a vector $c \in \mathbb{R}^n$, define the* Newton step *at a feasible point $x \in \mathbb{R}^n$ to be*

$$\delta_x := -(H(x))^\dagger \cdot \nabla f_t(x) = -(H(x))^\dagger \cdot (t \cdot c + \nabla F(x)) = -(H(x))^\dagger \cdot (t \cdot c + g(x)). \tag{5.7}$$

Definition 5.5.7 (Definition of potential Φ)**.** *Given $t > 0$ and feasible $x, y \in \mathbb{R}$, we define function $\Phi_t : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$:*

$$\Phi_t(x, y) := \|\nabla f_t(x)\|_{H(y)^{-1}}.$$

Definition 5.5.8 (Helper of potential function Ψ)**.** *Given feasible $x, y \in \mathbb{R}$, we define function $\Psi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$:*

$$\Psi(x, y) := \|g(x)\|_{H(y)^{-1}}.$$

5.5.2 Approximation Tools from Previous Work

Lemma 5.5.9 (Hessian approximation, [140])**.** *Let f be a self-concordant function with domain D_f. Define $H(x) := \nabla^2 f(x)$. For any two feasible point $x, z \in D_f$, if $\|x - z\|_{H(x)} \leq 1/4$, then we have*

$$(1 - \|x - z\|_{H(x)})^2 \cdot H(z) \preceq H(x) \preceq (1 - \|x - z\|_{H(x)})^{-2} \cdot H(z).$$

Lemma 5.5.10 (Perturbed linear system [37])**.** *Let $H \in \mathbb{R}^{d \times d}$ be a symmetric positive definite matrix. Let $\epsilon \in (0, 1/80)$. Let $b, v \in \mathbb{R}^d$ such that $\|v\|_{H^{-1}} \leq \epsilon \|b\|_{H^{-1}}$. Let $y = H^{-1}(b + v) \in \mathbb{R}^d$.*

Then there is a symmetric matrix $\Delta \in \mathbb{R}^{d\times d}$, such that $y = (H+\Delta)^{-1}b \in \mathbb{R}^d$ and

$$(1+20\epsilon)^{-1}H \preceq H + \Delta \preceq (1+20\epsilon)H$$

Lemma 5.5.11 ([140])**.** *Let f be a self-concordant function with domain D_f. For some $x \in D_f$, define $H(x) := \nabla^2 f(x)$ and $g(x) := \nabla f(x)$. Define $n(x) := H(x)^{-1}g(x)$ as the Newton step at x. If $\|n(x)\|_{H(x)} \leq 1/4$, then f has a minimizer z that*

$$\|z - x\|_{H(x)} \leq \|n(x)\|_{H(x)} + \frac{3\|n(x)\|^2_{H(x)}}{(1-\|n(x)\|_{H(x)})^3}$$

Lemma 5.5.12 ([140])**.** *Let f be a self-concordant function with domain D_f. Let $c \in \mathbb{R}^n$ be the objective vector. Define $\mathrm{OPT} := \min_{x\in D_f}\langle c, x\rangle$. Denote $H(x) := \nabla^2 f(x)$. For any $x, y \in D_f$, we have*

$$\frac{\langle c, y\rangle - \mathrm{OPT}}{\langle c, x\rangle - \mathrm{OPT}} \leq 1 + \|y - x\|_{H(x)}.$$

Proof. Define $c_x := H(x)^{-1}c$. Define $B(x) \subseteq \mathbb{R}^n$ be the ellipsoid $B(x) := \{y \in \mathbb{R}^n \mid \|y-x\|_{H(x)} < 1\}$. Since f is self-concordant, we know $B(x) \subseteq D_f$. Therefore for all $0 \leq t < 1/\|c_x\|_{H(x)}$, $x - t\cdot c_x \in D_f$. This means

$$\langle c, x\rangle - t\cdot\|c_x\|^2_{H(x)} \geq \mathrm{OPT}.$$

By letting $t \to 1/\|c_x\|_{H(x)}$, we get

$$\|c_x\|_{H(x)} \leq \langle c, x\rangle - \mathrm{OPT}.$$

Hence

$$\begin{aligned}\frac{\langle c, y\rangle - \mathrm{OPT}}{\langle c, x\rangle - \mathrm{OPT}} &= 1 + \frac{\langle c_x, y - x\rangle_{H(x)}}{\langle c, x\rangle - \mathrm{OPT}} \\ &\leq 1 + \frac{\|c_x\|_{H(x)}\|y-x\|_{H(x)}}{\langle c, x\rangle - \mathrm{OPT}} \\ &\leq 1 + \|y - x\|_{H(x)},\end{aligned}$$

where the first step is by the definition of inner product that for some positive semi-definite matrix $S \in \mathbb{R}^{n\times n}$, $\langle x, y\rangle_S := \langle Sx, y\rangle$, the second step is by Cauchy-Schwarz inequality.

□

Lemma 5.5.13 (x_t^* is nearly-optimal to x^* [140])**.** *Given linear programming*

$$\min_{x \in \mathbb{R}^n, Ax \geq b} c^\top x,$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$. Let $x^* \in \mathbb{R}^n$ be the optimal solution of the above LP. For every $t > 0$, let $x_t^* \in \mathbb{R}^n$ be the x that minimizes $f_t(x)$ (Def. 5.5.5), it must be that

$$c^\top x_t^* - c^\top x^* < m/t.$$

Proof. Since $x_t^* \in \mathbb{R}^n$ is the minimizer of $f_t(x)$, we have $\nabla f_t(x_t^*) = t \cdot c + g(x_t^*) = \mathbf{0}_n$. By $g(x_t^*) = -t \cdot c \in \mathbb{R}^n$, we get

$$c^\top x_t^* - c^\top x^* = (g(x_t^*))^\top \cdot (x^* - x_t^*)/t. \tag{5.8}$$

Note that x^*, $x_t^* \in \mathbb{R}^n$ are both feasible, so we have

$$\begin{aligned}
(g(x_t^*))^\top \cdot (x^* - x_t^*) = & \sum_{i \in [m]} \frac{a_i^\top (x^* - x_t^*)}{s_i(x_t^*)} \\
= & \sum_{i \in [m]} \frac{s_i(x_t^*) - s_i(x^*)}{s_i(x_t^*)} \\
= & \, m - \sum_{i \in [m]} \frac{s_i(x^*)}{s_i(x_t^*)} \\
< & \, m.
\end{aligned}$$

where the first step is by Definition of $g(x)$ (Def. 5.5.4), the second step is by Definition of $s_i(x)$ (Def. 5.5.3) that $s_i(x) = a_i^\top x - b$.

Combining with (Eq. (5.8)), the lemma follows. □

Lemma 5.5.14 (Nearly-optimal output: value version [140])**.** *Given linear programming*

$$\min_{x \in \mathbb{R}^n, Ax \geq b} c^\top x,$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$. Let $x^* \in \mathbb{R}^n$ be the optimal solution of the above LP. Let $\epsilon_N \in (0, 1/10)$. If for some $t > 0$ and $x \in \mathbb{R}^n$ we have $\Phi_t(x, x) \leq \epsilon_N$ (Def. 5.5.7), then we have

$$c^\top x - c^\top x^* \leq \frac{m}{t} \cdot (1 + 2\epsilon_N).$$

Proof. Denote $\mathrm{OPT} := c^\top x^*$ as the best value we can get in LP.

Let z be the minimizer of f_t. From $\Phi_t(x, x) \leq \epsilon_N$, we know $\|n(x)\|_{H(x)} \leq \epsilon_N$, where $n(x) := H(x)^{-1}\nabla f_t(x)$.

By Lemma 5.5.11, we have

$$\|z - x\|_{H(x)} \leq 2\epsilon_N. \tag{5.9}$$

Plug Eq. (5.9) into Lemma 5.5.12, we get

$$\frac{c^\top z - \mathrm{OPT}}{c^\top x - \mathrm{OPT}} \leq 1 + 2\epsilon_N.$$

By Lemma 5.5.13, we have

$$c^\top z \leq \mathrm{OPT} + m/t.$$

By combining the above three inequalities, we get

$$c^\top x - \mathrm{OPT} \leq \frac{m}{t} \cdot (1 + 2\epsilon_N).$$

□

5.6 Algorithm

In this section, we present our IPM algorithm. Later in §5.7, we will give the error analysis of the output of this algorithm. In §5.8, we bound the pass number while executing this algorithm in the streaming model. Specifically, we implement and analyze streaming SDD solver in §5.9 which is used in Line 16 of this algorithm.

Algorithm 3 Interior point method

procedure InteriorPointMethod($A, b, c, t_{\text{start}}, x_{\text{start}}, t_{\text{final}}, \epsilon_\Phi, o$) ▷ Lemma 5.7.1

  $m, n \leftarrow$ dimensions of A

  ▷ $A \in \mathbb{R}^{m \times n}$ is the input matrix, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$

  ▷ $t_{\text{start}}, x_{\text{start}}$ satisfy $\Phi_{t_{\text{start}}}(x_{\text{start}}, x_{\text{start}}) \leq \epsilon_\Phi$

  ▷ $t_{\text{final}} \in \mathbb{R}$ is the final goal of t

  ▷ $o \in \{0, 1\}$. If o is 0, the algorithm decreases t, otherwise the algorithm increases t

  $\epsilon_x \leftarrow 1/100$

  $\epsilon_t \leftarrow 1/(100\sqrt{m})$

  $\epsilon_\Phi \leftarrow \epsilon_\Phi$ ▷ $\epsilon_\Phi \leq 1/100$ is given from outside

  $t \leftarrow t_{\text{start}}$

  $x \leftarrow x_{\text{start}}$

  $T \leftarrow O(\sqrt{m}) \cdot |\log(t_{\text{final}}/t_{\text{start}})|$ ▷ number of iterations

  for $k \leftarrow 1$ to T **do**

    $\delta_x := -H(x)^{-1} \cdot \nabla f_t(x)$ ▷ δ_x is only used for analysis, not involve computation

    ▷ $H(x) \in \mathbb{R}^{n \times n}$, $\nabla f_t(x) \in \mathbb{R}^n$

    $\widetilde{\delta}_x \leftarrow$ StreamLS($H(x), \nabla f_t(x), \epsilon_x \cdot \epsilon_\Phi$) ▷ Algorithm 4

    ▷ Compute any $\widetilde{\delta}_x$ that $\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \leq \epsilon_x \cdot \|\delta_x\|_{H(x)}$

    $x^{\text{new}} \leftarrow x + \widetilde{\delta}_x$

    if $o = 0$ **then**

      $t^{\text{new}} \leftarrow t \cdot (1 - \epsilon_t)$

    else

      $t^{\text{new}} \leftarrow t \cdot (1 + \epsilon_t)$

    end if

    if ($o = 0$ and $t < t_{\text{final}}$) or ($o = 1$ and $t > t_{\text{final}}$) **then**

      break

    end if

    $t \leftarrow t^{\text{new}}$, $x \leftarrow x^{\text{new}}$

  end for

  return x

end procedure

5.7 Error Analysis of IPM

The goal of this section is to present Lemma 5.7.1.

Parameters	ϵ_x	ϵ_t	ϵ_Φ
Meaning	$\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \leq \epsilon_x$	$\|t^{\mathrm{new}}/t - 1| = \epsilon_t$	$\Phi_t(x,x) \leq \epsilon_\Phi$
Value	$1/200$	$1/(100\sqrt{m})$	$1/100$

Table 5.2: Table of parameters occurs in Algorithm 3.

Lemma 5.7.1. *Given any matrix $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$. Let $x^* \in \mathbb{R}^n$ be the solution of the following linear programming*

$$\min_{x \in \mathbb{R}^n, Ax \geq b} c^\top x.$$

If InteriorPointMethod *is given $A, b, c, t_{\mathrm{start}}, x_{\mathrm{start}}, t_{\mathrm{final}}$ that satisfy $\Phi_{t_{\mathrm{start}}}(x_{\mathrm{start}}, x_{\mathrm{start}}) \leq \epsilon_\Phi$ (where $\epsilon_\Phi = 1/100$, see Table 5.2), then it outputs an answer x which is a* nearly-optimal solution*:*

$$c^\top x - c^\top x^* \leq \frac{m}{t_{\mathrm{final}}} \cdot (1 + 2\epsilon_\Phi).$$

Proof. Since our initial point x_{start} and t_{start} satisfy

$$\Phi_{t_{\mathrm{start}}}(x_{\mathrm{start}}, x_{\mathrm{start}}) \leq \epsilon_\Phi,$$

we can apply Lemma 5.7.3 to get

$$\Phi_{t_{\mathrm{final}}}(x, x) \leq \epsilon_\Phi.$$

Applying Lemma 5.5.14 on t_{final} and x with our choose of $\epsilon_\Phi = 1/100 < 1/10$, we get

$$c^\top x - c^\top x^* \leq \frac{m}{t_{\mathrm{final}}} \cdot (1 + 2\epsilon_\Phi).$$

□

5.7.1 Assumptions on Parameters

Assumption 5.7.2. *Let $\epsilon_x \in (0, 1/10)$, $\epsilon_t \in (0, 1/\sqrt{m})$ and $\epsilon_\Phi \in (0, 1/10)$ denote three fixed parameters such that the following inqueqality hold.*

$$16(1 + \epsilon_t) \cdot \epsilon_\Phi^2 + 16 \cdot \epsilon_x + \sqrt{m} \cdot \epsilon_t \leq \epsilon_\Phi.$$

5.7.2 Bounding Potential Function Φ

The goal of this section is to prove Lemma 5.7.3.

Lemma 5.7.3. *For each t, let $\Phi_t : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be defined as Definition 5.5.7. Assume Assumption 5.7.2 holds. If the input $x_{\mathrm{start}}, t_{\mathrm{start}}$ satisfies*

$$\Phi_{t_{\mathrm{start}}}(x_{\mathrm{start}}, x_{\mathrm{start}}) \leq \epsilon_\Phi,$$

then for all iteration $k \in [T]$, we have

$$\Phi_{t^{(k)}}(x^{(k)}, x^{(k)}) \leq \epsilon_\Phi.$$

Proof. We use proof by induction. In the basic case where $k = 0$, we have $t^{(0)} := t_{\mathrm{start}}$ and $x^{(0)} := x_{\mathrm{start}}$ so that the condition holds by assumption.

When $k \geq 1$, for ease of notation, we define $x = x^{(k-1)}$, $t = t^{(k-1)}$, $x^{\mathrm{new}} = x^{(k)}$, $t^{\mathrm{new}} = t^{(k)}$ and define $\delta_x = H(x)^{-1}\nabla f_t(x)$ as in Definition 5.5.6.

First, from induction hypothesis, we have $\Phi_t(x, x) \leq \epsilon_\Phi$. By definition of Φ (Definition 5.5.7), this means

$$\|\delta_x\|_{H(x)} \leq \epsilon_\Phi.$$

Next, we will show $\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \leq \epsilon_x$.

By calling StreamLS$(H(x), \nabla f_t(x), \epsilon_x)$ (Line 16, Algorithm 3) and applying Theorem 5.9.9, we have

$$\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \leq \epsilon_x \cdot \|\delta_x\|_{H(x)}.$$

Define $b := H(x)\delta_x$ and $v := H(x)(\widetilde{\delta}_x - \delta_x)$ and write $\widetilde{\delta}_x = H^{-1} \cdot (b + v)$. Note that we have $\|v\|_{H(x)^{-1}} \leq \epsilon_x \cdot \|b\|_{H(x)^{-1}}$. Apply Lemma 5.5.10 and we get

$$\widetilde{\delta}_x = \overline{H}^{-1} H(x)\delta_x,$$

where $\overline{H} \in \mathbb{R}^{n \times n}$ is a positive semi-definite matrix that satisfies

$$(1 + 20\epsilon_x)^{-1} H(x) \preceq \overline{H} \preceq (1 + 20\epsilon_x) H(x).$$

Therefore, we have

$$\|\widetilde{\delta}_x - \delta_x\|_{H(x)} = \|(\overline{H}^{-1}H - I)\delta_x\|_{H(x)} \leq 20\epsilon_x\|\delta_x\|_{H(x)} = 20\epsilon_x\epsilon_\Phi \leq \epsilon_x.$$

Finally, we have

$$\begin{aligned}
\Phi_{t^{\mathrm{new}}}(x^{\mathrm{new}}, x^{\mathrm{new}}) &\leq \frac{t^{\mathrm{new}}}{t} \cdot \Phi_t(x^{\mathrm{new}}, x^{\mathrm{new}}) + |t^{\mathrm{new}}/t - 1| \cdot \Psi(x^{\mathrm{new}}, x^{\mathrm{new}}) \\
&\leq \frac{t^{\mathrm{new}}}{t} \cdot \Phi_t(x^{\mathrm{new}}, x^{\mathrm{new}}) + |t^{\mathrm{new}}/t - 1| \cdot \sqrt{m} \\
&\leq (1+\epsilon_t) \cdot \Phi_t(x^{\mathrm{new}}, x^{\mathrm{new}}) + \sqrt{m} \cdot \epsilon_t \\
&\leq 16(1+\epsilon_t) \cdot \Phi_t(x,x)^2 + 16\epsilon_x + \sqrt{m} \cdot \epsilon_t \\
&\leq 16(1+\epsilon_t) \cdot \epsilon_\Phi^2 + 16\epsilon_x + \sqrt{m} \cdot \epsilon_t \\
&\leq \epsilon_\Phi,
\end{aligned}$$

where the first step is by Lemma 5.7.4, the second step is by Lemma 5.7.5, the fourth step is by Lemma 5.7.6 since $\epsilon_x + \epsilon_\Phi \leq 1/4$, the fifth step is by induction hypothesis, the last step is by Assumption 5.7.2. □

Section	**Lemma**	LHS	RHS
§5.7.2	Lemma 5.7.3	$\Phi_{t^{\mathrm{new}}}(x+\widetilde{\delta}_x, x+\widetilde{\delta}_x)$	ϵ_Φ
§5.7.3	Lemma 5.7.4	$\Phi_{t^{\mathrm{new}}}(x+\widetilde{\delta}_x, x+\widetilde{\delta}_x)$	$(t^{\mathrm{new}}/t)\cdot\Phi_t(.) + (1-t^{\mathrm{new}}/t)\cdot\Psi(.)$
§5.7.4	Lemma 5.7.5	$\Psi(x+\widetilde{\delta}_x, x+\widetilde{\delta}_x)$	$\sqrt{m}$
§5.7.5	Lemma 5.7.6	$\Phi_t(x+\widetilde{\delta}_x, x+\widetilde{\delta}_x)$	$\Phi_t(x,x)^2$
§5.7.6	Lemma 5.7.7	$\Phi_t(x+\widetilde{\delta}_x, x+\widetilde{\delta}_x)$	$\Phi_t(x+\delta_x, x+\widetilde{\delta}_x)$
§5.7.7	Lemma 5.7.8	$\Phi_t(x+\delta_x, x+\widetilde{\delta}_x)$	$\Phi_t(x+\delta_x, x+\delta_x)$
§5.7.8	Lemma 5.7.9	$\Phi_t(x+\delta_x, x+\delta_x)$	$\Phi_t(x,x)^2$

Table 5.3: Summary of movement of potential function.

5.7.3 Bounding the Movement of t

The goal of this section is to prove Lemma 5.7.4.

Lemma 5.7.4. *Let Φ and Ψ be defined in Definition 5.5.7 and Definition 5.5.8. Then, for all positive $t, t^{\mathrm{new}} > 0$ and feasible $x \in \mathbb{R}^n$, we have*

$$\Phi_{t^{\mathrm{new}}}(x,x) \leq \frac{t^{\mathrm{new}}}{t} \cdot \Phi_t(x,x) + |t^{\mathrm{new}}/t - 1| \cdot \Psi(x,x),$$

Proof. We have

$$\begin{aligned}\nabla f_{t^{\mathrm{new}}}(x) &= (t^{\mathrm{new}} \cdot c + g(x)) \\ &= \frac{t^{\mathrm{new}}}{t} \cdot (t \cdot c + g(x)) + \left(1 - \frac{t^{\mathrm{new}}}{t}\right) \cdot g(x) \\ &= \frac{t^{\mathrm{new}}}{t} \cdot \nabla f_t(x) + \left(1 - \frac{t^{\mathrm{new}}}{t}\right) \cdot g(x),\end{aligned}$$

where the first step follows from the definition of $f_t(x)$ (Definition 5.5.5), the second step follows from moving terms, and the last step follows from the definition of $f_t(x)$ (Definition 5.5.5).

Finally, we can upper bound $\Phi_{t^{\mathrm{new}}}(x, x)$ as follows:

$$\begin{aligned}\Phi_{t^{\mathrm{new}}}(x, x) &= \|\nabla f_{t^{\mathrm{new}}}(x)\|_{H(x)^{-1}} \\ &\leq \frac{t^{\mathrm{new}}}{t} \cdot \|\nabla f_t(x)\|_{H(x)^{-1}} + \left(1 - \frac{t^{\mathrm{new}}}{t}\right) \cdot \|g(x)\|_{H(x)^{-1}} \\ &= \frac{t^{\mathrm{new}}}{t} \cdot \Phi_t(x, x) + \left(1 - \frac{t^{\mathrm{new}}}{t}\right) \cdot \Psi(x, x).\end{aligned}$$

Thus, we complete the proof. □

5.7.4 Upper Bounding the Potential Function

The goal of this section is to prove Lemma 5.7.5.

Lemma 5.7.5. *Let function $\Psi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ be defined as Definition 5.5.8. For all feasible $x \in \mathbb{R}^n$, we have*

$$\Psi(x, x) \leq \sqrt{m}.$$

Proof. Let $z := (H(x))^{-1} g(x) \in \mathbb{R}^n$ so that $\Psi(x, x) = \|z\|_{H(x)}$.

We can upper bound $\|z\|_{H(x)}^2$ as follows:

$$\begin{aligned}
\|z\|_{H(x)}^2 &= z^\top H(x) z \\
&= z^\top g(x) \\
&= \sum_{i\in[m]} \frac{z^\top a_i}{s_i(x)} \\
&\leq \sqrt{m} \cdot \Big(\sum_{i\in[m]} \frac{(z^\top a_i)^2}{s_i(x)^2} \Big)^{1/2} \\
&= \sqrt{m} \cdot \Big(z^\top (\sum_{i\in[m]} \frac{a_i a_i^\top}{s_i(x)^2}) z \Big)^{1/2} \\
&= \sqrt{m} \cdot (z^\top H(x) z)^{1/2} \\
&= \sqrt{m} \cdot \|z\|_{H(x)},
\end{aligned}$$

where the third step is by $g(x) = \sum_{i\in[m]} \frac{a_i}{s_i(x)}$ (Definition 5.5.4), the four step is by Cauchy-Schwarz, and the sixth step is by $H(x) = \sum_{i\in[m]} \frac{a_i a_i^\top}{s_i(x)^2}$ (Definition 5.5.4).

Therefore, we obtain

$$\Psi(x,x) = \|z\|_{H(x)} \leq \sqrt{m}.$$

□

5.7.5 Move Both: Final

The goal of this section is to prove Lemma 5.7.6

Lemma 5.7.6. *For any feasible $x \in \mathbb{R}^n$, let $\delta_x \in \mathbb{R}^n$ be defined as Definition 5.5.6. Given $\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \leq \epsilon_x$ and $\|\delta_x\|_{H(x)} \leq \epsilon_1$. If $\epsilon_x + \epsilon_1 \leq 1/4$, then we have*

$$\Phi_t(x + \widetilde{\delta}_x, x + \widetilde{\delta}_x) \leq 16 \cdot \Phi_t(x,x)^2 + 16\epsilon_x.$$

Proof. The proof is done by showing the following:

$$\begin{aligned}
\Phi_t(x + \widetilde{\delta}_x, x + \widetilde{\delta}_x) &\leq \Phi_t(x + \delta_x, x + \widetilde{\delta}_x) + 16\epsilon_x \\
&\leq 16 \cdot \Phi_t(x + \delta_x, x + \delta_x) + 16\epsilon_x \\
&\leq 16 \cdot \Phi_t(x,x)^2 + 16\epsilon_x,
\end{aligned}$$

where the first step is by Lemma 5.7.7, the second step follows from Lemma 5.7.8, and the third step follows from Lemma 5.7.9. □

5.7.6 Move Both: Part 1

The goal of this section is to prove Lemma 5.7.7.

Lemma 5.7.7. *Given $\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \le \epsilon_x$ and $\|\delta_x\|_{H(x)} \le \epsilon_1$. If $\epsilon_x + \epsilon_1 \le 1/4$, then we have*

$$\Phi_t(x + \widetilde{\delta}_x, x + \widetilde{\delta}_x) \le \Phi_t(x + \delta_x, x + \widetilde{\delta}_x) + 16\epsilon_x.$$

Proof. We have

$$\begin{aligned}
&\Phi_t(x + \widetilde{\delta}_x, x + \widetilde{\delta}_x) - \Phi_t(x + \delta_x, x + \widetilde{\delta}_x) \\
&= \|\nabla f_t(x + \widetilde{\delta}_x)\|_{H(x+\widetilde{\delta}_x)^{-1}} - \|\nabla f_t(x + \delta_x)\|_{H(x+\widetilde{\delta}_x)^{-1}} \\
&\le \|\nabla f_t(x + \widetilde{\delta}_x) - \nabla f_t(x + \delta_x)\|_{H(x+\widetilde{\delta}_x)^{-1}} \\
&= \|g(x + \widetilde{\delta}_x) - g(x + \delta_x)\|_{H(x+\widetilde{\delta}_x)^{-1}}, \qquad (5.10)
\end{aligned}$$

where the first step is by the definition of Φ_t (Definition 5.5.7), and the second step is by the triangle inequality.

Define $\phi(t) := g\big(x + \delta_x + t \cdot (\widetilde{\delta}_x - \delta_x)\big)$, for $t \in [0, 1]$. Then we have

$$g(x + \widetilde{\delta}_x) - g(x + \delta_x) = \phi(1) - \phi(0).$$

By mean value theorem, there exists $p \in [0, 1]$ such that

$$\phi(1) - \phi(0) = \phi'(p) = H(x + \delta_x + p \cdot (\widetilde{\delta}_x - \delta_x)) \cdot (\widetilde{\delta}_x - \delta_x).$$

Since $\|x + \delta_x + p \cdot (\widetilde{\delta}_x - \delta_x) - x\|_{H(x)} \le \|\delta_x\|_{H(x)} + p \cdot \|\widetilde{\delta}_x - \delta_x\|_{H(x)} \le \epsilon + \epsilon_1 \le 1/4$, by Lemma 5.5.9 we have

$$\frac{1}{4} H(x) \preceq H(x + \delta_x + p \cdot (\widetilde{\delta}_x - \delta_x)) \preceq 4H(x). \qquad (5.11)$$

Since $\|x + \widetilde{\delta}_x - x\|_{H(x)} \le \|\delta_x\|_{H(x)} + \|\widetilde{\delta}_x - \delta_x\|_{H(x)} \le \epsilon_x + \epsilon_1 \le 1/4$, by Lemma 5.5.9 we have

$$\frac{1}{4}H(x) \preceq H(x + \widetilde{\delta}_x) \preceq 4H(x). \tag{5.12}$$

Finally, we have

$$\begin{aligned}
&\|g(x + \widetilde{\delta}_x) - g(x + \delta_x)\|_{H(x+\widetilde{\delta}_x)^{-1}} \\
&= \left\|H(x + \delta_x + p \cdot (\widetilde{\delta}_x - \delta_x)) \cdot (\widetilde{\delta}_x - \delta_x)\right\|_{H(x+\widetilde{\delta}_x)^{-1}} \\
&\le 4\left\|H(x + \delta_x + p \cdot (\widetilde{\delta}_x - \delta_x)) \cdot (\widetilde{\delta}_x - \delta_x)\right\|_{H(x)^{-1}} \\
&\le 16\left\|\widetilde{\delta}_x - \delta_x\right\|_{H(x)} \\
&\le 16\epsilon_x,
\end{aligned}$$

where the first step is by Eq. (5.10), the second step is by Eq. (5.12), the third step is by Eq. (5.11).

□

5.7.7 Move Both: Part 2

The goal of this section is to prove Lemma 5.7.8.

Lemma 5.7.8. *Given $\|\widetilde{\delta}_x - \delta_x\|_{H(x)} \le \epsilon_x$ and $\|\delta_x\|_{H(x)} \le \epsilon_1$. If $\epsilon_x + \epsilon_1 \le 1/4$, then we have*

$$\Phi_t(x + \delta_x, x + \widetilde{\delta}_x) \le 16 \cdot \Phi_t(x + \delta_x, x + \delta_x).$$

Proof. Since $\|x + \widetilde{\delta}_x - x\|_{H(x)} \le \|\delta_x\|_{H(x)} + \|\widetilde{\delta}_x - \delta_x\|_{H(x)} \le \epsilon_x + \epsilon_1 \le 1/4$, by Lemma 5.5.9 we have

$$\frac{1}{4}H(x) \preceq H(x + \widetilde{\delta}_x) \preceq 4H(x).$$

Since $\|x + \delta_x - x\|_{H(x)} \le \epsilon_1 \le 1/4$, by Lemma 5.5.9 we have

$$\frac{1}{4}H(x) \preceq H(x + \delta_x) \preceq 4H(x).$$

Therefore,

$$\begin{aligned}\Phi_t(x+\delta_x, x+\widetilde{\delta}_x) &= \|\nabla f_t(x+\delta_x)\|_{H(x+\widetilde{\delta}_x)^{-1}} \\ &\leq 4\|\nabla f_t(x+\delta_x)\|_{H(x)^{-1}} \\ &\leq 16\|\nabla f_t(x+\delta_x)\|_{H(x+\delta_x)^{-1}} \\ &= 16\Phi_t(x+\delta_x, x+\delta_x),\end{aligned}$$

where the first step is by definition of Φ (Definition 5.5.7), the second step follows from $\frac{1}{4}H(x) \preceq H(x+\widetilde{\delta}_x) \preceq 4H(x)$, the third step follows by $\frac{1}{4}H(x) \preceq H(x+\delta_x) \preceq 4H(x)$, the last step is by definition of Φ (Definition 5.5.7). □

5.7.8 Move Both: Part 3

The goal of this section is to prove Lemma 5.7.9.

Lemma 5.7.9.

$$\Phi_t(x+\delta_x, x+\delta_x) \leq \Phi_t(x,x)^2$$

Proof. For simplicity, we define

$$x^{\mathrm{tmp}} := x+\delta_x \quad \text{and} \quad z := (H(x^{\mathrm{tmp}}))^{-1}\nabla f_t(x^{\mathrm{tmp}}).$$

First, we have

$$\begin{aligned}\nabla f_t(x^{\mathrm{tmp}}) &= t\cdot c + g(x^{\mathrm{tmp}}) \\ &= -H(x)\cdot\delta_x - g(x) + g(x^{\mathrm{tmp}}) \\ &= -\sum_{i\in[m]}\frac{a_i a_i^\top}{s_i(x)^2}\delta_x + \sum_{i\in[m]}\frac{a_i}{s_i(x)} - \sum_{i\in[m]}\frac{a_i}{s_i(x^{\mathrm{tmp}})} \\ &= \sum_{i\in[m]}\left(\frac{a_i a_i^\top}{s_i(x)s_i(x^{\mathrm{tmp}})}\cdot\delta_x - \frac{a_i a_i^\top}{s_i(x)^2}\cdot\delta_x\right) \\ &= \sum_{i\in[m]}\frac{a_i\left(a_i^\top\delta_x\right)^2}{s_i(x)^2 s_i(x^{\mathrm{tmp}})}, \end{aligned} \tag{5.13}$$

where the first step is by definition of f_t (Definition 5.5.5), the second step is by $H(x)\cdot\delta_x = -t\cdot c - g(x)$, the third step is by definition of $g(x)$ and $H(x)$ (Definition 5.5.4), and the fourth and last step are

by

$$s_i(x^{\mathrm{tmp}}) - s_i(x) = s_i(x+\delta_x) - s_i(x) = a_i^\top (x + \delta_x - x) + (b - b) = a_i^\top \delta_x.$$

As a result,

$$\begin{aligned}
\|z\|_{x^{\mathrm{tmp}}}^2 &= z^\top H(x^{\mathrm{tmp}}) z \\
&= z^\top \nabla f_t(x^{\mathrm{tmp}}) \\
&= \sum_{i\in[m]} \frac{z^\top a_i \left(a_i^\top \delta_x\right)^2}{s_i(x)^2 s_i(x^{\mathrm{tmp}})} \\
&\leq \Big(\sum_{i\in[m]} \frac{\left(z^\top a_i\right)^2}{s_i(x^{\mathrm{tmp}})^2} \Big)^{1/2} \cdot \sum_{i\in[m]} \frac{\left(a_i^\top \delta_x\right)^2}{s_i(x)^2} \\
&= (z^\top H(x^{\mathrm{tmp}}) z)^{1/2} \cdot (\delta_x^\top H(x) \delta_x) \\
&= \|z\|_{x^{\mathrm{tmp}}} \cdot \|\delta_x\|_x^2,
\end{aligned}$$

where the third step is by Eq. (5.13), the fourth step follows from Cauchy-Schwarz inequality, the fifth step is by definition of $H(x)$ (Definition 5.5.4).

Therefore, we have that

$$\Phi_t(x+\delta_x, x+\delta_x) = \|z\|_{x^{\mathrm{tmp}}} \leq \|\delta_x\|_x^2 = \Phi_t(x,x)^2.$$

□

5.8 Pass Complexity of IPM

The goal of this section is to prove Lemma 5.8.1.

Lemma 5.8.1 (Pass complexity of IPM)**.** *Consider Algorithm 3 with input $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, $x_{\mathrm{start}} \in \mathbb{R}^n$, $t_{\mathrm{start}} \in \mathbb{R}$, $t_{\mathrm{final}} \in \mathbb{R}$. Let $x \in \mathbb{R}^n$ be any feasible point. Define $H(x) \in \mathbb{R}^{n\times n}$ and $s(x) \in \mathbb{R}^m$ as the Hessian and slack variables (Def. 5.5.4, Def.5.5.3). Suppose we have the following conditions*

C.1 $\sum_{i\in[m]} a_i/s_i(x)$ can be computed in one pass in $f(n)$ space;

C.2 the Hessian $H(x) \in \mathbb{R}^{n\times n}$ can be decomposed into

$$H(x) = L_G(x) + D(x),$$

where $L_G(x) \in \mathbb{R}^{n\times n}$ is the Laplacian of some graph G with edge weight w and $D(x) \in \mathbb{R}^{n\times n}$ is diagonal matrix;

C.3 we can read the edge-weight pair (e, w_e) in one pass, and we can read the diagonal of $D(x) \in \mathbb{R}^{n\times n}$ in one pass.

Then Algorithm 3 can be implemented in the streaming model with $\widetilde{O}(n) + f(n)$ space and $\widetilde{O}(\sqrt{m})$ passes.

Proof. Now consider the iterating part (Line 13-28). First we show $\nabla f_t(x) \in \mathbb{R}^n$ can be computed in one pass.

By Definition 5.5.5,

$$\nabla f_t(x) = c \cdot t - \sum_{i\in[m]} a_i / s_i(x),$$

By condition C.1, $\nabla f_t(x) \in \mathbb{R}^n$ can be computed in one pass in $f(n)$ space.

Since we have Conditions C.2 and C.3, we are able to first read D in one pass and store its entries in space $O(n)$, and then apply Theorem 5.9.9 to show $\widetilde{\delta}_x$ can be computed in $\widetilde{O}(1)$ pass and $\widetilde{O}(n)$ space.

All other computation can be done in $\widetilde{O}(n)$ space.

Since there are totally $T = O(\sqrt{m}\log(m/\epsilon_{\text{ipm}}))$ iterations, the totally number of passes used is $\widetilde{O}(\sqrt{m})$. Since we can reuse the space for x and t, the total space used is $\widetilde{O}(n) + f(n)$. □

5.9 SDD Solver in the Streaming Model

In this section, we present an SDDM solver in the streaming model. Later in §5.12 we reduces the problem of solving an SDD_0 system to solving an SDDM system, giving an SDD_0 solver in the streaming model.

5.9.1 SDD and Laplacian Systems

We need the definition of a spectral sparsifier, and a streaming algorithm for computing it.

Definition 5.9.1 (δ-spectral sparsifier)**.** *Given a weighted undirected graph G and a parameter $\delta > 0$, an edge-reweighted subgraph H of G is an* δ-spectral sparsifier *of G if*[9]

$$(1-\delta)\cdot x^\top L_G x \le x^\top L_H x \le (1+\delta)\cdot x^\top L_G x, \quad \forall x \in \mathbb{R}^n,$$

[9]We also say L_H is an δ-spectral sparsifier of L_G.

where L_G and L_H are the Laplacians of G and H, respectively.

Lemma 5.9.2 ([95])**.** *Let G be a weighted graph and $\delta \in (0,1)$ be a parameter. There exists a streaming algorithm that takes G as input, uses $\delta^{-2} n \operatorname{poly}(\log n)$ space and 1 pass, and outputs a weighted graph H with $\delta^{-2} n \operatorname{poly}(\log n)$ edges such that with probability at least $1 - 1/\operatorname{poly}(n)$, H is a δ-spectral sparsifier of G.*

We will use the classic SDD solver in the sequential model, which is formally described below.

Theorem 5.9.3 ([149])**.** *There is an algorithm which takes input an SDD_0 matrix A, a vector $b \in \mathbb{R}^n$, and a parameter $\epsilon \in (0, 1/2)$, if there exists $x^* \in \mathbb{R}^n$ such that $Ax^* = b$, then with probability $1 - 1/\operatorname{poly}(n)$, the algorithm returns an $x \in \mathbb{R}^n$ such that $\|x - x^*\|_A \leq \epsilon \cdot \|x^*\|_A$ in*

$$\operatorname{nnz}(A) \cdot \operatorname{poly}(\log n) \cdot \log(1/\epsilon)$$

*time. The returned x is called an ϵ-*approximate solution *to the SDD_0 system $Ax = b$.*

5.9.2 The Preconditioner

To prove that our SDDM solver (Algorithm 4) gives the desired accuracy, we need the concept of a *preconditioner* (and how to compute the preconditioner of an SDDM matrix).

We define preconditioner as follows:

Definition 5.9.4 (Preconditioner)**.** *For any positive definite matrix $A \in \mathbb{R}^{n \times n}$ and accuracy parameter $\delta > 0$, we say P is a δ-precondioner if*

$$\|P^{-1}Ax - x\|_A \leq \delta \cdot \|x\|_A, \quad \forall x \in \mathbb{R}^n.$$

Lemma 5.9.5. *Let A be an SDDM matrix (Definition 5.3.3) and let $A = L_G + D$ where L_G is a Laplacian matrix of graph G and $D \succ 0$ is a diagonal matrix with non-negative entries.[10] For any $\delta \in (0, 1/2)$, if H is a δ-spectral sparsifier of G, and we define $P := L_H + D$. Then P satisfies the following two conditions:*

- *P is a (2δ)-preconditioner(Definition 5.9.4) of A,*
- *$P \succ 0$.*

[10] By Fact 5.3.4, such decomposition always exists.

Proof. The lemma clearly holds for $x = \mathbf{0}_n$, so assume not in the following. By Definition 5.9.1,

$$(1+\delta)x^\top L_G x \geq x^\top L_H x \geq (1-\delta)x^\top L_G x \geq 0,$$

then $L_H \succeq 0$ as it must be symmetric. Since $D \succ 0$, we have that $A \succ 0$, $P \succ 0$, and

$$(1+\delta)x^\top (L_G + D)x \geq x^\top (L_H + D)x \geq (1-\delta)x^\top (L_G + D)x > 0,$$

which is $(1+\delta)A \succeq P \succeq (1-\delta)A$. So we obtain

$$\frac{1}{1-\delta}A^{-1} \succeq P^{-1} \succeq \frac{1}{1+\delta}A^{-1},$$

which, by $\delta \in (0, 1/2)$, implies

$$(1+2\delta)A^{-1} \succeq P^{-1} \succeq (1-2\delta)A^{-1}. \tag{5.14}$$

Since all matrices in Eq. (5.14) are positive definite, we can left multiply $A^{1/2}$ and right multiply $A^{1/2}$ on Eq. (5.14) to get

$$(1+2\delta)I \succeq A^{1/2}P^{-1}A^{1/2} \succeq (1-2\delta)I. \tag{5.15}$$

Subtracting I from Eq. (5.15) then left multiplying $x^\top$ and right multiplying x, we obtain

$$2\delta x^\top x \geq x^\top \underbrace{(A^{1/2}P^{-1}A^{1/2} - I)}_{:=M} x \geq -2\delta x^\top x. \tag{5.16}$$

Since Eq. (5.16) holds for any $x \in \mathbb{R}^n$ and M is symmetric, using spectral theorem, we get

$$2\delta \geq \lambda_n(M) \geq \lambda_1(M) \geq -2\delta.$$

Next, we have that

$$\|Mx\|_2 = \max\{|\lambda_n(M)|, |\lambda_1(M)|\} \leq 2\delta\|x\|_2.$$

Let $y := A^{-1/2}x$. As $\text{rank}(A^{-1/2}) = n$, we get that for any $y \in \mathbb{R}^n$,

$$2\delta\|A^{1/2}y\|_2 \geq \|MA^{1/2}y\|_2 = \|A^{1/2}P^{-1}Ay - A^{1/2}y\|_2. \tag{5.17}$$

Finally, rewriting both sides of Eq. (5.17) by the definition of matrix norm, we obtain

$$2\delta\|y\|_A \geq \|P^{-1}Ay - y\|_A$$

for any $y \in \mathbb{R}^n$, giving the lemma. □

5.9.3 An Iterative Solver

In this section, we present a streaming SDDM solver (Algorithm 4) that takes $\widetilde{O}(n)$ space and $O(\log(1/\epsilon)/\log\log n)$ passes for approximately solving SDDM system $Ax = b$ with error parameter $\epsilon \in (0, 1/10)$.

Algorithm 4 StreamLS($A = L_G + D, b, \epsilon$). Return an ϵ-approx solution to $A^{-1}b$

procedure StreamLS($A = L_G + D \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, $\epsilon \in \mathbb{R}$)
  ▷ Note that L_G is from input stream. D is diagonal matrix stored.
  $\delta \leftarrow 1/\log n$
  $\widetilde{\delta} \leftarrow \delta/2$
  $T \leftarrow O(\max\{1, (\log(1/\epsilon))/\log\log n\})$ ▷ Number of iterations
  Compute and store a δ-spectral sparsifier H of L_G ▷ Use one pass, Lemma 5.9.2
  Compute a δ-preconditioner of A as $P := L_H + D$
  $r_0 \leftarrow b$, $x_0 \leftarrow \mathbf{0}_n$ ▷ $r_0, x_0 \in \mathbb{R}^n$
  for $t \leftarrow 0$ to $T - 1$ **do**
    Compute a $\widetilde{\delta}$-approximate solution y_t to $Py = r_t$ by an SDD_0 solver and store y_t in the memory ▷ Theorem 5.9.3
    $r_{t+1} \leftarrow r_t - Ay_t$

    $x_{t+1} \leftarrow x_t + y_t$
  end for
  return x_T.
end procedure

5.9.4 An Iterative Solver: the Space

The goal of this section is to prove Lemma 5.9.6.

Lemma 5.9.6. *Let $A = L_G + D \in \mathbb{R}^{n\times n}$ be an SDDM matrix where $L_G \in \mathbb{R}^{n\times n}$ is the Laplacian matrix of graph G with weight w, and $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix. If we can read all edge-weight pairs (e, w_e) in one pass, and if we can read the diagonal of matrix D in one pass, then*

StreamLS(A, b, ϵ) *takes* $O(\max\{1, \log(1/\epsilon)/\log\log n\})$ *passes and* $\widetilde{O}(n)$ *space.*[11]

Proof. By Lemma 5.9.2 and $\delta = 1/\log n$, the δ-spectral sparsifier H has $\widetilde{O}(n)$ edges and can be computed in 1 pass and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$. Therefore, computing the δ-preconditioner P also takes 1 pass and $\widetilde{O}(n)$ space. Note that $P \succ 0$ and thus the system $Py = r_t$ always has a solution.

It remains to prove that each iteration of Lines 9-14 takes 1 pass and $\widetilde{O}(n)$ space. Since any iteration t only needs the vectors subscripted by t and $t+1$, we can reuse the space such that the total space is $\widetilde{O}(n)$.

Since $\operatorname{nnz}(P) = \widetilde{O}(n)$ and

$$\widetilde{\delta} = \delta/2 = 1/(2\log n),$$

in Line 10, by Theorem 5.9.3, with probability $1 - 1/\operatorname{poly}(n)$ a $\widetilde{\delta}$-approximate solution y_t can be found in $\widetilde{O}(n\log(1/\widetilde{\delta})) = \widetilde{O}(n)$ time, and therefore in $\widetilde{O}(n)$ space. Note that this step does not read the stream.

In Line 11, computing r_{t+1} requires computing Ay_t, which is done by reading the stream of L_G and D for 1 pass and multiplying the corresponding entries and adding up to the corresponding coordinate. All vectors are in $\mathbb{R}^n$, so the total space used in each iteration is $\widetilde{O}(n)$. The lemma follows immediately from

$$T = O(\max\{1, \log(1/\epsilon)/\log\log n\})$$

and a union bound. □

5.9.5 An Iterative Solver: the Accuracy

The goal of this section is to prove Lemma 5.9.7 (which shows that the solver iteratively improves the accuracy of the solution).

Lemma 5.9.7. *Let* $\delta \in (0, 1/10)$. *For any* $t \in [0, T]$, $\|x_t - A^{-1}b\|_A \leq (4\delta)^t \cdot \|A^{-1}b\|_A$.

Proof. The proof is by an induction on t. In the basic case of $t = 0$, we have $x_t = 0$ so the statement clearly holds. Assuming the lemma holds for t, we prove the inductive step for $t+1$.

[11] The algorithm can be implemented in the standard RAM model with finite precision by introducing an $\widetilde{O}(1)$ factor in the encoding, which translates to a multiplicative factor of $\widetilde{O}(1)$ in the space [150].

Since y_t is a $\widetilde{\delta}$-approximate solution (Line 10, Algorithm 4), by Theorem 5.9.3 we have

$$\|y_t - P^{-1}r_t\|_P \leq \widetilde{\delta} \cdot \|P^{-1}r_t\|_P.$$

By definition of the matrix norm, this becomes

$$(y_t - P^{-1}r_t)^\top P(y_t - P^{-1}r_t) \leq \widetilde{\delta}^2 \cdot (P^{-1}r_t)^\top P(P^{-1}r_t). \tag{5.18}$$

Since P is a δ-preconditioner, assuming $y_t \neq P^{-1}r_t$ and applying Definition 5.9.1 on both sides of Eq. (5.18), we have

$$(1-\delta) \cdot (y_t - P^{-1}r_t)^\top A(y_t - P^{-1}r_t) \leq (1+\delta)\widetilde{\delta}^2 \cdot (P^{-1}r_t)^\top A(P^{-1}r_t). \tag{5.19}$$

If $y_t = P^{-1}r_t$, then Eq. (5.19) also holds since the right-hand side of Eq. (5.19) is non-negative due to $A \succeq 0$. Note that Eq. (5.19) implies

$$\begin{aligned}\|y_t - P^{-1}r_t\|_A &\leq \widetilde{\delta}\sqrt{(1+\delta)/(1-\delta)} \cdot \|P^{-1}r_t\|_A \\ &\leq 2\widetilde{\delta} \cdot \|P^{-1}r_t\|_A \\ &\leq \delta \cdot \|P^{-1}r_t\|_A, \end{aligned} \tag{5.20}$$

where the second step follows from $(1+\delta)/(1-\delta) \leq 4$ when $\delta \in (0, 1/2)$, and the last step follows from $\widetilde{\delta} = \delta/2$.

Before continuing, we observe the following, which easily follows from the update rule and an induction on t:

$$r_t = r_0 - A\sum_{i=0}^{t-1} y_i = r_0 - A(x_t - x_0) = b - Ax_t. \tag{5.21}$$

Using Eq. (5.21), we bound the left-hand side of Eq. (5.20) from above by the following:

$$\begin{aligned}
&\|y_t - P^{-1}r_t\|_A \\
&\leq \delta \cdot \|P^{-1}(b - Ax_t)\|_A \\
&= \delta \cdot \|P^{-1}A(A^{-1}b - x_t) - (A^{-1}b - x_t) + (A^{-1}b - x_t)\|_A \\
&\leq \delta \cdot \left(\|P^{-1}A(A^{-1}b - x_t) - (A^{-1}b - x_t)\|_A + \|A^{-1}b - x_t\|_A\right) \\
&\leq \delta \cdot \left(2\delta\|A^{-1}b - x_t\|_A + \|A^{-1}b - x_t\|_A\right) \\
&= 2\delta \cdot \|A^{-1}b - x_t\|_A, \qquad (5.22)
\end{aligned}$$

where the third step follows from the triangle inequality, the fourth step follows from Lemma 5.9.5, the last step follows from $\delta \in (0, 1/2)$.

Finally, we are ready to prove the inductive step:

$$\begin{aligned}
\|x_{t+1} - A^{-1}b\|_A &= \|x_t - A^{-1}b + y_t\|_A \\
&= \|(x_t - A^{-1}b + P^{-1}r_t) + (y_t - P^{-1}r_t)\|_A \\
&\leq \|x_t - A^{-1}b + P^{-1}r_t\|_A + \|y_t - P^{-1}r_t\|_A \\
&= \|x_t - A^{-1}b + P^{-1}b - P^{-1}Ax_t\|_A + \|y_t - P^{-1}r_t\|_A \\
&\leq \|P^{-1}A(A^{-1}b - x_t) - (A^{-1}b - x_t)\|_A + 2\delta \cdot \|A^{-1}b - x_t\|_A \\
&\leq 2\delta \cdot \|A^{-1}b - x_t\|_A + 2\delta \cdot \|A^{-1}b - x_t\|_A \\
&\leq (4\delta)^{t+1} \cdot \|A^{-1}b\|_A,
\end{aligned}$$

where the first step follows from $x_{t+1} = x_t + y_t$, the third step follows from the triangle inequality, the fourth step follows from Eq. (5.21), the fifth step follows from Eq. (5.22), the sixth step follows from Lemma 5.9.5, and the last step follows from the induction hypothesis, completing the proof. □

5.9.6 Main Result

The goal of this section is to prove Theorem 5.9.9

Lemma 5.9.8. *Let $\epsilon \in (0, 1)$ and $b \in \mathbb{R}^n$. Let $A = L_G + D \in \mathbb{R}^{n \times n}$ be an SDDM matrix where $L_G \in \mathbb{R}^{n \times n}$ is the Laplacian matrix of graph G with weight w, and $D \in \mathbb{R}^{n \times n}$ is a diagonal matrix. If we can read all edge-weight pairs (e, w_e) in one pass, and if we can read the diagonal of matrix D in one pass, then with probability $1 - 1/\operatorname{poly}(n)$, StreamLS$(A, b, \epsilon)$ returns an ϵ-approximate*

solution x, i.e.,

$$\|x - A^{-1}b\|_A \leq \epsilon \cdot \|A^{-1}b\|_A.$$

in $O(\max\{1, \log(1/\epsilon)/\log\log n\})$ passes and $\widetilde{O}(n)$ space.

Proof. It follows by Lemma 5.9.6, our choices of δ, T, and Lemma 5.9.7. □

By Lemma 5.9.8 and the reduction in §5.12, we obtain our main result.

Theorem 5.9.9. *There is a streaming algorithm which takes input an SDD_0 matrix A, a vector $b \in \mathbb{R}^n$, and a parameter $\epsilon \in (0,1)$, if there exists $x^* \in \mathbb{R}^n$ such that $Ax^* = b$, then with probability $1 - 1/\operatorname{poly}(n)$, the algorithm returns an $x \in \mathbb{R}^n$ such that $\|x - x^*\|_A \leq \epsilon \cdot \|x^*\|_A$ in $O(\max\{1, \log(1/\epsilon)/\log\log n\})$ passes and $\widetilde{O}(n)$ space.*

5.10 Minimum Vertex Cover

The goal of this section is to prove Lemma 5.10.1 (Correctness) and Lemma 5.10.4 (Pass complexity).

5.10.1 Algorithm

Algorithm 5 Minimum vertex cover

procedure MinimumVertexCover($G = (V_L, V_R, E), b_1 \in \mathbb{Z}^m, c_1 \in \mathbb{Z}^n, n, m$) ▷ Lemma 5.10.1
  Let A_1 be the signed edge-vertex incident matrix of G with direction V_L to V_R
  $L \leftarrow$ the bit complexity of A_1, b_1, c_1
  Modify A_1, b_1, c_1 (add constraints $x_{V_L} \geq 0$ and $x_{V_R} \leq 0$) to get A_2, b_2, c_2
  Modify A_2, b_2, c_2 according to Lemma 5.10.2 and get A_3, b_3, c_3 ▷ $A_2 = A_3$, $b_2 = b_3$
  $x_{\text{init}} \leftarrow 2^L \cdot (\mathbf{1}_{V_L} - \mathbf{1}_{V_R})$
  $t_{\text{init}} \leftarrow 1$
  $c_{\text{init}} \leftarrow -\nabla g(x_{\text{init}})$
  $\epsilon_\Phi \leftarrow 1/100$
  $t_1 \leftarrow \epsilon_\Phi \cdot (m 2^{4L+10})^{-1}$
  $x_{\text{tmp}} \leftarrow$ InteriorPointMethod($A_3, b_3, c_{\text{init}}, t_{\text{init}}, x_{\text{init}}, t_1, 0, \epsilon_\Phi/4$)
  $t_2 \leftarrow nm 2^{3L+10}$
  $x \leftarrow$ InteriorPointMethod($A_3, b_3, c_3, t_1, x_{\text{tmp}}, t_2, \epsilon_\Phi, 1$)
  Use Lemma 5.10.2 to turn x into a set S of tight constraints on LP of (A_1, b_1, c_1)
  ▷ $S \subseteq [m + 2n]$
  return $S \cap [m]$
end procedure

5.10.2 Correctness of Algorithm 5

The goal of this section is to prove Lemma 5.10.1.

Lemma 5.10.1 (Correctness of Algorithm 5)**.** *In Algorithm 5, let A be signed edge-vertex incident matrix of G with direction V_L to V_R. Let $[n] = V_L \cup V_R$. Let $b = b_1$ and $c = c_1$ be the input. If the linear programming*

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \ & c^\top x \\ \text{s.t. } & Ax \geq b \\ & x_v \geq 0, \forall v \in V_L \\ & x_v \leq 0, \forall v \in V_R \end{aligned} \tag{5.23}$$

is both feasible and bounded, then with probability at least $1/2$, Algorithm 5 returns a set of tight constraints on some optimal solution to the LP Eq.(5.23)*.*

Proof. First we need to show these two calls into InteriorPointMethod satisfy the initial condition that $x_{\mathrm{start}}, t_{\mathrm{start}}$ is a good start point.

The first call In the first call (Line 11), we have

$$\|t_{\mathrm{init}} \cdot c_{\mathrm{init}} + \nabla g(x_{\mathrm{init}})\|_{H(x_{\mathrm{init}})^{-1}} = \|0\|_{H(x_{\mathrm{init}})^{-1}} = 0 \leq \epsilon_\Phi/4,$$

where the first step is by definition of c_{init}.

Since the algorithm InteriorPointMethod ends up in parameter x_{tmp} and t_1, by Lemma 5.7.3, we have

$$\|t_1 \cdot c_{\mathrm{init}} + \nabla g(x_{\mathrm{tmp}})\|_{H(x_{\mathrm{tmp}})^{-1}} \leq \epsilon_\Phi/4. \tag{5.24}$$

Note that t_1 and x_{tmp} is the input to the second call.

The second call In the second call (Line 13), the desired term can be upper bounded by

$$\begin{aligned} & \|t_1 \cdot c_3 + \nabla g(x_{\mathrm{tmp}})\|_{H(x_{\mathrm{tmp}})^{-1}} \\ \leq & \|t_1 \cdot c_{\mathrm{init}} + \nabla g(x_{\mathrm{tmp}})\|_{H(x_{\mathrm{tmp}})^{-1}} + \|t_1 \cdot c_{\mathrm{init}} - t_1 \cdot c_3\|_{H(x_{\mathrm{tmp}})^{-1}} \\ \leq & \epsilon_\Phi/4 + t_1 \cdot \|c_{\mathrm{init}} - c_3\|_{H(x_{\mathrm{tmp}})^{-1}} \end{aligned}$$

where the first step is by triangle inequality, the second step is from Eq. (5.24).

Now let's bound the term $\|c_{\text{init}} - c_3\|_{H(x_{\text{tmp}})^{-1}}$.

$$\begin{aligned}
\|c_{\text{init}} - c_3\|_{H(x_{\text{tmp}})^{-1}} &\leq \|c_{\text{init}} - c_3\|_2 \cdot \lambda_{\min}(H(x_{\text{tmp}}))^{-1/2} \\
&\leq \|c_{\text{init}} - c_3\|_2 \cdot 2^{L+3} \\
&\leq (\|c_{\text{init}}\|_2 + \|c_3\|_2) \cdot 2^{L+3} \\
&\leq (\sqrt{m}2^{L+3} + \|c_3\|_2) \cdot 2^{L+3} \\
&\leq (\sqrt{m}2^{L+3} + m2^{3L+4}) \cdot 2^{L+3} \\
&\leq m2^{4L+8},
\end{aligned}$$

where the first step is by $\|x\|_H = \sqrt{x^\top Hx} \leq \|x\|_2 \cdot \sqrt{\lambda_{\max}(H)}$ and $\lambda_{\max}(H) = \lambda_{\min}(H^{-1})^{-1}$, the second step is by Lemma 5.10.3, the fourth step is by $c_{\text{init}} = -\nabla g(x_{\text{init}})$, $x_{\text{init}} = 2^L(\mathbf{1}_{V_L} - \mathbf{1}_{V_R})$ and the definition of g (Def. 5.5.4), the fifth step is by definition of c_3 (in Lemma 5.10.2).

By our choice of $t_1 := \epsilon_\Phi \cdot (m2^{4L+10})^{-1}$ (Line 10), we finally have

$$\|t_1 \cdot c_3 + \nabla g(x_{\text{tmp}})\|_{H(x_{\text{tmp}})^{-1}} \leq \epsilon_\Phi/4 + \epsilon_\Phi/4 \leq \epsilon_\Phi.$$

Let OPT be the optimal solution to the linear programming

$$\min_{x \in \mathbb{R}^n, A_3 x \geq b_3} c_3^\top x.$$

By $t_2 := nm2^{3L+10}$ and Lemma 5.7.1, we know our solution x (Line 13) is feasible and has value $c_3^\top x - \text{OPT} \leq \frac{m}{t_2} \cdot (1 + 2\epsilon_\Phi) \leq n^{-1}2^{-3L-2}$. Now, we can apply Lemma 5.10.2 to show that after one matrix vector multiplication, with probability at least $1/2$, we can output the tight constraints S of a basic feasible optimal solution of

$$\min_{x \in \mathbb{R}^n, A_2 x \geq b_2} c_2^\top x. \tag{5.25}$$

Note that this LP is exactly the same as Eq. (5.23) by our construction on A_2, b_2, c_2 (Line 4).

□

Lemma 5.10.2 (Lemma 43 of [113]). *Given a feasible and bounded linear programming*

$$\begin{aligned} \min_{x \in \mathbb{R}^n} & \ c^\top x \\ \text{s.t.} & \ Ax \geq b, \end{aligned} \tag{5.26}$$

where $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$, $c \in \mathbb{Z}^n$ all having integer coefficient. Let L be the bit complexity of Eq. (5.26).

Let $r \in \mathbb{Z}^n$ be chose uniformly at random from the integers $\{-2^{L+1}n, \cdots, 2^{L+1}n\}$. Consider the perturbed linear programming

$$\begin{aligned} \min_{x \in \mathbb{R}^n} & \ (2^{2L+3} n \cdot c + r)^\top x \\ \text{s.t.} & \ Ax \geq b. \end{aligned} \tag{5.27}$$

Then with probability at least $1/2$ over the randomness on $r \in \mathbb{Z}^n$, we have the following.

Let OPT *be defined as the optimal value of linear programming Eq.* (5.27). *Let x be any feasible solution for Eq.* (5.27) *with objective value less than* $\mathrm{OPT} + n^{-1}2^{-3L-2}$, *then we can find the tight constraints of a basic feasible optimal solution of Eq.* (5.26) *using one matrix vector multiplication with A. Moreover, we have $\|x - x^*\|_\infty \leq 1/n$, where x^* is the unique optimal solution for Eq.* (5.27). *Plus, the bit complexity of Eq.* (5.27) *is at most $3L + \log(8n)$.*

Lemma 5.10.3. *Let $H(x)$ be defined as in Lemma 5.10.6, then we have*

$$\lambda_{\min}(H(x)) \geq 2^{-2L-6}.$$

Proof. We can lower bound $\lambda_{\min}(H(x))$ in the following sense,

$$\begin{aligned} \lambda_{\min}(H(x)) &\geq \lambda_{\min}(D(x)) \\ &\geq \min_{i \in [n]} D(x)_{i,i} \\ &\geq 2^{-2L-6} \end{aligned}$$

where the first step is by Lemma 5.10.6 that $H(x) = L(x) + D(x)$ and both $L(x)$ and $D(x)$ are positive semi-definite matrices, the second step is by the fact that $D(x)$ is diagonal matrix, the third step is by Lemma 5.10.6 that $D(x)_{i,i} = s_{i+m}(x)^{-2} + s_{i+m+n}(x)^{-2}$ and $s(x) := Ax - b$ is bounded by $\|s\|_\infty \leq 2^{L+3}$. □

5.10.3 Pass Complexity of Algorithm 5

Lemma 5.10.4 (Pass complexity of Algorithm 5)**.** *Suppose there's an oracle $\mathcal{I}$ running in $f(n)$ space that can output $(b_1)_i$ given any $i \in [m]$. If the input satisfies $\|b\|_\infty, \|c\|_\infty$ is polynomially bounded, then Algorithm 5 can be implemented in the streaming model within $\widetilde{O}(n) + f(n) + |S|$ space and $\widetilde{O}(\sqrt{m})$ passes.*

Proof. In Line 2, we defined A_1 but never explicitly compute and store A_1 in memory.

In Line 3, we let $L \leftarrow O(\log n)$ as the upper bound on the bit complexity. This cost one pass. Indeed, Lemma 5.3.5 implies that $|\mathrm{d}_{\max}(A_1)| \leq 1$. So L is the upper bound on the bit complexity.

In Line 4 and Line 5, we defined $A_2, b_2, c_2, A_3, b_3, c_3$. This doesn't involve computation.

In Lines 6, 7, 8, 9, 10, 12, they are done locally in memory.

In Line 11 and Line 13, we will show the following three things in order to call Lemma 5.8.1.

1. $\sum_{i\in[m]} a_i/s_i(x)$ can be computed in one pass in $h(n)$ space;
2. the Hessian $H(x) \in \mathbb{R}^{n\times n}$ can be decomposed into

$$H(x) = L_G(x) + D(x),$$

   where $L_G(x) \in \mathbb{R}^{n\times n}$ is the Laplacian of some graph G with edge weight w and $D(x) \in \mathbb{R}^{n\times n}$ is diagonal matrix;
3. we can read the edge-weight pair (e, w_e) in one pass, and we can read the diagonal of $D(x)$ in one pass.

Part 1: By Lemma 5.10.5 and our assumption of the existence of such oracle $\mathcal{I}$, given a feasible $x \in \mathbb{R}^n$, we can output a stream of all edges $e_i = (u, v)$ together with its slack $s_i(x) = a_i^\top x - b_i$ in one pass within $O(n)+f(n)$ space. So $\sum_{i\in[m]} a_i/s_i(x)$ can be computed in one pass in $h(n) = O(n)+f(n)$ space.

Part 2: This follows by Lemma 5.10.6.

Part 3: By Lemma 5.10.6, the edge weight is $w_i = s_i(x)^{-2}$, and $D(x)_{i,i} = s_{i+m}(x)^{-2} + s_{i+m+n}^{-2}$. So Part 3 is satisfied by Lemma 5.10.5.

By Lemma 5.8.1, Line 11 and Line 13 can be implemented in streaming model within $\widetilde{O}(n)+f(n)$ space and $\widetilde{O}(\sqrt{m})$ passes.

In Line 14, by Lemma 5.10.2, it can be done in one matrix multiplication with A_3, which can be done in one pass and $O(n)$ space.

Outputting set S requires $|S|$ space. □

Lemma 5.10.5 (Output slack stream)**.** *Suppose there is an oracle $\mathcal{I}$ running in $f(n)$ space that can output b_i given any $i \in [m]$. Given a feasible $x \in \mathbb{R}^n$, we can output a stream of all edges $e_i = (u, v)$ together with its slack $s_i(x) = a_i^\top x - b_i$ in one pass within $O(n) + f(n)$ space.*

Proof. Without loss of generality, we assume every edge has its unique id $i \in [m]$ and we receive this id in the stream[12]. We read one pass of all edges. Upon receiving edge $e_i = (u, v)$, we output $x_u - x_v - b_i$ where we use $\mathcal{I}$ to calculate b_i. □

5.10.4 Building Blocks

Lemma 5.10.6 (Hessian is SDDM matrix)**.** *Let the graph $G = (V, E)$ be the input of Algorithm 5. Let $n = |V|$, $m = |E|$. Let $A_3 \in \mathbb{R}^{(m+n)\times n}$, $b_3 \in \mathbb{R}^{m+n}$ be defined as in Line 5 (Algorithm 5).*

Let $x \in \mathbb{R}^n$ be any feasible point, i.e. $A_3 x > b_3$. Let $s(x) \in \mathbb{R}^{m+n}$ and $H(x) \in \mathbb{R}^{n\times n}$ be the slack and hessian defined w.r.t. A_3 and b_3 (Def. 5.5.4). Then $H(x) \in \mathbb{R}^{n\times n}$ can be written as

$$H(x) = L(x) + D(x),$$

where $L(x) \in \mathbb{R}^{n\times n}$ is the Laplacian matrix of graph G with edge weight $\{s_1(x)^{-2}, \cdots, s_m(x)^{-2}\}$, $D(x) \in \mathbb{R}^{n\times n}$ is the diagonal matrix with each diagonal entry $D(x)_{i,i} = s_{i+m}(x)^{-2}$.

Proof. Let $A_1 \in \mathbb{R}^{m\times n}$ be signed edge-vertex incident matrix of G with direction V_L to V_R, then $A_3 \cdot x \geq b_3$ can be written as

$$\begin{bmatrix} A_1 \\ I_{V_L} - I_{V_R} \end{bmatrix} \cdot x \geq \begin{bmatrix} b_1 \\ 0 \end{bmatrix},$$

where $I_{V_L} \in \mathbb{R}^{n\times n}$ denotes the diagonal matrix with

$$I_{i,i} = \begin{cases} 1, & \text{if } i \in V_L; \\ 0, & \text{otherwise.} \end{cases}$$

[12]Actually we can remove this assumption by observing that our oracle Algorithm 2 also works for $i \in [n^2]$, in which we give every edge $e = (u, v)$ an index $\mathrm{id}(e) = (u \cdot n + v) \in [n^2]$.

and $I_{V_R} \in \mathbb{R}^{n\times n}$ denotes the diagonal matrix with

$$I_{i,i} = \begin{cases} 1, & \text{if } i \in V_R; \\ 0, & \text{otherwise.} \end{cases}$$

Denote $S \in \mathbb{R}^{m\times m}$ the diagonal matrix with each diagonal entry $S_{i,i} := s_i(x)$. Thus $H(x)$ can be written as

$$\begin{aligned} H(x) &= A_1^\top S^{-2} A_1 + \sum_{i=m+1}^{m+n} e_i e_i^\top \cdot s_i(x)^{-2} \\ &= L(x) + D(x), \end{aligned}$$

where the first step is by definition of $H(x)$ (Def. 5.5.4), the second step is by definition of $L(x)$ and $D(x)$. □

5.10.5 A Minimum Vertex Cover Solver

Note that Algorithm 5 is actually a high-accuracy fractional minimum vertex cover solver for general graph (not necessarily bipartite graph), since we do not use the bipartite property of matrix A in IPM.

Theorem 5.10.7. *Let G be a graph with n vertices and m edges. Consider the* fractional minimum vertex cover *problem Eq.* (5.29) *in which every edge e needs to be covered at least b_e times. Let $x^* \in \mathbb{R}^n$ be the optimal solution of Eq.* (5.29)*. There exists a streaming algorithm (Algorithm 5) such that for any $\delta > 0$, it can output a feasible vertex cover $x \in \mathbb{R}^n$ such that*

$$\mathbf{1}_n^\top x \leq \mathbf{1}_n^\top x^* + \delta$$

in $\widetilde{O}(\sqrt{m}) \cdot \log(1/\delta)$ passes and $\widetilde{O}(n) \cdot \log(1/\delta)$ space with probability $1 - 1/\operatorname{poly}(n)$.

Proof. The proof follows directly from the proof of Lemma 5.10.1 and Lemma 5.10.4. □

As a byproduct, we obtain a fast semi-streaming algorithm for (exact, integral) minimum vertex cover in bipartite graph.

Theorem 5.10.8. *Given a bipartite graph G with n vertices and m edges, there exists a streaming algorithm that computes a minimum vertex cover of G in $\widetilde{O}(\sqrt{m})$ passes and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$.*

Proof. Because G is bipartite, by Theorem 5.3.5, all extreme points of the polytope of LP (5.29) are integral. Call Algorithm 5 to solve the perturbed LP in Lemma 5.10.2. Since LP (5.29) is feasible and bounded, the high-accuracy solution obtained from IPM can be rounded to the optimal integral solution, which in total takes $\widetilde{O}(\sqrt{m})$ passes and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$. □

5.11 Combine

In this section, we give our main algorithm that combines all previous subroutines to give our final Theorem 5.11.1.

Theorem 5.11.1 (Main theorem, formal version of Theorem 5.1.1)**.** *Given a bipartite graph G with n vertices and m edges, there exists a streaming algorithm that computes a maximum weighted matching of G in $\widetilde{O}(\sqrt{m})$ passes and $\widetilde{O}(n)$ space with probability $1 - 1/\operatorname{poly}(n)$.*

Proof. By combining Lemma 5.11.2 and Lemma 5.11.3. □

5.11.1 Algorithms

Algorithm 6 is our main algorithm. Basically, it did $O(\log n)$ calls to boost the success probability. In each call, it first prepares the isolation oracle from §5.4 and then passes it to the IPM solver.

Algorithm 6 MAIN($G = (V_L, V_R, E, w)$)

procedure MAIN($G = (V_L, V_R, E, w)$) ▷ Theorem 5.11.1
  ▷ w denotes edge weights that are integers.
  $n \leftarrow |V_L| + |V_R|$, $m \leftarrow |E|$
  $M \leftarrow \emptyset$ ▷ M is maximum matching
  for $i = 1 \to O(\log n)$ **do**
    $\overline{b} \leftarrow$ ISOLATION(m, n^n) ▷ $\overline{b} \in \mathbb{Z}^m$, Algorithm 1, Lemma 5.4.2
    $b \leftarrow \overline{b} + n^{10} \cdot w$
    $c \leftarrow \mathbf{1}_{V_L} - \mathbf{1}_{V_R}$
    $S \leftarrow$ MINIMUMVERTEXCOVER(G, b, c, n, m) ▷ $S \subseteq [m]$, Algorithm 5
    if $|S| \leq n$ **then**
      Let M' be maximum weighted matching found in edge set S
      if $w(M') > w(M)$ **then** ▷ Update maximum weighted matching
        $M \leftarrow M'$
      end if
    end if
  end for
  return M
end procedure

5.11.2 Pass Complexity

The goal of this section is to bound the pass number (Lemma 5.11.2).

Lemma 5.11.2 (Pass complexity)**.** *Given a bipartite graph with n vertices and m edges, Algorithm 6 can be implemented such that it runs in $\widetilde{O}(\sqrt{m})$ passes in streaming model in $\widetilde{O}(n)$ space.*

Proof. In Line 6, 7, we actually do not explicitly calculate $\overline{b}$ and b and stored them in memory. We instead use an oracle $\mathcal{I}$ stated in Lemma 5.4.1 that gives b_i bits by bits. So this step does not cost space.

In Line 8, we calculate and store $c \in \mathbb{Z}^n$ in $\widetilde{O}(n)$ space.

In Line 9, we call MinimuVertexCover. In order to use Lemma 5.10.4, we need to prove the following properties.

1. By Lemma 5.4.2, $\|\bar{b}\|_\infty \leq n^7$ so $\|b\|_\infty \leq n^{10}$. And $\|c\|_\infty = 1$. So both $\|b\|_\infty$ and $\|c\|_\infty$ is polynomially bounded.

2. By Lemma 5.4.1, there is an oracle $\mathcal{I}$ that uses $O(\log(n^n) + \log(m)) = \widetilde{O}(n)$ space that can output $\overline{b}_i$. Since each edge e_i comes with its weight w_i in the stream, we can output $b_i = \overline{b}_i + n^{10} w_i$ when given $i \in [m]$.

3. We can always assume $|S| \leq n$ since otherwise we will never enter Line 10.

By applying Lemma 5.10.4, this call can be done in $\widetilde{O}(n)$ space and $\widetilde{O}(\sqrt{m})$ passes.

In Line 11, since we are finding maximum matching in a graph with n vertices and n edges, we can store them in memory, then check all the 2^n possible sets of edges. This cost $\widetilde{O}(n)$ space without any pass. We find the set of edges that is a matching and has the maximum weight.

With $O(\log n)$ iterations overhead (Line 5), the number of passes blow up by an $O(\log n)$ factor. So overall, we used $\widetilde{O}(n)$ space and $\widetilde{O}(\sqrt{m})$ passes. □

5.11.3 Correctness

The goal of this section is to prove the correctness of our algorithm (Lemma 5.11.3).

Lemma 5.11.3 (Correctness)**.** *Given a bipartite graph G with n vertices and m edges. With probability at least $1 - 1/\operatorname{poly}(n)$, Algorithm 6 outputs one maximum matching.*

Proof. Consider the linear programming Eq. (5.28). According to part 1 of Lemma 5.11.7, with probability at least $1/4$, there is a unique solution y^* to Eq. (5.28).

By Lemma 5.10.1, with probability at least $1/2$ the algorithm MinimumVertexCover successfully returns a subset S of tight constraints which corresponds to an optimal solution x^*, s^* on dual

problem Eq. (5.29) (note that the linear programming in Lemma 5.10.1 and Eq. (5.29) only differ in signs). This means $s_i^* = 0$ if and only if $i \in S$. According to part 1 and part 2 of Lemma 5.11.7, $|S| \leq n$ and $y_i^* = 0$ for all $i \notin S$. Therefore, there exists a maximum matching using only edges in S, and we will find it in Line 11.

Overall, in each iteration of the for loop (Line 5), with probability at least $(1/2) \cdot (1/4) = 1/8$ we can find a maximum matching. After $O(\log n)$ loops, we can find a maximum matching with probability at least $1 - 1/\operatorname{poly}(n)$. □

5.11.4 Primal to Dual

This section tells the preliminary stuff for proving correctness Lemma 5.11.3. We give Theorem 5.11.6, which is later used in Lemma 5.11.7.

Definition 5.11.4 (Maximum weighted matching)**.** *Given a bipartite graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. Let $A \in \{0,1\}^{m \times n}$ be the unsigned edge-vertex incident matrix. Given weight $b \in \mathbb{Z}^m$ on every edge, the maximum weighted matching can be written as the following linear programming:*

$$\begin{aligned} \textbf{Primal} \quad & \max_{\mathbf{y} \in \mathbb{R}^{\mathbf{m}}} b^\top y \\ & \text{s.t. } A^\top y \leq \mathbf{1}_n \\ & y \geq 0 \end{aligned} \tag{5.28}$$

Its dual form is

Definition 5.11.5 (Fractional minimum vertex cover)**.** *Let $A \in \mathbb{Z}^{m \times n}$, $b \in \mathbb{Z}^m$ be defined as in Def. 5.11.4. The dual form of Eq.* (5.28) *is*[13]

$$\begin{aligned} \textbf{Dual} \quad & \min_{\mathbf{x} \in \mathbb{R}^{\mathbf{n}}} \mathbf{1}_n^\top x \\ & \text{s.t. } Ax \geq b \\ & x \geq 0 \end{aligned} \tag{5.29}$$

Theorem 5.11.6 (Strong duality from complementary slackness [134])**.** *Let $y \in \mathbb{R}^m$ be a feasible solution to the primal Eq.* (5.28)*, and let $x \in \mathbb{R}^n$ be a feasible solution to the dual Eq.* (5.29)*. Let*

[13]The dual LP is a *generalized* version of the minimum vertex cover problem: each edge i needs to be *covered* by at least b_i times, where the case of $b = \mathbf{1}_m$ is the classic minimum vertex cover.

$s := Ax - b \in \mathbb{R}^m$. *Then* $x \in \mathbb{R}^n, s \in \mathbb{R}^m, y \in \mathbb{R}^m$ *satisfy*

$$y^\top s = 0 \quad \textit{and} \quad x^\top(\mathbf{1}_n - A^\top y) = 0$$

if and only if $x \in \mathbb{R}^n, s \in \mathbb{R}^m$ *is optimal to the dual and* $y \in \mathbb{R}^m$ *is optimal to the primal.*

5.11.5 Properties of Primal and Dual LP Solutions

The goal of this section is to prove Lemma 5.11.7.

Lemma 5.11.7 (Properties of LP solutions)**.** *Given a bipartite graph G with n vertices and m edges. Let $w \in \mathbb{N}^m$ be edge weight. Let $\overline{b} \in \mathbb{Z}^m$ be the output of* Isolation(m, Z) *where $Z := n^n$ (Algorithm 1). Let $b := \overline{b} + n^{10} \cdot w \in \mathbb{Z}^m$. Let $A \in \{0,1\}^{m \times n}$ be edge-vertex incident matrix of G (unsigned). Consider the linear programming in Eq.* (5.28) *and Eq.*(5.29) *with parameter A and b. With probability at least* $1/4$*, we have*

1. *There is a unique solution $y^* \in \mathbb{R}^m$ to the primal LP (Eq.* (5.28)*). Furthermore, $y^* \in \{0,1\}^m$, y^* is the maximum candidate matching of G;*
2. *Let $x^* \in \mathbb{R}^n$, $s^* \in \mathbb{R}^m$ be optimal solution to the dual LP (Eq.*(5.29)*). Then we have the following properties on s^*.*
   (a) *For any $i \in [m]$, if $s_i^* > 0$ then $y_i^* = 0$;*
   (b) $\|s^*\|_0 \geq m - n$

Proof. **Part 1**

Let the feasible space of y be $S := \{y \in \mathbb{R}^m \mid Ay \leq \mathbf{1}_n, y \geq 0\}$. We implicitly have that $y \leq \mathbf{1}_m$, so S is a bounded region. Let $\overline{S}$ denote all extreme points on S.

First, we argue that there is a unique extreme point in $\overline{S}$ which has the optimal solution.

By Lemma 5.3.5, we know all extreme points is integral. Since $\mathbf{0}_m \leq y \leq \mathbf{1}_m$, all extreme points are in $\{0,1\}^m$, which correspond to a matching. Because we set $b := \overline{b} + n^{10} \cdot w$ as our objective vector, we can write $\langle b, y\rangle = \langle \overline{b}, y\rangle + n^{10} \cdot w^\top y$. Since $\|\overline{b}\|_\infty \leq n^7$ by Lemma 5.4.2, the extreme point who has the optimal objective value must be a maximum weighted matching. Let $\mathcal{F}$ be all possible matchings. We have $|\mathcal{F}| \leq n^n = Z$.

By applying Lemma 5.4.2, with probability at least $1/4$, we know that there is a unique extreme point y^* in $\overline{S}$ which has the optimal solution.

Because our feasible space S is bounded, all point $y \in S \backslash \overline{S}$ can be written as a linear combination of extreme points on S. That is, if we write $\overline{S} = \{y^{(1)}, \cdots, y^{(s)}\}$ where $s := |\overline{S}|$, then all point $y \in S \backslash \overline{S}$ can be written as

$$y = \sum_{i \in [s]} a_i y^{(i)},$$

where $0 \leq a_i < 1, \forall i \in [s]$ and $\sum_{i \in [s]} a_i = 1$. Therefore, we have

$$b^\top y = \sum_{i \in [s]} a_i \cdot (b^\top y^{(i)}) < \max_{i \in [s]} b^\top y^{(i)}.$$

So y^* is actually the unique optimal solution among all points in S.

Part 2 Part (a) follows trivially from Theorem 5.11.6.

Now we prove Part (b). Assume $\|s^*\|_0 < m - n$. Let $x^* \in \mathbb{R}^n$ be any optimal dual solution that relates to s^*, i.e. $Ax^* - b = s^*$. We will show that there exist a feasible solution to the primal $y' \in \mathbb{R}^m$ such that $y' \neq y^*$, $\langle y', s^* \rangle = 0$, $\langle x^*, \mathbf{1}_n - A^\top y' \rangle = 0$. By Theorem 5.11.6, y' is also an optimal solution to the primal LP, contradicting with the uniqueness of y^*. (In fact, we will prove that such y''s are infinitely many.)

Consider the following linear system

$$y_i = 0, \ \forall i \in [m] \text{ such that } s_i^* \neq 0 \tag{5.30}$$

$$A^\top y = A^\top y^*. \tag{5.31}$$

In constraint Eq. (5.30) there are less than $\|s^*\|_0 < m - n$ equalities, while in constraint Eq. (5.31) there are n equalities. So if we write the above system in the matrix form $\widetilde{A} y = \widetilde{c}, \ y \geq \mathbf{0}_m$, it must be that $\text{rank}(\widetilde{A}) \leq \|s\|_0 + n < m$. We obtain

$$y = \widetilde{A}^\dagger \widetilde{c} + (I - \widetilde{A}^\dagger \widetilde{A}) z,$$

where $z \in \mathbb{R}^m$ is a free variable. Since $\text{rank}(\widetilde{A}^\top \widetilde{A}) \leq \text{rank}(\widetilde{A}) < m$, it must be $I_m - \widetilde{A}^\dagger \widetilde{A} \neq \mathbf{0}_{m \times m}$. Observe that $f(z) := \widetilde{A}^\dagger \widetilde{c} + (I - \widetilde{A}^\dagger \widetilde{A}) z$ is an affine function passing through point y^*. Also note that $y^* \geq \mathbf{0}_m$ and $y^* \neq \mathbf{0}_m$ (otherwise $y^* = \mathbf{0}_m$ then we can increase an arbitrary component of y^* to 1 to increase $b^\top y$, contradicting with the optimality). As a result, $f(z)$ must pass through

infinitely many points in the subspace $y \geq \mathbf{0}_m$. Let y' be any such solution. By Eq. (5.30), we have

$$\langle y', s^* \rangle = 0$$

By Eq.(5.31) and Theorem 5.11.6, we have

$$\langle x^*, \mathbf{1}_n - A^\top y' \rangle = \langle x^*, \mathbf{1}_n - A^\top y \rangle = 0$$

By Theorem 5.11.6, y' is also an optimal solution, contradicting with the uniqueness of y^*. □

5.12 Solver Reductions

5.12.1 From SDDM$_0$ Solver to SDD$_0$ Solver

We recall Gremban's reduction in [149] that reduces the problem of sovling an SDD$_0$ system to solving an SDDM$_0$ system. Let A be an SDD$_0$ matrix, decompose A into $D + A_{\text{neg}} + A_{\text{pos}}$, where D is the diagonal of A, A_{neg} contains all the negative off-diagonal entries of A with the same size, and A_{pos} contains all the positive off-diagonal entries of A with the same size. Consider the following linear system

$$\widehat{A} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \widehat{b},$$

where

$$\widehat{A} = \begin{bmatrix} D + A_{\text{neg}} & -A_{\text{pos}} \\ -A_{\text{pos}} & D + A_{\text{neg}} \end{bmatrix} \quad \text{and} \quad \widehat{b} = \begin{bmatrix} b \\ -b \end{bmatrix}$$

The matrix $\widehat{A}$ can be (implicitly) computed in the streaming model: in one pass we compute and store the diagonal matrix D by adding the edge weights incident on each vertex; then $\widehat{A}$ is given as a stream of edges (entries) since whenever an edge (an entry in A) arrives, we immediately know its position in $\widehat{A}$. Note that if $Ax = b$ admits a solution, then $x = (x_1 - x_2)/2$ is exactly its solution.

Moreover, if

$$\left\| \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \widehat{A}^\dagger \widehat{b} \right\|_{\widehat{A}} \leq \epsilon \|\widehat{A}^\dagger \widehat{b}\|_{\widehat{A}},$$

then x satisfies $\|x - A^\dagger b\|_A \leq \epsilon \|A^\dagger b\|_A$. So we obtain an SDD_0 solver with asymptotically the same number of passes and space as an SDD_0 sovler.

5.12.2 From SDDM solver to SDDM_0 solver

In this section, we show that to approximately solve an SDDM_0 system $Ay = b$, it suffices to preprocess the input in $O(1)$ passes, approximately solve an SDDM system $\widetilde{A}y = \widetilde{b}$ with at most the same size, and possibly do some post-process in $O(1)$ passes.

If A is positive definite, then we can solve the system by an SDDM solver, so assume not in the following.

From Fact 5.3.4 we know that A must be a Laplacian matrix. Therefore, it remains to reduce the problem of (approximately) solving a Laplacian system $Ly = b$ to (approximately) solving an SDDM system. The following facts are well-known.

Fact 5.12.1. *Given a Laplacian matrix L corresponding to graph G, the following holds:*

- $L \succeq 0$*;*
- $L\mathbf{1}_n = 0$ *and thus* $\lambda_1(L) = 0$*;*
- G *is connected iff* $\lambda_2(L) > 0$*.*

Given a Laplacian matrix L as a stream of entries, it is equivalent to treat it as a stream of edges of G. In one pass, we can identify all the connected components of G using $\widetilde{O}(n)$ space (e.g., by maintaining the spanning forest of G). Next, any entry in the stream is identified and assigned to the subproblem corresponding to the connected component that contains it.[14] This does not influence the worst-case pass and space complexity, because each subproblem uses space proportional to the size of its connected component and the total number of passes depends on the connected component that takes up the most passes. Therefore, we can assume that G is connected, which implies that $\mathrm{rank}(L) = n - 1$ by Fact 5.12.1.

The goal of approximately solving $Ly = b$ is for given error parameter $\epsilon > 0$, finding an ϵ-approximate solution x satisfying

$$\|x - L^\dagger b\|_L \leq \epsilon \|L^\dagger b\|_L.$$

[14]The above process is equivalent to partition L into block diagonal matrices, solve each linear system with respect to the submatrices and corresponding entries of b, and combine the result.

If y is an (exact) solution to system $Ly = b$, then $y' := y - y_1 \mathbf{1}_n$ is also a solution, where y_1 is the first entry of y. So we can assume that the first entry of $L^\dagger b$ is 0. (There might be many solutions, but we fix one with the first entry being 0.) Let $\widetilde{A}$ be the matrix L with the first row and column deleted, and let $\widetilde{b}$ be the vector b with the first entry deleted. Note that $\widetilde{A} \succ 0$.

Let $\widetilde{x}$ be an ϵ-approximate solution to the system $\widetilde{A}x = \widetilde{b}$, and let x be the vector $\widetilde{x}$ with 0 inserted as its first entry. It must be that

$$\|x - L^\dagger b\|_L = \|\widetilde{x} - \widetilde{A}^{-1}\widetilde{b}\|_{\widetilde{A}}$$

because vector $\widetilde{A}^{-1}\widetilde{b}$ is the vector $L^\dagger b$ with the first entry deleted.

Finally, we have that $\|x - L^\dagger b\|_L \leq \epsilon \|L^\dagger b\|_L$ since

$$\|\widetilde{x} - \widetilde{A}^{-1}\widetilde{b}\|_{\widetilde{A}} \leq \epsilon \|\widetilde{A}^{-1}\widetilde{b}\|_{\widetilde{A}} \quad \text{and} \quad \|L^\dagger b\|_L = \|\widetilde{A}^{-1}\widetilde{b}\|_{\widetilde{A}},$$

which gives an ϵ-approximate solution to the original system $Ly = b$.

Bibliography

[1] Kook Jin Ahn and Sudipto Guha. Graph sparsification in the semi-streaming model. In *International Colloquium on Automata, Languages, and Programming*, pages 328–338. Springer, 2009.

[2] Kook Jin Ahn and Sudipto Guha. Laminar families and metric embeddings: Non-bipartite maximum matching problem in the semi-streaming model. *CoRR*, abs/1104.4058, 2011.

[3] Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. In *ICALP*, pages 526–538. Springer, 2011.

[4] Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, pages 526–538, 2011.

[5] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):1–40, 2018.

[6] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Trans. Parallel Comput.*, 4(4):17:1–17:40, 2018.

[7] Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, New Jersey, USA, December 3-7, 1990*, pages 1–20, 1990.

[8] Selim G. Akl. *Design and analysis of parallel algorithms.* Prentice Hall, 1989.

[9] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 574–583, 2014.

[10] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 674–685, 2018.

[11] Kurt M Anstreicher. The volumetric barrier for semidefinite programming. *Mathematics of Operations Research*, 25(3):365–380, 2000.

[12] Sepehr Assadi. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 321–335, 2017.

[13] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1635. SIAM, 2019.

[14] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1616–1635, 2019.

[15] Sepehr Assadi and Aaron Bernstein. Towards a unified theory of sparsification for matching problems. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019*, volume 69, pages 11:1–11:20, 2019.

[16] Sepehr Assadi, Nikolai Karpov, and Qin Zhang. Distributed and streaming linear programming in low dimensions. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, pages 236–253, 2019.

[17] Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 3–12, 2017.

[18] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete*

Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pages 1723–1742, 2017.

[19] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1345–1364, 2016.

[20] Sepehr Assadi, Gillat Kol, Raghuvansh R. Saxena, and Huacheng Yu. Multi-pass graph streaming lower bounds for cycle counting, max-cut, matching size, and other problems. In *FOCS*, 2020.

[21] Sepehr Assadi, S. Cliff Liu, and Robert E. Tarjan. An auction algorithm for bipartite matching in streaming and massively parallel computation models. In *The SIAM Symposium on Simplicity in Algorithms (SOSA@SODA'21)*, 2021.

[22] Sepehr Assadi and Ran Raz. Near-quadratic lower bounds for two-pass graph streaming algorithms. In *FOCS*. `https://arxiv.org/pdf/2009.01161.pdf`, 2020.

[23] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 461–470, 2019.

[24] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers*, 36(10):1258–1263, 1987.

[25] Paul Beame and Johan Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[26] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284, 2013.

[27] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. *J. ACM*, 64(6):40:1–40:58, 2017.

[28] Soheil Behnezhad, Laxman Dhulipala, Hossein Esfandiari, Jakub Lacki, and Vahab S. Mirrokni. Near-optimal massively parallel graph connectivity. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1615–1636, 2019.

[29] Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially faster massively parallel maximal matching. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1637–1649, 2019.

[30] Aaron Bernstein. Improved bounds for matching in random-order streams. In *ICALP*, 2020.

[31] Aaron Bernstein. Improved bounds for matching in random-order streams. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, pages 12:1–12:13, 2020.

[32] Dimitri P Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of operations research*, 14(1):105–123, 1988.

[33] Dimitri P Bertsekas. Linear network optimization, 1991.

[34] Dimitri P Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational optimization and applications*, 1(1):7–66, 1992.

[35] Jan van den Brand. A deterministic linear program solver in current matrix multiplication time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 259–278. SIAM, 2020.

[36] Jan van den Brand, Yin Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *FOCS*, 2020.

[37] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. Solving tall dense linear programs in nearly linear time. In *STOC*, 2020.

[38] Paul Burkhardt. Graph connectivity in log-diameter steps using label propagation. *CoRR*, abs/1808.06705, 2018.

[39] Charles Carlson, Alexandra Kolla, Nikhil Srivastava, and Luca Trevisan. Optimal lower bounds for sketching graph cuts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2565–2569. SIAM, 2019.

[40] Timothy M Chan and Eric Y Chen. Multi-pass geometric algorithms. *Discrete & Computational Geometry*, 37(1):79–102, 2007.

[41] Yi-Jun Chang, Martin Farach-Colton, Tsan-Sheng Hsu, and Meng-Tsung Tsai. Streaming complexity of spanning tree computation. In *37th international symposium on theoretical aspects of computer science (STACS 2020)*, 2020.

[42] Suresh Chari, Pankaj Rohatgi, and Aravind Srinivasan. Randomness-optimal unique element isolation with applications to perfect matching and related problems. *SIAM Journal on Computing*, 24(5):1036–1050, 1995.

[43] Lijie Chen, Gillat Kol, Dmitry Paramonov, Raghuvansh R. Saxena, Zhao Song, and Hucheng Yu. Almost optimal super-constant-pass streaming lower bounds for reachability. In *STOC*, 2021.

[44] Kenneth L Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM (JACM)*, 42(2):488–499, 1995.

[45] Michael B Cohen, Rasmus Kyng, Gary L Miller, Jakub W Pachocki, Richard Peng, Anup B Rao, and Shen Chen Xu. Solving sdd linear systems in nearly $m \log^{1/2} n$ time. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing(STOC)*, pages 343–352, 2014.

[46] Michael B Cohen, Yin Tat Lee, and Zhao Song. Solving linear programs in the current matrix multiplication time. In *Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC)*, 2019.

[47] Stephen A. Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97, 1986.

[48] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, June 25-29, 2018*, pages 471–484, 2018.

[49] Samuel I Daitch and Daniel A Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the fortieth annual ACM symposium on Theory of computing (STOC)*, pages 451–460, 2008.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[51] Gabrielle Demange, David Gale, and Marilda Sotomayor. Multi-item auctions. *Journal of political economy*, 94(4):863–872, 1986.

[52] Nikhil R Devanur, Kamal Jain, and Robert D Kleinberg. Randomized primal-dual analysis of ranking for online bipartite matching. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 101–107. SIAM, 2013.

[53] Martin Dietzfelbinger, Miroslaw Kutylowski, and Rüdiger Reischuk. Exact lower time bounds for computing boolean functions on CREW prams. *J. Comput. Syst. Sci.*, 48(2):231–254, 1994.

[54] Shahar Dobzinski, Noam Nisan, and Sigal Oren. Economic efficiency requires interaction. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing (STOC)*, pages 233–242, 2014.

[55] Shahar Dobzinski, Noam Nisan, and Sigal Oren. Economic efficiency requires interaction. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 233–242, 2014.

[56] Sally Dong, Yin Tat Lee, and Guanghao Ye. A nearly-linear time algorithm for linear programs with small treewidth: A multiscale representation of robust central path. In *STOC*, 2021.

[57] Sebastian Eggert, Lasse Kliemann, Peter Munstermann, and Anand Srivastav. Bipartite matching in the semi-streaming model. *Algorithmica*, 63(1-2):490–508, 2012.

[58] Sebastian Eggert, Lasse Kliemann, Peter Munstermann, and Anand Srivastav. Bipartite matching in the semi-streaming model. *Algorithmica*, 63(1-2):490–508, 2012.

[59] Alireza Farhadi, Mohammad Taghi Hajiaghayi, Tung Mah, Anup Rao, and Ryan A Rossi. Approximate maximum matching in random streams. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1773–1785. SIAM, 2020.

[60] Alireza Farhadi, Mohammad Taghi Hajiaghayi, Tung Mai, Anup Rao, and Ryan A. Rossi. Approximate maximum matching in random streams. In *Proceedings of the 2020 ACM-SIAM*

Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020, pages 1773–1785, 2020.

[61] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *ICALP*, pages 531–543. Springer, 2004.

[62] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[63] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2009.

[64] Faith E. Fich, Friedhelm Meyer auf der Heide, Prabhakar Ragde, and Avi Wigderson. One, two, three \dots infinity: Lower bounds for parallel computation. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 48–58, 1985.

[65] François Le Gall. Powers of tensors and fast matrix multiplication. In Katsusuke Nabeshima, Kosaku Nagasaka, Franz Winkler, and Ágnes Szántó, editors, *International Symposium on Symbolic and Algebraic Computation, ISSAC '14, Kobe, Japan, July 23-25, 2014*, pages 296–303. ACM, 2014.

[66] Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 491–500, 2019.

[67] Hillel Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[68] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, July 23-27, 2018*, pages 129–138, 2018.

[69] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth*

Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019, pages 1636–1653, 2019.

[70] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 698–710, 1991.

[71] Steve Goddard, Subodh Kumar, and Jan F. Prins. Connected components algorithms for mesh-connected parallel computers. In *Parallel Algorithms, Proceedings of a DIMACS Workshop, Brunswick, New Jersey, USA, October 17-18, 1994*, pages 43–58, 1994.

[72] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 468–485. SIAM, 2012.

[73] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 468–485. SIAM, 2012.

[74] Michael T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation (preliminary version). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 711–722, 1991.

[75] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 374–383, 2011.

[76] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, pages 374–383, 2011.

[77] John Greiner. A comparison of parallel algorithms for connected components. In *SPAA*, pages 16–25, 1994.

[78] Venkatesan Guruswami and Krzysztof Onak. Superlinear lower bounds for multipass graph processing. In *2013 IEEE Conference on Computational Complexity*, pages 287–298. IEEE, 2013.

[79] Shay Halperin and Uri Zwick. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996.

[80] Shay Halperin and Uri Zwick. Optimal randomized erew pram algorithms for finding spanning forests. *Journal of Algorithms*, 39(1):1–46, 2001.

[81] Isidore Heller and CB Tompkins. An extension of a theorem of dantzig's. *Linear inequalities and related systems*, 38:247–254, 1956.

[82] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.

[83] Changwan Hong, Laxman Dhulipala, and Julian Shun. Exploring the design space of static and incremental graph connectivity algorithms on gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 55–69, 2020.

[84] John E Hopcroft and Richard M Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[85] Tsan-Sheng Hsu, Vijaya Ramachandran, and Nathaniel Dean. Parallel implementation of algorithms for finding connected components in graphs. *Parallel Algorithms: Third DIMACS Implementation Challenge, October 17-19, 1994*, 30:20, 1997.

[86] Baihe Huang, Shunhua Jiang, Zhao Song, and Runzhou Tao. Solving tall dense sdps in the current matrix multiplication time. *arXiv preprint arXiv:2101.08208*, 2021.

[87] Haotian Jiang, Tarun Kathuria, Yin Tat Lee, Swati Padmanabhan, and Zhao Song. A faster interior point method for semidefinite programming. In *FOCS*, 2020.

[88] Shunhua Jiang, Zhao Song, Omri Weinstein, and Hengjie Zhang. Faster dynamic matrix inverse for faster lps. In *STOC*, 2021.

[89] Donald B. Johnson and Panagiotis Takis Metaxas. Connected components in O (logˆ3/2 n) parallel time for the CREW PRAM. *J. Comput. Syst. Sci.*, 54(2):227–242, 1997.

[90] Sagar Kale and Sumedh Tirodkar. Maximum matching in two, three, and a few more passes over graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 15:1–15:21, 2017.

[91] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1679–1697. SIAM, 2013.

[92] Michael Kapralov. Better bounds for matchings in the streaming model. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1679–1697, 2013.

[93] Michael Kapralov, Sanjeev Khanna, and Madhu Sudan. Approximating matching size from random streams. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 734–751. SIAM, 2014.

[94] Michael Kapralov, Sanjeev Khanna, and Madhu Sudan. Approximating matching size from random streams. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 734–751, 2014.

[95] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. *SIAM J. Comput.*, 46(1):456–477, 2017.

[96] Michael Kapralov, Slobodan Mitrovic, Ashkan Norouzi-Fard, and Jakab Tardos. Space efficient approximation to maximum matching size from uniform edge samples. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1753–1772, 2020.

[97] Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, and Navid Nouri. Faster spectral sparsification in dynamic streams. In *arXiv preprint*. `https://arxiv.org/pdf/1903.12165.pdf`, 2019.

[98] Michael Kapralov, Navid Nouri, Aaron Sidford, and Jakab Tardos. Dynamic streaming spectral sparsification in nearly linear time and space. In *arXiv preprint*. `https://arxiv.org/pdf/1903.12150.pdf`, 2019.

[99] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948, 2010.

[100] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948, 2010.

[101] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing(STOC)*, pages 302–311. ACM, 1984.

[102] Jonathan A Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. In *STACS*, 2011.

[103] Jonathan A Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving sdd systems in nearly-linear time. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing (STOC)*, pages 911–920, 2013.

[104] Shahbaz Khan and Shashank K. Mehta. Depth first search in the semi-streaming model. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019, March 13-16, 2019, Berlin, Germany*, volume 126 of *LIPIcs*, pages 42:1–42:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[105] Dénes König. Über graphen und ihre anwendung auf determinantentheorie und mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916.

[106] Christian Konrad. A simple augmentation method for matchings with applications to streaming algorithms. In *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018, August 27-31, 2018, Liverpool, UK*, pages 74:1–74:16, 2018.

[107] Christian Konrad, Frédéric Magniez, and Claire Mathieu. Maximum matching in semi-streaming with few passes. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 15th International Workshop, APPROX 2012, and 16th International Workshop, RANDOM 2012, Cambridge, MA, USA, August 15-17, 2012. Proceedings*, pages 231–242, 2012.

[108] Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving sdd linear systems. In *FOCS*, pages 235–244, 2010.

[109] Ioannis Koutis, Gary L Miller, and Richard Peng. A nearly-$m \log n$ time solver for sdd linear systems. In *52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 590–598, 2011.

[110] Rasmus Kyng and Sushant Sachdeva. Approximate gaussian elimination for laplacians-fast, sparse, and simple. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 573–582. IEEE, 2016.

[111] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 85–94, 2011.

[112] Yin Tat Lee and Aaron Sidford. Path finding ii: An $\widetilde{O}(m\sqrt{n})$ algorithm for the minimum cost flow problem. *arXiv preprint arXiv:1312.6713*, 2013.

[113] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in $O(\sqrt{rank})$ iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.

[114] Yin Tat Lee and Aaron Sidford. Efficient inverse maintenance and faster algorithms for linear programming. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 230–249. IEEE, 2015.

[115] Yin Tat Lee, Zhao Song, and Qiuyi Zhang. Solving empirical risk minimization in the current matrix multiplication time. In *COLT*, 2019.

[116] S Cliff Liu, Zhao Song, and Hengjie Zhang. Breaking the n-pass barrier: A streaming algorithm for maximum weight bipartite matching. *arXiv preprint arXiv:2009.06106*, 2020.

[117] S. Cliff Liu and Robert E. Tarjan. Simple concurrent labeling algorithms for connected components. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA*, pages 3:1–3:20, 2019.

[118] Yang P Liu, Arun Jambulapati, and Aaron Sidford. Parallel reachability in almost linear work and square root depth. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1664–1686. IEEE, 2019.

[119] Yang P Liu and Aaron Sidford. Faster divergence maximization for faster maximum flow. In *FOCS*, 2020.

[120] Yang P Liu and Aaron Sidford. Faster energy maximization for faster maximum flow. In *STOC*, 2020.

[121] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 1–10, 1985.

[122] Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 253–262. IEEE, 2013.

[123] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010.

[124] Andrew McGregor. Finding graph matchings in data streams. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 170–181. Springer, 2005.

[125] Andrew McGregor. Finding graph matchings in data streams. In *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*, pages 170–181, 2005.

[126] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*, 2015.

[127] Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 478–489, 1985.

[128] Marcin Mucha and Piotr Sankowski. Maximum matchings via gaussian elimination. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 248–255. IEEE, 2004.

[129] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354, 1987.

[130] Yu Nesterov and Arkadi Nemirovsky. Self-concordant functions and polynomial-time methods in convex programming. *Report, Central Economic and Mathematic Institute, USSR Acad. Sci*, 1989.

[131] Yurii Nesterov and Arkadi Nemirovski. Conic formulation of a convex programming problem and duality. *Optimization Methods and Software*, 1(2):95–115, 1992.

[132] Yurii Nesterov and Arkadi Nemirovski. *Interior-point polynomial algorithms in convex programming*, volume 13. Siam, 1994.

[133] Krzysztof Onak. Round compression for parallel graph algorithms in strongly sublinear space. *CoRR*, abs/1807.08745, 2018.

[134] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[135] Ami Paz and Gregory Schwartzman. A $(2 + \epsilon)$-approximation for maximum weight matching in the semi-streaming model. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2153–2161, 2017.

[136] Richard Peng and Daniel A Spielman. An efficient parallel solver for sdd linear systems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing (STOC)*, pages 333–342, 2014.

[137] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 50–61, 2013.

[138] John H Reif. Optimal parallel algorithms for graph connectivity. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1984.

[139] James Renegar. A polynomial-time algorithm, based on newton's method, for linear programming. *Mathematical Programming*, 40(1-3):59–93, 1988.

[140] James Renegar. *A mathematical view of interior-point methods in convex optimization*. SIAM, 2001.

[141] Alvin E Roth and Marilda Sotomayor. Two-sided matching. *Handbook of game theory with economic applications*, 1:485–541, 1992.

[142] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *J. ACM*, 65(6):41:1–41:24, 2018.

[143] Yossi Shiloach and Uzi Vishkin. An O(log n) parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[144] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic - June 23 - 25, 2014*, pages 143–153, 2014.

[145] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. A fast GPU algorithm for graph connectivity. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Workshop Proceedings*, pages 1–8, 2010.

[146] Jyothish Soman, Kishore Kothapalli, and P. J. Narayanan. Some GPU algorithms for graph connected components and spanning tree. *Parallel Processing Letters*, 20(4):325–339, 2010.

[147] Zhao Song. *Matrix Theory : Optimization, Concentration and Algorithms*. PhD thesis, The University of Texas at Austin, 2019.

[148] Zhao Song and Zheng Yu. Oblivious sketching-based central path method for solving linear programming. In *ICML*, 2021.

[149] Daniel A. Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pages 81–90. `https://arxiv.org/abs/cs/0310051`,divided into `https://arxiv.org/abs/0809.3232`, `https://arxiv.org/abs/0808.4134`, `https://arxiv.org/abs/cs/0607105`, 2004.

[150] Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Analysis Applications*, 35(3):835–885, 2014.

[151] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining, WSDM 2018, Marina Del Rey, CA, USA, February 5-9, 2018*, pages 540–546, 2018.

[152] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[153] Robert Endre Tarjan. *Data structures and network algorithms*. SIAM, 1983.

[154] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

[155] W. Thrash. A note on the least common multiples of dense sets of integers. ., 1993.

[156] Sumedh Tirodkar. Deterministic algorithms for maximum matching on general graphs in the semi-streaming model. In *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, pages 39:1–39:16, 2018.

[157] Pravin M Vaidya. An algorithm for linear programming which requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$arithmetic operations. In *FOCS*. IEEE, 1987.

[158] Pravin M Vaidya. Speeding-up linear programming using fast matrix multiplication. In *FOCS*. IEEE, 1989.

[159] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[160] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms*, 4(1):45–50, 1983.

[161] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB*, 7(14):1821–1832, 2014.

[162] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. Fastsv: A distributed-memory connected component algorithm with fast convergence. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 46–57. SIAM, 2020.